# Providing Decentralised Adaptive Services

—

# Entwicklung und Betrieb dezentraler adaptiver Dienste

Der Technischen Fakultät der

Universität Erlangen-Nürnberg

zur Erlangung des Grades

DOKTOR-INGENIEUR

vorgelegt von

**Rüdiger Kapitza** - Erlangen 2007

Als Dissertation genehmigt von
der Technischen Fakultät der
Universität Erlangen-Nürnberg

**Tag der Einreichung:** 3.07.2007

**Tag der Promotion:** 21.09.2007

**Dekan:** Prof. Dr.-Ing. Alfred Leipertz

**Berichterstatter:** Prof. Dr.-Ing. Schröder-Preikschat, Prof. Dr.-Ing. Franz J. Hauck

## Abstract

Recently, many research activities have investigated means to cope with the exponential growth of computing power, numbers of devices, and most importantly, information. The main focus of these activities is managing the complexity and optimising resource utilisation.

This thesis presents novel solutions that optimise the utilisation of distributed resources for a stable and scalable execution of long-running services. This challenge is tackled by providing development and infrastructure support for two object-oriented distributed service models named *replicated decentralised adaptive services* and *peer-to-peer based decentralised adaptive services*. Both service models avoid a central point of failure and are capable of adapting their distribution according to service and environment demands.

Replicated decentralised adaptive services are supported by a framework that enables the flexible development of fault-tolerant services and supports active replication. In contrary to related approaches the proposed solution builds on top of the fragmented object model as provided by the Aspectix middleware. This approach is a lean way of adding fault-tolerant replication to services based on standard middleware. Furthermore, the framework retains the flexibility of the fragmented object paradigm by providing semantic annotation support at the interface level to enable a custom code generation for an optimised handling of requests and the introduction of object-specific code at client-side. This flexibility enables the development of complex dependable distributed services as this thesis demonstrates by the implementation of a distributed version control system. Dynamic distribution of replicated decentralised adaptive service is enabled by an extended reference concept that ensures the reliable binding of services despite location changes. Furthermore, support for a fast state transfer of arbitrarily large service states is provided.

Peer-to-peer based decentralised adaptive services target the utilisation of distributed resources using peer-to-peer concepts in context of a standard middleware. The proposed solution integrates the JXTA peer-to-peer infrastructure into CORBA using the fragmented object support of Aspectix. Services can make use of peer-to-peer mechanisms to be more scalable and fault-tolerant than traditional client/server distributed objects. Existing CORBA applications can now transparently access such extended services.

Both service models constitute the basis for the *environment for decentralised adaptive services (EDAS)*, which enables the distribution and management of services across multiple administrative domains. As dynamic distribution might move services to nodes on which the required service-specific code is not available, EDAS offers support for dynamic loading of code. The presented loading service differs from related approaches, as it uses a decentralised peer-to-peer based lookup, selection and loading process for providing platform-specific code.

Finally, EDAS advances the state of the art in distributed resource management. The supported mechanisms include a novel decentralised and fault-tolerant protocol to enforce global resource quotas and a decentralised placement algorithm that enables a resource-aware distribution of decentralised adaptive services.

**Parts of the material presented in this thesis has previously been published as:**

1. Rüdiger Kapitza, Franz J. Hauck: DLS: A CORBA service for dynamic loading of code. In: On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE *(International Symposium on Distributed Objects and Applications (DOA) Sicily, Italy 2003)* , pages 1333-1350, Berlin, Heidelberg : Springer, 2003.

2. Hans P. Reiser, Franz J. Hauck, Rüdiger Kapitza, Andreas I. Schmied: Integrating fragmented objects into a CORBA environment. In: Net Object Days - Tagungsband *(Net.ObjectDays 2003 Erfurt, Germany 2003)*, pages 264–272, Erfurt : tranSIT GmbH, 2003.

3. Rüdiger Kapitza, Franz J. Hauck, Hans P. Reiser: Decentralized, adaptive services: The AspectIX approach for a flexible and secure grid environment. In: Grid Services Engineering and Management *(Grid Services Engineering and Management (GSEM 2004) Erfurt, Germany 2004)*, pages 107-118, Berlin, Heidelberg : Springer, 2004.

4. Franz J. Hauck, Rüdiger Kapitza, Andreas I. Schmied: A flexible and extensible object middleware: CORBA and beyond. In: SEM 2005 *(Fifth International Workshop on Software Engineering and Middleware (SEM 2005) Lisbon, Portugal 2005)*, pages 69-75, New York, USA: ACM Press, 2005.

5. Rüdiger Kapitza, Franz J. Hauck: EDAS: Providing an environment for decentralized adaptive services. In: DSM '05: Proceedings of the 2nd International Doctoral Symposium on Middleware *(2nd International Doctoral Symposium on Middleware 2005 (DSM '05) Grenoble, France, 2005)*, New York, USA: ACM Press, 2005

6. Rüdiger Kapitza, Holger Schmidt, Franz J. Hauck: Platform-independent object migration in CORBA. In: On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE *(International Symposium on Distributed Objects and Applications (DOA) Agia Napa, Cyprus 2005)*, pages 900-917, Berlin, Heidelberg : Springer, 2005.

7. Rüdiger Kapitza, Hans P. Reiser, Franz J. Hauck: Stable, time-bound references in context of dynamically changing environments. In: ICDCSW '05: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems – Workshops (ICDCS 2005 Workshops) *(ICDCS Columbus, Ohio 2005)*, pages 603-609, Washington, DC, USA : IEEE Computer Society, 2005.

8. Hans P. Reiser, Jörg Domaschka, Franz J. Hauck, Rüdiger Kapitza, Wolfgang Schröder-Preikschat: Consistent replication of multithreaded distributed objects. In: Proceedings of the IEEE Symposium on Reliable Distributed Systems *(IEEE Symposium on Reliable Distributed Systems (SRDS 2006) Leeds, UK)*, pages 257-266, Washington, DC, USA : IEEE Computer Society, 2006.

9. Rüdiger Kapitza, Meik Felser, Hans P. Reiser, Franz J. Hauck: Decentralised diffusion-based quota management. In: Proceedings of the 4th International Workshop on Middleware for Grid Computing *(in conjunction with ACM/IFIP/USENIX 7th International Middleware Conference 2006, Melbourne, Australia)*, New York, USA: ACM DL, 2006

10. Jörg Domaschka, Franz J. Hauck, Rüdiger Kapitza, Hans P. Reiser: Deterministic multithreading for replicated CORBA objects. In: Proceedings of the 18th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2006) *(PDCS 2006 Dallas, TX, USA)*, pages 516–521, IASTED, 2006.

11. Rüdiger Kapitza, Udo Bartlang, Holger Schmidt, Franz J. Hauck: Dynamic integration of peer-to-peer services into a CORBA compliant middleware. In: On the Move to Meaningful Internet Systems 2006 - CoopIS, DOA, and ODBASE *(International Symposium on Distributed Objects and Applications Montpellier, France)*, pages 28-29, Berlin, Heidelberg : Springer, 2006.

12. Rüdiger Kapitza, Franz J. Hauck: EDAS: Providing an environment for decentralized adaptive services. In: IEEE Distributed Systems Online 7(3) (2006), Nr. 0603-o3004

13. Hans P. Reiser, Rüdiger Kapitza, Jörg Domaschka, Franz J. Hauck: Fault-tolerant replication based on fragmented objects. In: Proceedings of the 6th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems *(6th IFIP International Conference on Distributed Applications and Interoperable Systems Bologna, Italy)*, pages 256-271, Berlin, Heidelberg : Springer, 2006.

14. Rüdiger Kapitza, Holger Schmidt, Udo Barlang, Franz J. Hauck: A generic infrastructure for decentralised dynamic loading of platform-specific code. In: Proceedings of the 7th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems *(7th IFIP International Conference on Distributed Applications and Interoperable Systems Phapos, Zypern)*, pages 256-271, Berlin, Heidelberg : Springer, 2007.

15. Rüdiger Kapitza, Thomas Zeman, Franz J. Hauck, Hans P. Reiser: Parallel state transfer in object replication systems. In: Proceedings of the 7th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems *(7th IFIP International Conference on Distributed Applications and Interoperable Systems Phapos, Zypern)*, pages 256-271, Berlin, Heidelberg : Springer, 2007.

# Table of Contents


**1 Introduction** **1**
1.1 Objectives . . . 2
1.2 Contributions . . . 3
1.2.1 Development Support for Decentralised Adaptive Services . . . 4
1.2.2 Infrastructure Support for Decentralised Adaptive Services . . . 6
1.3 Thesis Outline . . . 7

**2 The Aspectix Middleware** **9**
2.1 Standard CORBA Architecture . . . 10
2.2 Status Quo of Extensible Middleware . . . 11
2.2.1 Standard Customisation Points of CORBA . . . 11
2.2.2 Adaptable Middleware Platforms . . . 12
2.3 Profile Manager . . . 12
2.4 Aspectix Profile Manager . . . 15
2.5 Summary . . . 17

**3 Replicated Decentralised Adaptive Services** **19**
3.1 Object Replication Style . . . 21
3.2 Approaches to Replication Support in Distributed Object Middleware . . . 21
3.3 Fault-Tolerant Replication with Fragmented Objects . . . 23
3.3.1 Details of the Fragment Architecture . . . 24
3.3.2 Summary . . . 26
3.4 Time-Bound Object References . . . 26
3.4.1 State of the Art . . . 27
3.4.2 Basic Concept . . . 30
3.4.3 Update of Time-Bound References . . . 30
3.4.4 Secure Binding of Distributed Objects . . . 32
3.4.5 Integration of a Location Service Extension . . . 36
3.4.6 Use Case: Reference Management of a Decentralised Version Control System . . . 39
3.4.7 Related Approaches . . . 40
3.4.8 Summary . . . 41
3.5 State Transfer . . . 41
3.5.1 State of the Art . . . 42
3.5.2 State Transfer Application Interface . . . 46

3.5.3 Transfer Format . . . . . 48
3.5.4 Parallel Download Approaches . . . . . 50
3.5.5 Adaptation of Parallel Download Approaches for Parallel State Transfer 51
3.5.6 Synchronisation Approaches . . . . . 55
3.5.7 Writing Partial States . . . . . 57
3.5.8 Implementation Overview . . . . . 58
3.5.9 Experimental Evaluation . . . . . 60
3.5.10 Summary . . . . . 64
3.6 Deterministic Multithreading Support . . . . . 66
3.6.1 Deterministic Thread Scheduling . . . . . 66
3.6.2 Code Transformation Support . . . . . 67
3.6.3 Summary . . . . . 67
3.7 Semantic Annotation Support for Customised Implementation . . . . . 68
3.7.1 State of the Art . . . . . 69
3.7.2 Semantic Annotations . . . . . 70
3.7.3 Customised Code Generation and Request Handling . . . . . 71
3.7.4 Summary . . . . . 72
3.8 Run-Time Infrastructure . . . . . 72
3.9 Performance Evaluation . . . . . 74
3.9.1 Evaluation of Efficiency Gains Using Semantic Annotations . . . . . 74
3.9.2 Evaluation of Deterministic Multithreading Support . . . . . 75
3.10 Case Study: A Replicated Decentralised Version Control System . . . . . 77
3.10.1 Basic Concepts of Git . . . . . 79
3.10.2 Local Repository Operations . . . . . 80
3.10.3 Distributed Repository Operations . . . . . 81
3.10.4 Summary . . . . . 83
3.11 Summary . . . . . 84

**4 Peer-to-Peer based Decentralised Adaptive Services 85**
4.1 Related Approaches . . . . . 87
4.2 JXTA Overview . . . . . 87
4.2.1 Peers and Peer Groups . . . . . 87
4.2.2 Advertisements . . . . . 88
4.2.3 Messages and Pipes . . . . . 88
4.2.4 Resolving Resources in JXTA . . . . . 88
4.2.5 JXTA Architecture . . . . . 89
4.2.6 Summary . . . . . 90
4.3 Mapping JXTA Services on Fragmented Objects . . . . . 90
4.4 JXTA Profile Manager . . . . . 92
4.5 Case Study: Multi-User Chat Application . . . . . 92
4.6 Summary . . . . . 93

**5 Environment for Decentralised Adaptive Services 95**
5.1 State of the Art . . . 97
5.1.1 Distributed Middleware . . . 97
5.1.2 Middleware Supporting Mobile Code . . . 99
5.1.3 Distributed Deployment Platforms . . . 100
5.1.4 Summary . . . 101
5.2 Home Environment and Associated Services . . . 102
5.3 Service Environment . . . 104
5.4 Integrating Decentralised Adaptive Services . . . 105
5.5 System Overview . . . 106
5.6 Summary . . . 107

**6 Decentralised Dynamic Loading of Platform-Specific Code 109**
6.1 State of the Art . . . 110
6.2 Requirements and Properties for Implementation Selection . . . 112
6.3 Basic Infrastructure . . . 113
6.4 Data Structures of the Implementation Repository . . . 113
6.5 Basic Workflow of Publication, Selection and Loading of Code . . . 114
6.6 Dynamic Lookup and Loading of Services in JXTA . . . 115
6.6.1 Managing Lookup and Loading of Services via Advertisements . . . 115
6.6.2 Shortcomings and Weaknesses . . . 116
6.7 Decentralised Dynamic Loading Based on JXTA . . . 117
6.7.1 Extended Advertisements . . . 117
6.7.2 Decentralised Implementation Repository . . . 118
6.7.3 Code Provider . . . 118
6.7.4 Dynamic Loader . . . 119
6.8 Integration of Dynamic Loading Support into Aspectix and EDAS . . . 121
6.9 Case Study: Extended Multi-User Chat Application . . . 121
6.10 Summary . . . 123

**7 Resource Management 125**
7.1 Resources and Resource Limits . . . 126
7.2 Resource Monitoring and Accounting Infrastructure . . . 126
7.2.1 Basic Support for Resource Monitoring . . . 127
7.2.2 Node Infrastructure for Resource Monitoring . . . 127
7.3 Restricting Resource Usage using Quotas . . . 128
7.3.1 Diffusive Load Balancing . . . 129
7.3.2 Diffusion-based Quota Protocol . . . 130
7.3.3 Basic Concept . . . 130
7.3.4 Protocol Description . . . 131
7.3.5 Simulation . . . 136
7.3.6 Integrating Quota Management into EDAS . . . 138
7.3.7 Related Approaches . . . 140
7.3.8 Summary . . . 140

7.4 Decentralised Application Placement . . . . . 141
7.4.1 State of the Art . . . . . 142
7.4.2 Resource Partitioning of Nodes . . . . . 150
7.4.3 Gaining Global Knowledge about Available Resources . . . . . 151
7.4.4 Assigning Resources to Projects . . . . . 152
7.4.5 Providing Distance Information . . . . . 152
7.4.6 Placement of Services . . . . . 153
7.4.7 Summary . . . . . 155
7.5 Summary . . . . . 155

**8 Conclusion** **157**
8.1 Main Contributions . . . . . 158
8.2 Limitations and Future Work . . . . . 159

**Einleitung** **169**
Ziele der Arbeit . . . . . 171
Wissenschaftlicher Beitrag . . . . . 172
Entwicklungsunterstützung für dezentrale adaptive Dienste . . . . . 173
Infrastrukturunterstützung für dezentrale adaptive Dienste . . . . . 175
Gliederung . . . . . 176

**Zusammenfassung** **179**
Kernpunkte des wissenschaftlichen Beitrags . . . . . 180
Limitationen der Arbeit und Ausblick . . . . . 181

**Bibliography** **182**

# List of Figures


2.1 UML class diagram profile manager . . . . . 13
2.2 Example for a fragmented object . . . . . 15
2.3 Architecture of a local fragment . . . . . 16

3.1 Replication with Fragmented Objects . . . . . 24
3.2 Architectural Overview of the Framework for Replicated Decentralised Adaptive Services . . . . . 25
3.3 Basic concept of time-bound object references . . . . . 31
3.4 UML state diagram of a secure binding process . . . . . 33
3.5 Location service interface . . . . . 37
3.6 Reference version history of a time-bound object reference . . . . . 39
3.7 Basic non-blocking state transfer protocol . . . . . 45
3.8 State transfer approaches . . . . . 46
3.9 FT-CORBA interface for state transfer (IDL) . . . . . 47
3.10 Stream-based state transfer interface (Java) . . . . . 48
3.11 Parallel-access approaches . . . . . 53
3.12 Dynamic request scheme with adaptive batchsize . . . . . 55
3.13 Architecture of JGroups . . . . . 59
3.14 Message Exchange of the Non-Blocking Distributed State Transfer Protocol . . 60
3.15 State transfer in a LAN environment . . . . . 61
3.16 State transfer in a LAN environment measurements at 100 MB state size . . . . 62
3.17 State transfer in a LAN environment with load injection . . . . . 63
3.18 State transfer in a WAN environment . . . . . 64
3.19 The impact of a non-blocking state transfer on blocking time and message delay 65
3.20 Interface of the generic fragment factory . . . . . 73
3.21 Creation of first and additional replicas . . . . . 74
3.22 Efficiency gains using semantic annotations . . . . . 75
3.23 Nested invocation example . . . . . 76
3.24 Producer/Consumer and Time-bounded `wait` . . . . . 77
3.25 Example of the basic Git storage model . . . . . 80
3.26 Commit example . . . . . 80
3.27 Basic operations to manage the local repository . . . . . 82
3.28 Operations to manage and synchronise the local with the global repository . . 83

4.1 Resolving resources in the JXTA rendezvous network . . . . . 89

4.2 Chat application interface (IDL) . . . . . 93

5.1 The three core components of EDAS: home environment, service environment and decentralised adaptive service . . . . . 96
5.2 EDAS Scenario comprising three home environments supporting one service environment that hosts two services . . . . . 103
5.3 Service management interface . . . . . 105
5.4 Platform management interface . . . . . 106
5.5 EDAS system layout on a node . . . . . 107

6.1 An approach towards a generic code classification . . . . . 112
6.2 Sequence chart code sharing . . . . . 119
6.3 Collaboration of central system components . . . . . 120
6.4 Fragment implementation exchange (Java) . . . . . 122

7.1 Resource accounting and monitoring infrastructure . . . . . 128
7.2 Concept of diffusion-based quota balancing . . . . . 131
7.3 Connecting nodes with each other . . . . . 133
7.4 Simplified basic balancing process . . . . . 133
7.5 Recovering quota after neighbour crash . . . . . 134
7.6 Initiate balancing process for refundable resources . . . . . 136
7.7 Extended balancing process for non-refundable resources . . . . . 137
7.8 Simulation results . . . . . 139
7.9 Services for resource-aware distribution . . . . . 142

# 1 Introduction

Over the past few years, many research activities have been initiated to cope with the exponential growth of computing power, numbers of devices, and most importantly, information. Manageability and resource utilisation can be identified as the main research goals of these initiatives.

Manageability has various aspects — in general it should hide complexity to give users the opportunity to focus on things they like to do instead on the process of how to do them. Currently big IT companies like IBM [KC03], Sun Microsystems, HP and Microsoft try to construct self-managing computers and compute infrastructures. This includes very difficult tasks, such as the automatic detection and compensation of hard- and software failures, but also autonomous run-time optimisation and resource management of large applications in IT infrastructures.

Beside manageability, improving utilisation plays an important role. It reduces costs and offers the opportunity to solve more complex tasks with already existing resources. The most prominent example for this trend is the readvent of *peer-to-peer computing* as we know it today. Starting in 1999 with Napster, a simple file sharing protocol for music, peer-to-peer computing emerged to one of the most active research efforts in the distributed systems community since then. Apart from the public interest in trading copyright-protected music for free, peer-to-peer computing offers users the opportunity to share their spare resources with a community or a project. Prominent examples are the SETI [SWB+97] project that searches for signs of extraterrestrial intelligence and ePost a decentralised highly available email system [MPHD06].

The idea of peer-to-peer computing breaks the till then applied roles of clients as consumers and servers as information providers and gives each participant the opportunity to act as both. On the downside, this approach inherits certain risks as peer-to-peer networks face frequent and sometimes massive changes of the participating nodes. This has to be addressed by the underlying protocols, for example through massive replication and decentralisation of all system-relevant tasks. More severely than system dynamics, using the resources of all available peers implicates that all participants have the potential to attack the system [Dou02, SCRD04]. Although this is subject to ongoing research efforts, public file sharing networks and other peer-to-peer applications have already shown their usability in every day life.

Infrastructures for *grid computing* aim at the virtualising of commodity computers, servers, and storage systems as one large computing system [FKT01]. These infrastructures address utilisation by combining compute resources and manageability by supporting the autonomous scheduling and execution of compute- or data-intensive jobs. Thus resource management is a key issue, needed for an efficient and automated distribution of large sets of tasks on a grid. Such grid infrastructures are often deployed at enterprise level, but there

are also planetary-scale grids, spanning multiple organisations all over the world. Unlike peer-to-peer computing, typical grid infrastructures such as the Globus Toolkit [Fos05] or Condor [TTL02] offer the opportunity to deploy and execute large applications as big sets of short- and mid-term tasks. However, these grid systems entail the manual set up and management of numerous system components like information services, distributed schedulers or repositories for data and program code.

In parallel to the ongoing development of grid infrastructures for large compute- and data-intensive applications, there is a trend to provide general grid infrastructures for arbitrary distributed services. The main difference in the requirements for current and future grid computing infrastructures is the lifetime of the managed entities. For recent systems, the lifetime of an application is a short and bound period. In future grid systems, services will run for a long, possibly infinite time. Thus scheduling, resource management and planning support of current grid systems cannot be applied for such long-running services. Early protagonists of these systems are PlanetLab [PACR03], a planetary-scale testbed of computers for distributed services and the Xenoserver [HHaIP03] project. Both systems target to combine thousands of nodes, spread all over the world, controlled by a resource management layer that can be used to schedule, deploy and execute arbitrary services. Xenoserver, but also PlanetLab, support the execution of legacy services in virtual operating system containers, e.g., by using Xen [BDF+03]. This offers the transparent deployment and execution of existing services without modifications. However, these systems face frequent changes due to the joining and leaving of nodes, and services will be confronted with changing user demands. This enforces the repeated redistribution of services depending on the change rate of the system and user demands. Facing these changes, services have to be aware of their redistribution, thus the use of existing services without any modification is rarely possible. Thus, new kinds of infrastructure and service models have to be developed.

Albeit, first grid-like systems for arbitrary distributed services have already been developed and in the case of PlanetLab are already used; the development of these new infrastructures and services has just been started. Existing systems give only first answers, if at all, to the following questions:

- How do new service models look like?
- What is a suitable architecture for these systems?
- How to manage the resources for long-running services?
- How to dynamically deploy and redeploy services?

## 1.1 Objectives

The goal of this thesis is to address the aforementioned challenges by providing long-running services that are designed for dynamic distribution and redistribution. Thus, the infrastructure and the hosted services need to be aware of each other. Therefore, service models and framework support should be provided to ease the development of distribution-aware services. Furthermore, advanced infrastructure support is needed for their execution.

Long-running services should be aware of their distribution and be able to control it. In the context of this thesis, distribution changes are considered as a way to be *adaptive* to changing service demands and environment conditions. Service demands are caused by user access or other service-specific requirements like fault tolerance. Changing environment conditions are caused by maintenance reboots, node failure and varying resource availability in general. Both service demands and environment conditions change over time or even periodically fluctuate. Summarising, in context of this thesis services will be considered as *adaptive services* if they are distribution-aware and able to use this knowledge to adapt their distribution according to their needs and conditions posed by the executing environment.

As distributed environments always have to face node failures and as services targeted in scope of this thesis should run for a long, possibly infinite time they have to be able to tolerate a limited number of node crashes. Commonly, this is achieved using replication. Depending on the employed mechanisms, node crashes can still have an impact on the service and might cause minor data loss or a temporary service unavailability. However, in a long-term perspective the service stays accessible and operational. All kinds of services discussed throughout this thesis should be able to tolerate node failures to some extend. This requirement is summarised by the term of *decentralised services*. Such services are distributed over multiple nodes and employ replication to avoid a single point of failure.

Taking both aspects together this thesis aims at providing service models to support the development and provision of *decentralised adaptive services*. These services tolerate a limited number of node failures and are able to adapt their distribution according to requirements like available resources and user demands. This requires replication support, as well as migration support, and distribution awareness of the services.

Essentially the same demands as for services apply for their executing infrastructure; it should be decentralised and adaptive. This can be achieved by taking decentralised adaptive services as base. However, it requires extended support for the execution and management of services. An essential requirement is the dynamic provision of service code. Before a service is migrated to a new location it has to be ensured that its code is available or can be dynamically deployed at the target platform. Even more important is support for resource management that constitutes the foundation for a resource-aware service distribution. This requires resource accounting mechanisms to determine the resource demand of a service and mechanisms to restrict resource usage at the service level as well as on broader stages.

## 1.2 Contributions

As basis for the development of decentralised adaptive services the object-oriented paradigm was chosen, as objects build a well-defined mechanism to structure and develop all kinds of applications especially distributed ones. Object-oriented middleware such as the Common Object Request Broker Architecture (CORBA) [Obj04], .NET [MWN02], Java Remote Method Invocation (Java RMI) [Wal98] and ICE [Hen04] are commonly used for the implementation of complex distributed applications. These platforms provide good support for traditional client/server interaction and offer various extension points for its customisation. However,

all these middleware infrastructures miss extended support for self-adaptation and self-optimisation as well as scalability, fault-tolerance, and end-to-end quality of service in the context of high dynamics. To suit these needs of common middleware platforms the Aspectix project supports the fragmented object model [Sha86, MGNS94]. This has been done for the CORBA middleware by proving a flexible reference handling concept as demonstrated by the Aspectix Object Request Broker (ORB) [RHKS03, HKRS05] and for the Java RMI middleware by offering the FORMI extension [KKSH05, KDH+06]. Fragmented objects allow the implementation of truly distributed objects going far beyond a simple customisation of client/sever interaction [BMV03, Obj01] or the provision of smart proxies [KK00, SMS02] but even enable multiple clients to interact in peer-to-peer setting. In context of this thesis, the Aspectix ORB was chosen to implement the proposed service models and the platform for decentralised adaptive services. The fragmented object approach per se offers only basic support for the implementation of truly distributed objects. Therefore, one of the contributions of this thesis is to provide extended support for decentralised adaptive services that, basing on the integration of the fragmented object model, seamlessly integrate into common object-oriented middleware. In fact from a client-perspective, a decentralised adaptive service can be hardly distinguished form a plain client/server object. Exchanging a simple service by a decentralised adaptive one only requires the rebinding of an object reference.

### 1.2.1 Development Support for Decentralised Adaptive Services

Two kinds of decentralised adaptive services were identified:

- *replicated decentralised adaptive services*
- *peer-to-peer based decentralised adaptive services*

This distinction accounts the fact that services possess different fault-tolerance and scalability requirements in context of the targeted dynamic environments. The first service model addresses high fault-tolerance demands enabling the concurrent modification of data despite its replication. The second service model targets the scalable provision of read-only content.

#### Replicated Decentralised Adaptive Services

*Replicated decentralised adaptive services* at its core represent objects that are actively replicated [Mul93] which provides a high degree of fault tolerance. Active replication assumes that objects or services can be deterministically replicated over a set of nodes by distributing state-modifying requests in total order using a group-communication framework. Unlike former work [Maf95, MMSN98, Fel98, GNSY00, MMVB00, FH02, RBC+03] the provided framework for replicated decentralised services offers advanced mechanisms for service mobility that addresses the dynamics in a global-scale distributed execution environment and provides a flexible client/server programming model based on semantic annotations and custom code generation.

One of the problems of service mobility is to locate and reliably access a mobile service despite location changes. Usually this issue is solved by using a well know always-available

location service, or mechanisms for request forwarding or the combination of both approaches. In context of this thesis, the issue of reliably binding a service is tackled by introducing *time-bound object references*. These extended references give a timely restricted guarantee to successfully bind a service, even if the service changes its distribution. This basic concept was enhanced by a secure binding process enabling to tolerate attacks at binding time performed by nodes that previously hosted the service.

Existing object replication systems support the instantiation of new services and the replacement of crashed replicas. Both tasks require a state transfer from one operational replica to a newly instantiated one. In a traditional fault-tolerant application scenario service creation and node faults are usually considered a seldom task and therefore the same applies for state transfer. For replicated decentralised services this is no longer the case as distribution changes are expected to be more frequent or even take place on a regular basis. Furthermore, most systems [CM00] use a blocking state transfer approach that requires a complete delay of all requests during the time to acquire the state from the service and the transfer of the state to the new node. Additionally, all systems known to the author use an approach that transfers the state from only one replica to the newly joining replica. The basic assumptions behind these two facts are that state transfer is not only considered to be a seldom task but also that service states are rather small. Not only the first assumption is wrong for decentralised adaptive services but also the second assumption cannot be held in context of this thesis. Service states can be very large as this is demonstrated by an example application given by this thesis, a replicated decentralised version control system. Consequently, extended support for state transfer was developed that allows a non-blocking and parallel state transfer from multiple replicas. Both techniques and further discussed approaches drastically speed up the state transfer. A faster state transfer improves service availability due to a shorter request blocking time and it improves fault-tolerance of actively replicated services as node crashes can be compensated in a shorter time.

As earlier mentioned, the integration of the fragmented object model into a common object-oriented middleware offers the opportunity to introduce arbitrarily distributed objects that can seamlessly replace plain client/server distributed objects. If replicated decentralised adaptive services just use the fragmented object support as a basis for integration, the flexibility of the model would be lost. Such an approach can be compared to many existing fault-tolerant object replication frameworks that simply use an extended code generation process to create custom stubs and skeletons from an interface description. However, this does not fit the needs of real complex services like evaluated by the development of the replicated decentralised version control system. Despite the need for fault tolerance, many services require flexible support for arbitrary object-internal communication and interaction. This demand was accounted by the introduction of a small set of semantic annotations at the interface level that are evaluated during a custom code generation process, thereby enabling the integration of developer-supplied code at client-side.

Concluding, replicated decentralised adaptive services provide fault-tolerance based on active replication, offer an advanced support for mobility and supply a flexible way to address custom requirements concerning client-side interaction and object-internal communication. This builds a solid foundation for complex and fault-tolerant services in dynamic en-

vironments that host data, which is concurrently accessed and modified by multiple users.

#### Peer-to-Peer based Decentralised Adaptive Services

However, active replication as supported by the proposed replicated decentralised adaptive service model is not suited for massive data replication needed to provided large amounts of data that is accessed read-only by a lot of clients but only modified by its content providers. For these kinds of application demands peer-to-peer mechanisms have been put into use in recent times as they enable the utilisation of existing resources and are in most cases scalable.

Accounting this fact, this thesis proposes *peer-to-peer based decentralised adaptive services* by combining the fragmented object model with mechanisms provided by the peer-to-peer infrastructure JXTA [Gon01]. Here the contribution is the integration of a mature peer-to-peer middleware into a CORBA-compliant environment and vice versa to supplement JXTA with the benefits of an object-oriented middleware. Peer-to-peer based decentralised adaptive services provide a lightweight and flexible approach that can be ported to any other common middleware that supports the fragmented object approach. Peer-to-peer mechanisms as provided by the JXTA middleware offer replication of routing information and support the replication of functionality and data according to user demands. Thus, peer-to-peer based decentralised adaptive services are fault tolerant and adaptive.

### 1.2.2 Infrastructure Support for Decentralised Adaptive Services

Taking both decentralised adaptive service models and optionally combine their concepts, object-oriented services for dynamic environments addressing almost all purposes can be implemented. Basing on this support, a grid-like infrastructure for long-running decentralised adaptive services named *Environment for Decentralised and Adaptive Services (EDAS)* was developed. Two organisational and architectural concepts have been identified named *home environment*, representing a domain and *service environments*, managing the execution of multi-domain spanning projects.

When coping with decentralised adaptive services and fragmented objects in general, the object binding usually requires the dynamic loading of code, as it is not feasible to pre-install all available implementations at every node of a distributed system. The reason is that some nodes would only use the code and these nodes are usually not known in advance. This is even more an issue if the set of participating nodes changes over time like it is expected for EDAS and decentralised adaptive services. To enable the dynamic and flexible set-up of new services EDAS supports the decentralised dynamic loading of platform-specific code. Although this support is integrated into the EDAS environment, it can be used as a standalone infrastructure for code loading. Unlike former work of the author [KH03] the presented approach is decentralised and provides an extended selection process to find a suitable implementation. In contrast to related approaches [PKF05, SM05, PC03] it focuses not on a specific implementation language or platform and is decentralised.

As outlined by the objectives section, resource management is an essential part of an execution platform for distributed services. This especially is the case for EDAS and its

support for decentralised adaptive services. Distributed resource management primarily requires resource accounting at the node level. Thus, this thesis provides a node-wide accounting infrastructure using existing support of the current Sun Java Virtual Machine (Sun JVM) version 1.5. Basing on the proposed accounting infrastructure, a decentralised quota protocol to enforce global project-wide resource limits is described. In contrast to traditional approaches (e.g., *leases* [GC89]) this protocol is scalable, fault-tolerant, and reactive to demand changes. In context of EDAS it enables the management of consumable resources like network transmission volume. Finally, decentralised adaptive services should be able to adapt their distribution according to resource requirements and resource availability. Consequently, a protocol for a resource- and distance-aware online placement of distributed and possibly replicated services that fits especially the needs of decentralised adaptive services is proposed. Unlike existing work [SH03, OAPA05] in the area of open and dynamic systems, the presented approach addresses admission control through assignment of predefined resource sets.

Subsuming, an infrastructure is provided that enables the distributed execution of decentralised adaptive services extended by the support for dynamic loading of code and support for resource management including a resource aware distribution of services.

## 1.3 Thesis Outline

Chapter 2 gives a brief overview about the main features of the Aspectix middleware namely the flexible handling of remote object references and the support of the fragment object model. These features of the Aspectix middleware constitute the basis for supporting the two proposed decentralised adaptive service models.

Chapter 3 outlines replicated decentralised adaptive services. Starting with a digest of related approaches the general framework and infrastructure for fault-tolerant replication based on fragmented objects is described. This provides the foundation for introducing extended mechanisms to support service migration like time-bound object references and techniques or fast state transfer. To enable the flexible and efficient implementation of replicated decentralised adaptive services the support for deterministic multithreading and the semantic annotation at the interface level is outlined. To set up and execute replicated services a minimal infrastructure is described. The chapter closes with a performance evaluation and an extended case study that covers the implementation of a decentralised version control system.

Chapter 4 starts with an overview of the JXTA middleware and then describes the integration of JXTA services as fragmented objects into the Aspectix middleware, thereby, providing the peer-to-peer based decentralised adaptive service model. Finally, a multi-user chat application evaluates the presented service model.

Chapter 5 describes the conceptual architecture of EDAS and outlines its realisation using the two proposed service models. This builds the basis for executing and distributing decentralised adaptive services across multiple domains.

Chapter 6 targets the dynamic deployment of software not only for decentralised adaptive services but also for distributed applications in heterogeneous environments in general.

Starting from general considerations how code should be described to enable a fast and scalable selection process in a distributed environment this chapter leads to an infrastructure featuring dynamic selection, loading, and provision of platform-specific code in context of EDAS.

Chapter 7 addresses the resource management of EDAS to enable a resource-aware distribution of decentralised adaptive services. The chapter starts with providing some general information about resources and at which level their usage has to be managed. In a next step, a node-wide infrastructure for resource management is outlined. Basing on this support a decentralised quota management protocol and a protocol for resource-aware application placement is proposed.

Chapter 8 concludes this thesis, by subsuming the main contributions and briefly outlining current limitations at a conceptual level but also at the level of implementation; finally it gives an overview about possible future work.

# 2 The Aspectix Middleware

Many complex distributed applications have been implemented by using one of the common object-oriented middleware platforms such as CORBA, .NET, Java RMI and ICE. Despite the fact that these infrastructures offer mature development support for applications that can be implemented using a traditional client/server interaction they miss sufficient support for non-standard requirements. Examples are extended support for self-adaptation and self-optimisation as well as scalability, fault-tolerance, and end-to-end quality of service in the context of high system dynamics. To account these needs of future distributed applications custom middleware platforms have been developed (see Section 2.2.2). However, these systems are usually special purpose solutions and similar support has to be time-consumingly reimplemented if ported to another platform. The Aspectix project has chosen a different approach by integrating the fragmented object model into common object-oriented middleware. Originally proposed by Shapiro [Sha86, MGNS94] this object model enables the implementation of truly distributed objects going far beyond a simple customisation of client/sever interaction as provided by current middleware platforms. It is even superior to the smart proxy [KK00, SMS02] approach, as it enables multiple clients to interact in a peer-to-peer setting (see Chapter 4). The integration of the fragmented object model into a common middleware offers all benefits of a mature platform and gives the opportunity of arbitrary extensions to fit non-standard application demands. The feasibility of the integration has been verified in context of the CORBA middleware by the Aspectix ORB and by the FORMI extension for Java RMI. For other platforms like .NET a similar approach is possible. Fragmented objects thereby can seamlessly replace plain distributed client/server objects by simply rebinding an object reference. To some extend this follows the observations and reasoning of Dickman, Makpangou and Shapiro [DMS92] that proposed the combination of the fragmented object model and the uniform object references.

The support for fragmented objects as offered by the Aspectix ORB but also by the FROMI extension is very basic. Meaning both platforms provide a lean object skeleton for fragmented objects and offer the instantiation and binding of fragmented objects. Thus, fragmented objects so far have to be implemented more or less manually. However, fragmented objects constitute an ideal basis for the implementation of higher support offered by custom frameworks or the integration of existing ones. For this thesis, the Aspectix ORB was chosen as an implementation basis for the development support of decentralised adaptive services. FORMI offers a similar good basis but was not yet available by the time of the start of this work. Furthermore, FORMI is limited to the Java platform whereas the Aspectix ORB and its concepts can be ported to any CORBA-supported language.

The Aspectix ORB is CORBA-compliant and more extensible compared to standard CORBA ORBs. This is achieved by a modularisation of the handling of interoperable object references (IORs). A CORBA object reference consist of a general reference part and a variable

number of profiles. These profiles enable to include contact addresses of the object. The Aspecitx ORB offers a generic reference manager with portable profile managers. These profile managers thereby encapsulate all tasks related to reference handling, i.e., reference creation, reference marshalling and unmarshalling, external representation of references as strings, and type casting of representatives of remote objects. This provides the necessary foundation for supporting fragmented objects. Currently, profile managers for standard CORBA, for the fragmented object model and other non-CORBA middleware platforms, like Jini [Mic01] and Java RMI are provide. The Aspectix ORB is based on the open-source CORBA implementation JacORB [Bro97] but as outlined in [HKRS05] only small modifications to existing CORBA implementations are necessary to support such a design. Thus, the principle approach to support a generic reference handling infrastructure and consequently the support of fragmented objects can be ported within weeks to other ORB implementations.

This thesis utilises the reference handling mechanisms of the ORB to provide replicated decentralised adaptive services on top of the basic support for the fragmented object model. Additionally, a new profile manager is presented in Chapter 4 that handles references for peer-to-peer based decentralised adaptive services.

The necessary background for this work is briefly introduced by providing an overview about the CORBA object model and handling of CORBA object references. This is followed up by a discussion about CORBA-compliant extension points and the state of the art of non-compliant middleware extensions to fit custom application demands. Finally, the Aspectix profile manager and the support for the fragmented object model is presented.

## 2.1 Standard CORBA Architecture

CORBA uses a *monolithic object model*: CORBA objects reside on a specific node and are transparently accessed by local proxies called *stubs*. The stubs use a remote procedure call (RPC)-based communication protocol to contact the actual object, to pass parameters, and to receive results from object invocations. A CORBA-based middleware implementation is free to choose the actual protocol, but has to support the Internet Inter-ORB Protocol (IIOP) for interoperability.

CORBA uses IORs to address remote objects. The IOR is a data structure composed of a set of profiles. Each profile may specify contact information of the remote object for one specific interaction protocol; for interoperability between ORBs of different vendors, an IIOP profile is mandatory. In addition to protocol profiles, the IOR may contain a set of tagged components. Each tagged component is a name-value pair with a unique tag registered with the Object Management Group (OMG) and arbitrary associated data; these components define protocol-independent information, like a unique object id.

In standard CORBA, IORs are created internally at the server ORB. A server application creates a *servant* instance, registers the servant with the ORB (or, to be more specific, with an object adapter of the ORB). Usually, this is enough to provide a local object reference that enables to access the object (e.g., for plain CORBA the stub object). At the same time an IOR can be provided that contains an automatically created IIOP profile that includes hostname, port, object adapter name, and object id for accessing the object via IIOP. Additionally it

may contain other profiles representing alternative ways to access the object. An IOR can be passed to remote clients, either implicitly or explicitly. If a reference to a remote object is passed as a parameter or return value, the IOR data structure will be implicitly serialised and transferred. Upon deserialisation, the receiving client ORB automatically instantiates a local stub for accessing the remote object, initialised with the information available in the IOR. If multiple profiles exist in the IOR, the ORB may use a vendor-specific strategy to select a single profile that is understood by the ORB. The IIOP profile should be understood by all ORBs. Beside implicit reference transfer, an explicit transfer is possible. The server application may call an `object_to_string()` method at the ORB, which serialises the local object reference to an IOR and finally transforms it to a string. This string can be transferred to a client; the client application may call the local ORB method `string_to_object()` to create the stub for the remote object referenced by the IOR.

## 2.2 Status Quo of Extensible Middleware

Many practical tasks, e.g., authorisation, security, load balancing, fault tolerance, or special communication protocols require extensions to this basic model. For example, fault-tolerant replication requires that multiple communication addresses (i.e., of all replicas) are known to a client. Typically, this means that these addresses have to be encoded into the remote reference; binding to and invoking methods at such a remote object require a more complex handling at the client side compared to the simple stub-service design. As a second example, a peer-to-peer-like interaction between users of a service might sometimes be desired. In this case, the client/serve structure needs to be completely abandoned.

A good example for the lack of extensibility of standard CORBA is the fault-tolerant CORBA standard (FT-CORBA) [Obj04, Chapter 23]. It was not possible to define this standard in a way that all platform implementations are portable across different ORBs. Instead, each FT-CORBA-compliant middleware has usually its vendor-specific implementation inside of a single ORB (see Section 3.2). The Aspectix reference handling support offers the opportunity to implement FT-CORBA by a plugin that makes any ORB, independent of its vendor, aware of fault-tolerance mechanisms.

### 2.2.1 Standard Customisation Points of CORBA

Existing concepts for interception, custom object adapters, and smart proxies provide some mechanisms for such extensions. In part, they are included in the CORBA standard; in part, they are only available in non-standard implementations. The portable interceptor specification [Obj01, BMV03] supports interception within the official CORBA standard. The specification defines request interceptors and IOR interceptors as standardised way to extend the middleware functionality. Using request interceptors, several hooks may be inserted both at client and at server side to intercept remote method calls. These hooks allow to redirect the call (e.g., for load balancing), abort it with an exception (e.g., for access restriction), extract and modify context information embedded in the request, and perform monitoring tasks. A direct manipulation of the request is not permitted by the specification. Multiple request interceptors may be used simultaneously. Interceptors add additional overhead

on each remote method invocation; furthermore, they do not allow to modify the remote invocations completely. IOR interceptors, on the other hand, are called when a portable object adapter (POA) needs to create an IOR for a service. This allows inserting additional data into the IOR (e.g., a tagged component for context information that is later used in a request interceptor).

CORBA allows defining custom POA implementations. The POA is responsible to forward incoming invocation requests to a servant implementation. This extension point allows integrating server-side mechanisms like access control, persistence, and life-cycle management. It has, however, no influence on the interaction of clients with a remote service.

Beside the standardised IIOP protocol, any CORBA implementation may support custom invocation protocols. Additional IOR profiles may be used for this purpose. However, establishing such extensions is not standardised and every vendor may include his own proprietary variant, which limits the interoperability of this approach.

### 2.2.2 Adaptable Middleware Platforms

Smart proxies are a concept for extending ORB flexibility, which is not yet standardised by the OMG, but which is implemented by some ORB vendors, e.g., in the ACE ORB/-Tao [WPSO01]. They allow replacing the default CORBA stub by a custom proxy that may implement some extended functionality. As such, they enable implementing parts of the objects functionality at the client side. Closely related to our research is the work on OpenORB at Lancaster University [CBCP02]. This reflective middleware project uses reflection to define dynamic (re)configuration of componentised middleware services. One main difference to our design is that it completely restructures the middleware architecture, whereas our concept can be integrated in any existing CORBA implementation with only minimal modifications. It nevertheless provides equal flexibility in reconfigurations. PolyORB [VHPK04] is a generic middleware system that aims at providing a uniform solution to build distributed applications. It supports several personalities at the application-programming interface (API) level and the network-protocol level. The personalities are compliant to several existing standards. This way, it provides middleware-to-middleware interoperability. Implementations of personalities are specific to PolyORB, unlike the proposed approach, which is intended to be portable between different ORB implementations. However, PolyORB offers a genericity at the API-level that is currently not addressed by the Aspectix ORB.

## 2.3 Profile Manager

One of the core tasks of any object-oriented middleware is to provide mechanisms to create remote references, to pass them across host boundaries, and to use them for remote invocations. Unlike other approaches, the Aspectix profile manager completely separates the handling of references from the middleware core. The impact of such a design is not only that a single middleware implementation gets more flexible but also if applied to other implementations profile managers should be portable across different platforms.

In CORBA, an IOR is composed by a *type ID* and one or more *tagged profiles*. The type id usually states the most derived type of the interface of the object. Every profile consists

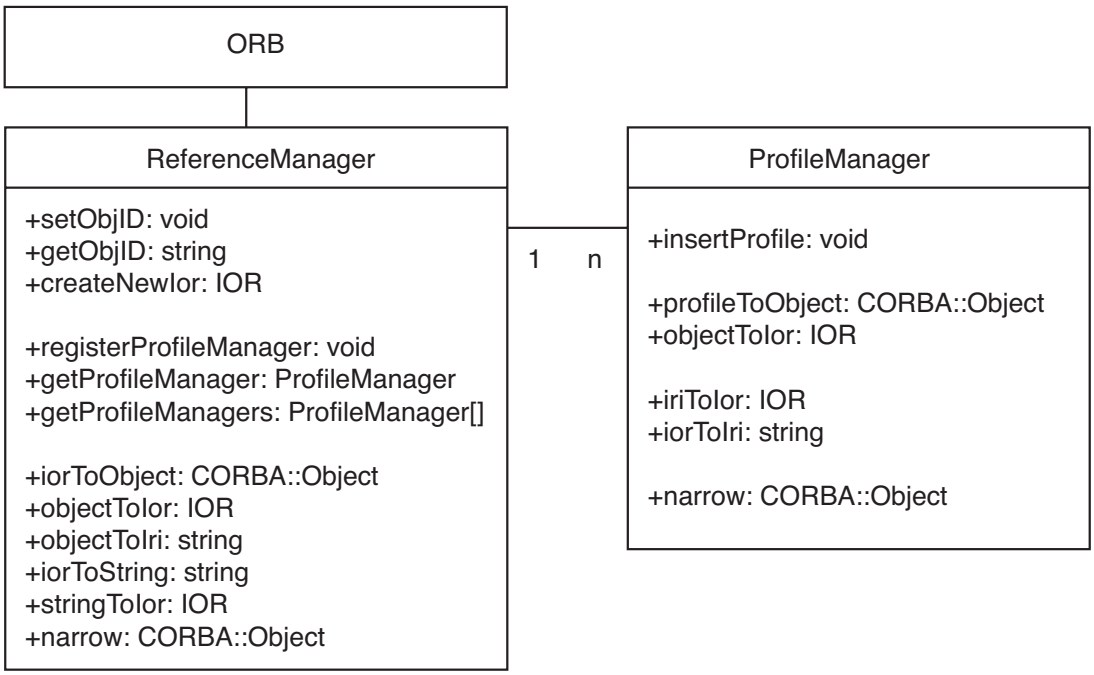



Fig. 2.1: UML class diagram profile manager

of a unique profile id that specifies the type of a profile and the profile data that is specific to the type of profile. For standard CORBA the IIOP protocol has to be supported. The dependent profile includes the protocol version, contact address and an object key. The last component is necessary to locate the object at server-side by the ORB.

Factoring out the basic remote-reference handling of the ORB core affects five core functions of the middleware:

- creation of new object references
- marshalling of references
- demarshalling of references
- explicit binding of references
- changing the type of a remote reference

All five operations have to be provided by a profile manager. As shown in the UML class diagram Figure 2.1 the reference manager enables the registration and management of profile managers. Profile managers subscribe for one or more IOR profile types. These types are registered with the OMG to ensure their uniqueness. Several tasks of reference handling are invoked at the reference manager and forwarded to the appropriate profile manager. In the following the five core functions of reference handling are described:

**Creating object references.** Usually the creation of an object reference is tied to the registration of a servant at a POA. In context of the Aspectix middleware the reference creation and object registration have been separated. In a first step the reference manager is asked for a new empty reference by calling `createNewIor()` resulting in the creation of

an empty IOR body. This empty IOR structure is then filled with one or more profiles by invoking the `insertProfile()` operation of appropriate profile managers.

**Marshalling of object references.** A CORBA object passed as a method parameter in a remote invocation needs to be serialised as an IOR. This is achieved by handing over the local object reference to the reference manager via `objectToIor()`. As there is no way to generally determine the responsible profile managers, the reference manager forwards the object to all of them. Every profile manager checks whether the object is activated for its middleware context and in case it provides the full IOR including all available profiles.

**Unmarshalling of object references.** If a standard ORB receives a serialised IOR as a method parameter or return result, it implicitly converts it into a local representation and passes this representation (typically a stub) to the application. Unmarshalling of references is handled by the `iorToObject()` operation of the reference manager. This time the responsible profile managers that possibly can unmarshall the reference are determined by the profile types included in the reference. Since a reference can include multiple profiles but only one profile can be used to bind the remote object, the reference manager invokes the profile manager in a pre-configured order. If none of the managers can deal with the IOR, an exception is thrown.

**Explicit binding to remote references.** A user application may explicitly call the ORB method `string_to_object()`, passing some kind of stringified representation of the references. Usually, this may either be a string representation of the marshalled IOR, or a *corbaloc* or *corbaname* URI. This ORB operation can be split into two steps: First, the string is parsed and converted into an IOR object. As the stringified IOR can have an extension-specific IRI format—an IRI is the internationalised version of an Uniform Resource Identifier (URI)—each profile manager is asked for conversion by using `iriToIor()`. The generic IOR format will be handled by the reference manager itself. Second, this IOR object is converted into a local representation using the same process as used with unmarshalling. Invocations of `object_to_string()` is handled in a similar way.

**Narrow operation.** Changing the type of a remote reference usually demands for an exchange of the local object reference (e.g., for plain CORBA the stub object), as an extended interface has to be provided, which is not supported by the current instance. This exchange can be handled by generated, type specific helper classes. In context of the Aspectix ORB these helper classes call the `narrow()`-operation of the generic reference manager passing the qualified type name and the reference. The manager calls every profile manager for the narrow operation. A profile manager can check whether the object reference belongs to its CORBA extension. If this is the case, the manager will take care of the narrow operation otherwise the reference manager will turn to the next profile manager. This approach requires an implementation language providing a reflection API as it is the case for Java. If reflection is not available, it might be necessary to generate helper classes including custom type conversion code for every supported middleware extension.

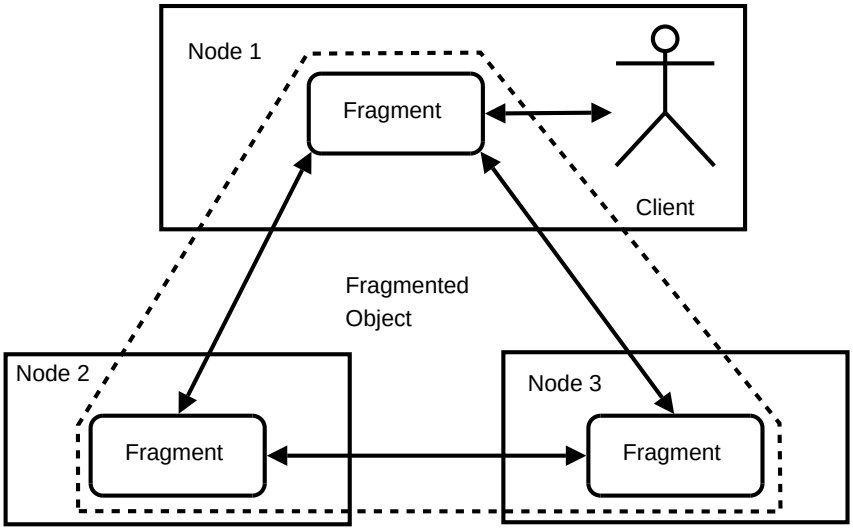



Fig. 2.2: Example for a fragmented object

## 2.4 Aspectix Profile Manager

The Aspectix middleware supports a fragmented object model. Unlike the traditional RPC-based client/server model, the fragmented object model does no longer distinguish between client stubs and the server object (see Figure 2.2). From an abstract point of view, a fragmented object is an entity with unique identity, interface, behaviour, and state, as in classic object-oriented design. The implementation, however, is not bound to a certain location, but may be distributed arbitrarily over various fragments. Any client that wants to access the fragmented object needs a local fragment, which provides an interface identical to that of a traditional stub. This local fragment may be specific for this object and this client. Two objects with the same interface may lead to completely different local fragments.

This internal structure gives a high degree of freedom on where state and functionality of the object is located and how the interaction between fragments is done. The internal distribution and interaction is not only transparent on the external interface, but may even change dynamically at run-time. Fragmented objects can easily simulate the traditional client/server structure by using the same fragment type at all client locations that works as a simple stub. Similarly, the fragmented object model allows a simple implementation of smart proxies by using the smart proxy as fragment type for all clients. Moreover, this object model allows arbitrary internal configurations to partition the object, or to replicate for fault-tolerance reasons (see Chapter 3). So fragmented objects go certainly beyond smart proxies as every fragment might serve a part of the object state and functionality like in a peer-to-peer scenario (see Chapter 4). Finally, the communication between fragments may be arbitrarily adjusted, e.g., to ensure quality-of-service properties or use available special-purpose communication mechanisms. All of these mechanisms are fully encapsulated in the fragmented objects and are not directly visible on the outer interface that all clients use.

To support the fragmented object model Aspectix provides an own profile manager. This is needed as the binding process of fragments differs significantly from simply loading a

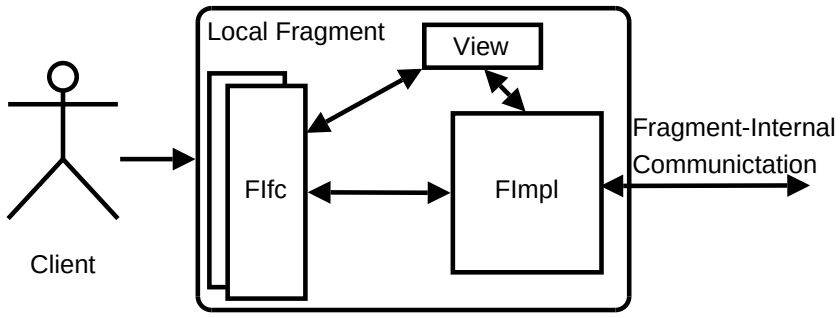



Fig. 2.3: Architecture of a local fragment

CORBA-based stub. A local fragment as it is supported by the Aspectix middleware is internally composed of three components: the *View*, the *Fragment Interface (FIfc)*, and the *Fragment Implementation (FImpl)*. Figure 2.3 shows this basic architecture of a fragmented object. The fragment implementation is the actual code that provides the fragment behaviour. The fragment interface is the interface that the client uses. Due to type casts, a client may have more than one interface instance for the same fragmented object. Interfaces are instantiated in CORBA using narrow operations; all existing interfaces delegate invocations to the same implementation. The View is responsible for all management tasks; it stores the object id and IOR, keeps track of all existing interfaces, and manages dynamic reconfigurations that exchange the local FImpl. The View updates all references from FIfcs to the FImpl and has to coordinate method invocations at the object that run concurrently to reconfigurations.

As the internal structure of a fragmented object is not limited in terms of communication endpoints and client-side functionality, a custom IOR profile is required. The IOR creation is highly application specific, thus it is not fully automatic as in traditional CORBA. Instead, the developer of the fragmented object explicitly defines, which information needs to be present in the object. The reference manager creates an empty IOR for a specified IDL type, and subsequently the APX profile manager can be used to add an APX profile to this IOR. This profile consists of the following three components:

- Code reference
- Peers
- Version

The *code reference* allows specifying the default FImpl that is instantiated during the binding process. It could either be a direct code reference like a Java class or a symbolic name that is resolved via a code loading service. Currently Aspectix provides the DLS (dynamic loading service [KH03]) that is capable to dynamically load object-specific local fragment implementations. In Chapter 6 an extended loading service is presented that targets the software distribution in a decentralised environment. The *peers* component identifies one or more communication endpoints that are handed over to the instantiated fragment implementation during the binding process. The version number is used to distinguish the most recent IOR as done by Fault-Tolerant CORBA (FT-CORBA) [BMPV02]. Each time the

peer set changes or a version of the default fragment implementation is introduced, the version number is increased.

When an ORB binds an IOR with an APX profile, the corresponding profile manager checks, if a fragment of the specific object already exists in which case a reference to the existing local fragment is returned. Otherwise, a new View object is created and connected to a newly instantiated FImpl. The View is provided by the Aspectix middleware and offers essential support for handling of FIfcs and a safe exchange of fragment implementations.

Profile information specifies how this FImpl is loaded (direct Java class name for Java-only environments, a code factory reference, or a unique ID for lookup to the dynamic loading service (DLS) [KH03]. Finally, a default FIfc of type `APXObject` is created and returned to the client application that can now access the object. The APXObject builds the base type of all FIfcs and provides additional methods for handling the fragment-associated View object.

## 2.5 Summary

The Aspectix ORB provides a flexible and lightweight approach to integrate other non-CORBA middleware platforms based on a generic reference manager and custom profile managers. On top of these mechanisms, Aspectix offers support for the fragmented object model. This support constitutes the foundation to provide replicated and peer-to-peer based decentralised adaptive services as outlined in Chapter 3 and Chapter 4. Apart its flexibility the fragmented object model, as provided by the Aspectix middleware, is transparent to the client. Existing applications can be used without modification and benefit from fault-tolerant or peer-to-peer based solutions.

# 3 Replicated Decentralised Adaptive Services

This chapter focuses on the development and the efficient provision of replicated decentralised adaptive services. Theses services are able to host information that is concurrently accessed and modified by multiple clients. According to the objectives of this thesis, replicated decentralised adaptive services should tolerate a limited number of crash-stop failures and adapt their distribution according to user and environment demands. In addition to these main objectives, this thesis aims at supporting a flexible development approach that is similar to those of common object-oriented client/server applications. However, extended framework support beyond the provision of a plain RPC-style stub/skeleton code generation is desired, to address the implementation of services with non-standard requirements (e.g., custom internal communication) .

Replicated decentralised adaptive services are provided by a framework that, at its core, supports active replication of object-oriented services. In contrary to related approaches, this framework introduces flexible mechanisms for client-side computing. Additionally, it offers better performance by introducing developer-controlled custom code generation and support for deterministic multithreading. Finally, special attention has been put on mechanisms for service mobility by providing extended support for service migration and an extended object reference concept.

After summarising the objectives of this chapter and giving a brief summary of the main contributions of replicated decentralised adaptive services, an overview about this chapter is given in the following:

To provide the necessary background for this chapter the following section discusses the two principle forms of object replication named *active* and *passive*. The section also discusses why only active replication is suitable for replicated decentralised adaptive services. Finally, the state of the art in fault-tolerant services based on object replication infrastructures is described. Starting from this basis, Section 3.3 presents the core replication architecture of the framework for replicated decentralised adaptive services. At this point basic support for fault-tolerant object replication is available based on the support of fragmented object model. In the following, the extended features that differentiate the proposed framework for replicated decentralised adaptive services from previous systems (see Section 3.3) are separately described and each contribution is compared to related approaches.

As one of the major objectives of this thesis is to provide adaptive services, extended support for service mobility has to be provided, and the impact of dynamic distribution has to be addressed. One of the major consequences of service mobility is the problem of outdated or even stale object references. Common concepts to address this problem and reference handling of replicated objects in the context of FT-CORBA are discussed in detail in Section 3.4.1. Next, Section 3.4 presents an approach to reliably address and bind replicated decentralised adaptive services using the proposed time-bound object references.

However, not only the service binding of fault-tolerant services has to be treated differently in dynamic environments. The necessary state transfer to set up a new replica of a stateful service is an even more important issue. This is especially the case as distribution changes occur more frequently than in static environments. Therefore, Section 3.5 investigates on how to speed up and optimise the state transfer. The section starts with an overview about related approaches in non-replicated and replicated systems. Afterwards, it is discussed how to acquire the state from a replica and how to provide support for large application states. Next, it is explored which transfer format is suitable. As one of the major contribution in this section is the support of a parallel state transfer that is so far not offered by any other system, a detailed description about parallel download approaches for file transfer is given. Using this information as background, two adapted algorithms are presented and additional techniques to speed up state transfer are discussed. Finally, an implementation of the presented parallel download algorithms is presented and evaluated by experiments in the context of a local area network (LAN), but also in an internet setting.

One of the major weaknesses of active replication is the communication overhead introduced by using a group communication framework to enforce object determinism. This issue becomes even more severe if only one thread is allowed to execute inside of an object to preserve object determinism as required by many existing systems. The consequence is a very restricted programming model and serious performance degradation. To offer an implementation model similar to a traditional client/server, model extended support for multithreading inside of replicated decentralised services is provided. This includes a deterministic scheduler that supports the basic set of the native Java synchronisation primitives.

To enable developing services with non-standard requirements, as originally addressed by integrating fragmented objects into CORBA, the framework for replicated decentralised adaptive services provides custom development support. This is achieved by introducing semantic annotations at the remote interface level of a service. Theses annotations are evaluated during the code generation process that is necessary for creating the basic service skeleton and request-marshalling code. Section 3.7 addresses the support for semantic annotations, which offer support to optimise the request handling and to introduce service-specific code at client-side.

To start and execute a replicated decentralised adaptive service, a minimal run-time infrastructure is needed that is described in Section 3.8. This basic infrastructure was integrated in the EDAS environment that is subject to Chapter 5.

To evaluate the benefits of the proposed framework, namely the support for multithreading and custom code generation two evaluations were performed. Section 3.9 details the results of a performance evaluation of the multithreading support and the performance-relevant semantic annotations. Finally, Section 3.10 presents a use case, a prototype implementation of a replicated decentralised version control system. This service has extended requirements that demand for custom internal communication and extended client-side computing support. Thus, an implementation with a traditional object replication framework would be cumbersome, if not impossible.

## 3.1 Object Replication Style

A common approach to offer continuously executing and highly available services is to replicate them over multiple nodes. There are two basic kinds of replication styles, known as: *active replication* and *passive replication* also called primary-backup approach [Mul93].

Active replication requires that every client request be processed by each replica. All replicas return their response to the client. Duplicated responses are suppressed; only one is passed to the client. If one or more replicas fail this is transparently masked out by the remaining replicas and the replication infrastructure.

Using passive replication, there is only one designated replica called the *primary* that processes all client requests and returns the associated responses. There are two forms of passive replication *warm passive* and *cold passive* [Obj04, Chap. 23]. Cold passive replication requires only the primary to be operational and additionally a facility that logs all requests to a stable storage. From time to time, a snapshot of the service state is taken by the primary and distributed to all other replicating nodes. If the primary fails, one of the other replicas activates a snapshot, replays the requests not covered by the snapshot from the stable storage, and starts acting as the new primary. Warm passive replication requires sending the results of every request to all replicas. In contrast to cold passive replication, the replicas are up and operational all the time. Thus, warm passive replication has a shorter failover-time, as only one of the passive replicas has to be selected as the new primary. The drawbacks of this approach are the extended resource demand, as all replicas have to be active in comparison to cold passive where only the master node is active. Another issue of warm passive replication is the selection of a replica that covers the latest requests.

Passively replicated services usually have a better response time as requests do not have to be distributed to all replicas via an atomic broadcast. On the downside, if the primary replica crashes there will be a longer failover procedure reducing the availability of a service, whereas replica faults can be immediately masked out in case of active replication.

The use of passive replication is indicated if node failures are rare and high performance matters. However, replicated decentralised adaptive services are dynamically distributable and node failures might occur more frequently than in static environments. Hence, only active replication is appropriate and has been chosen for this thesis.

## 3.2 Approaches to Replication Support in Distributed Object Middleware

Providing fault-tolerant services on top of CORBA has always been a topic of major research since the early beginnings of CORBA. So far, three general implementation approaches have been proposed and implemented, referred to as the *integration approach*, the *interception approach*, and the *service approach*. To make fault-tolerant services available in CORBA in a standardised way the Object Management Group (OMG) provides since 2001 the FT-CORBA specification [Obj04, Chap. 23]. It defines the basic conditions to realise fault tolerance at the middleware level in CORBA but does not provide a specific implementation. This section

gives a brief overview on how to integrate fault tolerance in CORBA and other standard middleware infrastructures. A more detailed and general description about fault-tolerant services on top of CORBA and the FT-CORBA specification can be found in [FN04].

The integration approach modifies the ORB internals to provide the necessary fault-tolerance support. Usually this includes the modification or even replacement of the communication facilities (e.g. by replacing the TCP/IP-based IIOP transport by a proprietary group communication protocol). As the modifications to provide the necessary fault-tolerance mechanisms can usually be restricted to the ORB core and do not require modifications of the ORB interface, applications can be left unchanged. The integration approach represents the most efficient solution but has the drawback of a substantially more complex ORB that might be noncompliant with the CORBA standard. Orbix+Isis [BR93] and Electra [Maf95] are prominent examples that successfully applied the integration approach.

The service approach introduces fault-tolerant mechanisms by adding a new service as a CORBA Common Object Service. As it entirely relies on CORBA objects the ORB does not need to be modified; therefore the approach is CORBA-compliant. However, implementations following this approach such as the Object Group Service (OGS) [Fel98] or the DOORS system [GNSY00] are not transparent to clients. A client has to pass requests to the service that distributes the requests to all replicas and collects the dependent responses. Every request has to be passed from the client to the service through the ORB, which incurs a performance overhead.

The interception approach introduces fault tolerance transparently for the application and the ORB via an interceptor component. Such a component can be attached to an application to provide additional functionality and change the behaviour of an application without its modification. Thus, interceptor-based systems like Eternal [MMSN98] can transparently intercept and redistribute client requests to replicas. CORBA provides a specified interface [Obj01] to dynamically introduce interceptors as well on client as on server-side. Hence, the interceptor approach can be used to introduce fault-tolerance mechanisms for any off-the-shelf ORB.

There are some other CORBA-based systems supporting fault-tolerant services that cannot be exactly classified, e.g., FTS [FH02] and IRL [MMVB00]. FTS for example uses the interceptor approach at client-side but provides an own object adaptor to handle replication at server side.

Besides CORBA there are other standard middleware infrastructures that have been enhanced to provide replicated services such as Java RMI and the .NET remoting infrastructure. Both systems can be extended in a similar way as they provide the facility to transparently load and execute customised stubs. This is a key feature to transparently introduce new remote-invocation semantics.

The Java RMI middleware architecture is composed of three layers. The first layer is the stub and skeleton layer; it enables the distribution-transparent invocation of remote objects and is responsible for marshalling and unmarshalling of requests. The remote reference layer is second; it determines the call semantics of a Java RMI Object. Finally, there is the transport layer, which uses the Java socket API to handle communication between clients and servers. New call semantics can be introduced by extending the reference layer.

The Jgroup project [Mon99] used this to integrate the group communication paradigm for the support of one-to-many semantics into Java RMI. The group communication was integrated by providing an own implementation of `RemoteRef`, the central extension point of the reference layer. A similar approach has been taken in the AROMA system [NMMS00] and by Cazzola et al. [CAC+02]. Besides offering a special `RemoteRef`-implementation, Cazzola et al. additionally changed the stub layer. When invoking an operation on a replicated object the client receives a result array containing the invocation results of every replica. Thus, the client-side transparency has been given up to provide the opportunity for parallel execution. This goes beyond the provision of fault tolerance.

The .NET remoting infrastructure provides the possibility to dynamically load custom stub code ("real proxy") instead of the default stub. The support for custom stubs constitutes the basis to transparently add custom support for replication [RDH05].

## 3.3 Fault-Tolerant Replication with Fragmented Objects

As an alternative to the three previously presented approaches the integration of the fragmented object model into the CORBA middleware and the proposed framework for replicated decentralised adaptive services constitutes a fourth solution. As outlined in Chapter 2 the fragmented object model offers great flexibility and can be integrated in most common object-oriented middleware infrastructures. Additionally custom code generation and framework support for replicated services can be easily layered on top. This approach requires fewer modifications to the middleware as only the basic support for fragmented objects has to be integrated. It might even be possible to use the same replication framework with different middleware implementations that support the fragmented object model. Thus, the contribution of the presented framework for replicated decentralised adaptive services is a reduced integration overhead of fault-tolerance support and better portability.

In case of plain fragmented objects, as the Aspectix ORB currently supports them, a developer has not only to implement the object functionality but also the whole internal communication of the fragmented object (see Chapter 2). This is not necessary for replicated decentralised adaptive services. A developer simply specifies the service interface using the interface definition language (IDL) as required for any other CORBA or fragmented object and passes the interface description to the Aspectix IDL-compiler IDLflex [RSH01]. This compiler can be configured via Extensible Markup Language (XML)-based mapping files to generate custom code. In the case of replicated decentralised adaptive services an own mapping was implemented that automates the generation of basically two kinds of fragments (Figure 3.1): *Replica fragments*, which encapsulate implementation and state of a service and *access fragments*, which are mainly used to connect clients to the service. Unlike common CORBA and CORBA-FT, implementations the code generation at client- but also at server-side can be customised. This is achieved by the introduction of semantic annotations at the interface level (see Section 3.7). Semantic annotations enable optimisation of the request processing, but also offer the opportunity to introduce custom code into access fragments. Thus, access fragments represent not only some kind of group stub

for replicated execution handling but may take the role of a smart proxy [KK00, SMS02]. In some cases access fragments can even provide further functionality and act as service providing entities by exchanging information with other access fragment instances.

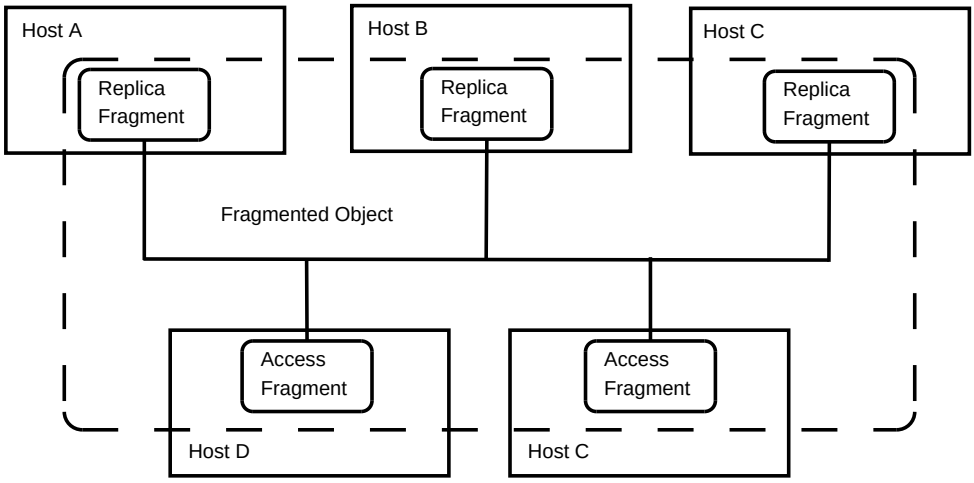



Fig. 3.1: Replication with Fragmented Objects

### 3.3.1 Details of the Fragment Architecture

Based on the general architecture presented in the previous section, this section targets the description of the internal structure of access and replica fragments. Figure 3.2 depicts the layered design of the replicated decentralised adaptive service model. Three different kinds of code can be identified: Developer-supplied code, representing the actual object implementation residing on server but also on client-side. IDL-generated framework code responsible for the provision of a CORBA-compliant object interface and location-independent access. Finally, there is static code representing the core framework responsible for the object independent request handling and transmission.

Starting at client-side, a request is passed to a service via its interface. The request is intercepted or completely executed, if there is a developer supplied custom method implementation. This is useful if for example a request can be immediately answered using cached results of previous invocations or if a requested method invocation needs no service state (e.g., a value conversion). If the request is not locally answered, it is passed to the *context handler* layer. The context handler layer is responsible for adding call context information that enables the unique identification of requests. This is necessary as an actively replicated service $A$ might call another replicated service $B$. Such a nested invocation is only processed by one selected replica of $A$ that actually executes the invocation. If this replica crashes, the invocation of the other service has to be reissued. In this situation, the unique identification of requests allows $B$ to suppress duplicated executions at the replica side.

Following the context handler layer, a request is marshalled and passed to the network layer. Active replication requires the usage of a totally ordered group communication system to ensure consistency. In the context of this thesis the JGroups system [Ban98a] was

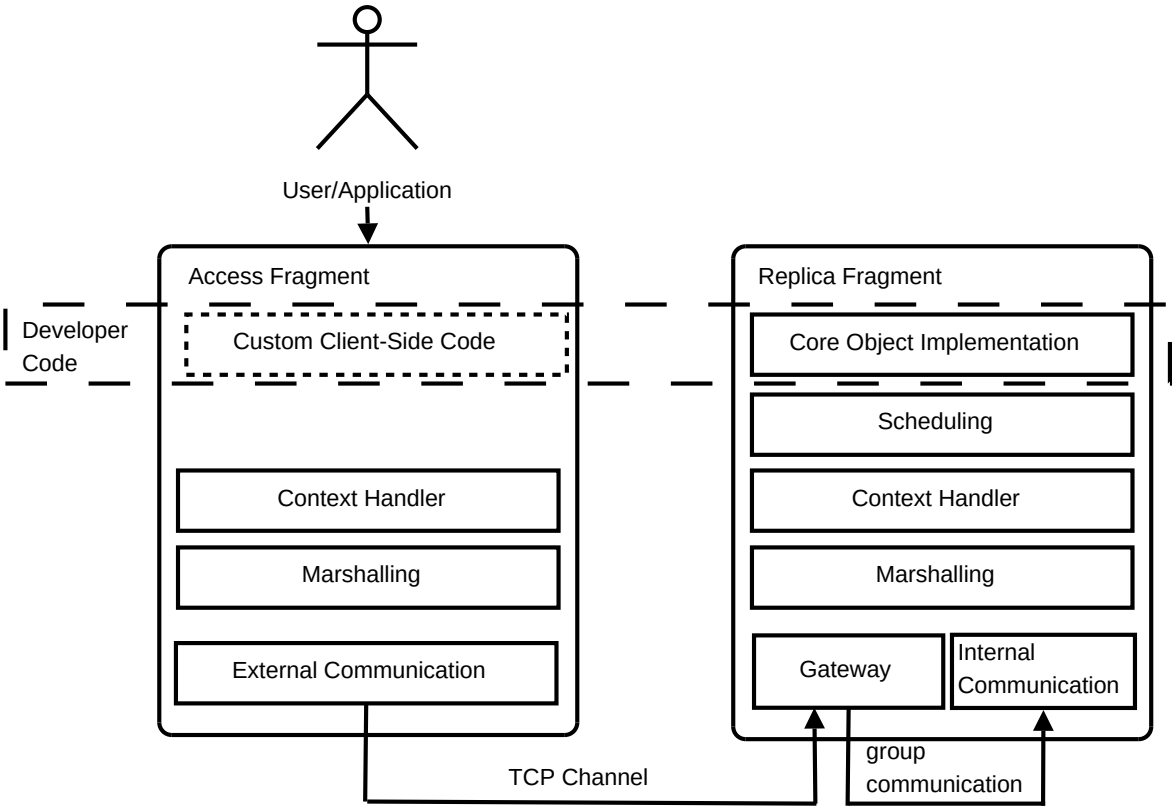



Fig. 3.2: Architectural Overview of the Framework for Replicated Decentralised Adaptive Services

used. It follows the *closed-group* approach, which requires that all communicating entities be members of a group. For this reason access fragments cannot send messages directly to the replica group but forward their message to one of the gateway components provided by every replica fragments. Thus, an access fragment only needs to know the gateway access information.

On the contrary, the usage of group-communication frameworks following the *open-group model* may improve performance (e.g., the access fragment can directly multicast its request to all replicas) and is beneficial for the support of Byzantine fault tolerance [LSP82], as access fragments may receive replies from multiple replicas and use voting. For this reason Reiser recently integrated his open-group group-communication framework [Rei06] into the prototype implementation.

The *external communication* component is responsible for sending requests from the client-side to one of the *gateway* components. If a replica fragment fails, the external communications component connects another gateway component and reissues the call. If the call was already successfully executed, a cached result is returned by the gateway. Otherwise, it forwards incoming requests to all replicas via the group communication framework. Requests on the group communication layer are consumed by the *internal communication* layer and passed upward into a totally ordered queue for subsequent processing by the upper layer.

In the upper layer, the request is first processed by the *marshalling* component, which is

responsible for deserialisation of requests and serialisation of replies. Following that, the context handler component suppresses duplicated requests if there are any, and passes the request upward to a *scheduling* component. This component is responsible for the internal message management and provides a deterministic scheduler that preserves ordering in context of multiple threads modifying the state of the developer-provided implementation. See Section 3.6 for an extended description on the scheduling component.

### 3.3.2 Summary

The outlined architecture builds the core of replicated decentralised adaptive services. It offers the fault-tolerant replication of object-oriented services. It is more lightweight and more portable than related approaches as it only relies on the support of the fragmented object model. Thus, the goal of providing services that can be distributed over a set of nodes and that are able to tolerate a limited number of node faults has already been reached. Still missing at this point to provide long-running services are mechanisms to support dynamic distribution for accounting environment and service demand changes. This is addressed in the following sections.

## 3.4 Time-Bound Object References

As mentioned before, this thesis aims at providing services that can adapt their distribution according to service and external requirements. A serious consequence of service mobility is the problem of outdated or even stale object references. Usually, whether a service is replicated or not, an object reference includes some form of physical communication endpoint (e.g., an IOR includes an internet address and port number). This communication endpoint information is invalidated by service migration. If a service is replicated, a reference might include more than one communication endpoint, and therefore only outdates at the first place, but after a limited number of distribution changes even references of replicated services become stale at some point. Usually, this issue is addressed using a location service that has to be always accessible or by employing a request forwarding mechanism. This thesis argues that both approaches are sufficient if a static execution environment is available but fail if the set of nodes hosting the execution infrastructure of services is as well as the hosted services dynamic. This applies for replicated decentralised adaptive services as they can completely change their distribution along with set of nodes hosting the services.

To avoid stale references despite dynamically changing environments the concept of an extended reference that gives time-bound availability guarantees named *time-bound object reference* was developed. This concept exploits the fact that replicated decentralised adaptive services are spaced over multiple locations and usually change their distribution by stages. Time-bound object references provide information how fast the distribution of a service may change, thereby offering clients the option to update to the most recent version before the currently held reference becomes stale. Thus, in contrast to previous approaches, the handling of reference changes was integrated into the replicated services itself, requir-

ing less infrastructure services and, even more important, being independent of a static set of supporting nodes.

The basic concept of time-bound object references preserves only the connection to a bound object and specifies the validity of a reference. Another important aspect is reliably binding an object despite distribution changes. There are two commonly used approaches to handle reference updates: request forwarding or using a location-service. Both approaches require to either trust in former object locations or in the node hosting the location service. This is reasonable in traditional middleware environments because the set of nodes hosting the objects and the location service are static and well known. This means a node that belongs to a domain is never expected to be excluded. In context of EDAS and replicated decentralised adaptive services, this could happen for security and trust reasons. Even more serious it is expected that former supporters of a service that have been excluded from the set of resource providing nodes might turn into attackers and try to inhibit the binding process. Such a scenario has so far not been considered in the context of standard middleware infrastructures. Mechanisms like public-key–based authentication offer only a partial solution to the problem as they only allow clearly identifying a host. To address the problem that nodes formerly hosting a service try to attack the object binding, time-bound object references were extended and a special binding process is proposed.

Despite that the basic concept of time-bound object references does not rely on a location service, it can still benefit from such an infrastructure, for example, if a reference has become stale and the associated service needs to be bound. Consequently, it is outlined how to combine time-bound object references with a location service, while still preserving a secure binding process.

Giving the necessary background for the proposed concepts the following subsection outlines the standard approaches to enable object binding despite mobility and the handling of fault-tolerant service references in the context of FT-COBRBA. Basing on the Aspectix profile for fragmented objects, the mechanisms to provide time-bound object references, which assure a safe and reliable object binding, are then explained in detail. This includes mechanisms to handle reference updates and a secure object binding process. Finally, the support of a location service is introduced. This section closes with a use case scenario and a brief description of related approaches.

### 3.4.1 State of the Art

This subsection outlines the two basic approaches to handle reference changes in the context of object mobility and the support for reference handling for fault-tolerant services as specified by the FT-CORBA standard.

#### Standard approaches to enable object binding despite object mobility

Most common object-orieted middleware platforms, such as .NET, Java RMI or CORBA, lack sufficient support to provide reliable references for accessing distributed mobile objects. If a non-replicated object is migrated to another node, its reference becomes stale as the reference includes a physical address that is no longer valid. However, this problem can be

solved by introducing additional mechanisms and using the extension points provided by the middleware platform.

A possible solution to the problem is offered by the introduction of a request forwarding mechanism that basically resends requests from the old object location to the new one. Such an approach is supported by CORBA and can also be provided by other platforms. If an object is migrated to another location using for example the CORBA Life Cycle Service [Obj02] the object leaves its new address information at the previous location. This information can be kept by the POA that was responsible for managing the object. If an application issues a request using the outdated reference the POA at the old location receives the request. Instead of throwing an exception that the object does not exist, a forwarding exception is returned. This forwarding exception includes the new reference. Received by the ORB instance at the client-side, the exception is not propagated to the client, but handled at the system level. The ORB resolves the new reference from the exception and resends the original request to the new location. This process could happen even multiple times, if the object changes its location more than once. The major weakness of this approach is the forwarding chain is broken if one of the previous locations of the object is shutdown or crashes. Consequently, if a client binds an outdated object reference and issues a request that at some point will be forwarded to the unavailable node, an exception will be thrown that indicates that the object does not exist — while it is still accessible on a different node.

Another option is the employment of a location service enabling the registration and look up of objects. Initially, the object has to be registered at the location service. Each time an object is migrated to another location its reference has to be updated at the location service. Clients that bind an old object reference version might get a connection error when accessing the object and have to look it up at the location service to rebind the reference, which includes the current location of the object.

In context of CORBA this approach can also be transparently provided to the client as described by Henning [Hen98]. In this case a reference is created that points to the location service instead to the real object location but includes information to uniquely identify the object. If the reference is bound and requests are issued, they are forwarded to the location service, which in turn throws a forwarding exception. Again, at the client-side ORB the exception is evaluated and the request is transparently sent to the real location of the object. The ORB caches the new location information and sends all following requests to it. If the object moves again to a new place the client-side ORB receives an exception and tries the original version of the reference, that still points to the location service. If the object has already registered its new location at the location service, the service returns this one. This approach works well even if nodes that previously hosted the object crash or get shutdown. However, the location service always has to be available and represents a central point of failure.

Request forwarding but also the use of a location service are per se not fault tolerant as a forwarding node or the location service can fail. Thus, forwarding is not a real option in a large distributed system as it required all previous locations of a object to stay available which usually cannot be guaranteed. If a location service is used, only the node hosting

the service and the service itself has to be available. This can be ensured using additional mechanism like replication.

Still, both mechanisms rely on the fact that — despite objects are mobile — some supporting nodes have to keep their address and stay accessible. In context of a decentralised adaptive service and EDAS this is not the case, so both approaches are not sufficient.

### Object References in FT-CORBA

As briefly detailed in Section 3.2 the FT-CORBA specification aims at simplifying and standardising the development of fault-tolerant, object-oriented distributed services. This section investigates on how to do the reference handling of replicated objects according to this specification. An extended description can be found in [BMPV02].

According to the FT-CORBA specification, each fault-tolerant object is replicated by a *replication manager* that manages replication groups and creates/destroys objects of a fault-tolerant domain. A replicated object is realised as a group of individual CORBA objects each having the same interface and a unique reference. By aggregating the IORs of each replica of the group an *Interoperable Object Group Reference (IOGR)* is formed. The replication of an object is transparent to the client application. A client uses an IOGR in the same manner as it would use an IOR. If an object group is extended or shrunk, the IOGR will be updated. To distinguish object-group changes the IOGR contains a version number, which is increased by every change. To make clients and servers aware of group changes the version information is passed with every request. If a lower version number is detected at the server-side, an updated version will be forwarded to the client. If a higher version number is detected, the new IOGR is accepted or the replication manager will be asked for advice.

This approach to reference replicated objects has certain weaknesses in the context of dynamically changing environments. If all hosting peers of a replicated object group change over time, previously generated IOGRs referencing these peers become invalid and the object can no longer be accessed. Only if at least one of the originally hosting peers is still operational and belongs to the fault-tolerant domain, the associated replication manager can provide the current object reference. This way the replication manager has to be known to all nodes of the domain and represents a central well-known service that has to be available all the time. To sum up, the FT-CORBA specification assumes a more or less static domain and requires a replicated replication manager to omit a central point of failure.

Again, as pointed out in the previous section for dynamic environments like EDAS (see Section 5), but as well for peer-to-peer systems, and in the area of mobile computing, these assumptions cannot be kept. Such environments require that not only the members hosting an object group can completely change over time, but also the group of supporting nodes. In case of FT-CORBA, this would mean to change all nodes of a fault-tolerant domain. Even worse, IOGRs will not only outdate, or get stale, but clients that have already successfully bound a replicated object could loose their connection to the object if they do not frequently call methods of the object and thereby update their local reference. Furthermore, there is no information available when an IOGR invalidates and therefore the contact between a client and a server may silently break away. The consequences are outdated and

invalid IOGRs, lost connections, and in the worst-case scenario, the binding of wrong or even malicious objects.

### 3.4.2 Basic Concept

Referring to Chapter 2, the Aspectix middleware provides a flexible profile manager that is capable to handle profiles for IIOP, Jini and fragmented objects. The basic Aspectix profile for fragmented objects consists of three components: A *code reference* describing the local fragment implementation, a *peers component* providing information about communication endpoints and a *version component*. So far, the Aspectix profile can be compared to the FT-CORBA profile except for the code reference that is necessary to load and instantiate the initial fragment implementation. In addition to the basic Aspectix profile, time-bound object references possess a *validity period* that defines how long at least one of the communication endpoints referred by the peer component is accessible. Therefore, it simply defines the lifetime of a reference by providing a time-stamp.

The introduction of a reference lifetime has various consequences. First of all it has to be assured that no reference version is invalidated before the corresponding validity period expires. This means the peer set must not change completely before the validity period of a reference (e.g., an IOR in case of Aspectix) times out. Hence, a new version may have the same validity period as previous versions but should never have an earlier one. Furthermore, if the most recent reference version is at risk to expire a new version with a longer lifetime has to be provided. Clients, on the other hand, have the duty to ask for a new IOR version before the validity period of their current reference expires. This is usually only necessary in case of long periods without normal call activity, as new versions are usually piggy-backed with call results, or after an off-line period. In contrary to FT-CORBA, a decentralised adaptive service thereby can safely change its distribution in the context of dynamic environment without accidentally invalidating a bounded reference.

Figure 3.3 depicts the introduction of a validity period in detail. The left side illustrates the current approach of FT-CORBA, which does not provide any explicit information about a reference lifetime. The right side displays the validity period of each reference that is included in every time-bound reference. The black dots denote the valid communication endpoints, whereas the grey ones mark stale communication endpoints. An existing reference is represented by an oval around a set of communication endpoints. In standard FT-CORBA, a client can hold a reference and, as long as the client does not execute a request, the reference may silently outdate. Finally, a reference gets staled as indicated by the dotted oval. In case of the time-bound reference concept, each reference includes a validity period indicated by the arrow and the depended vertical line. The middleware and the object implementation now have the duty to update a reference before the most recent reference version expires.

### 3.4.3 Update of Time-Bound References

On the technical side, time-bound object references require every party to have a weakly synchronised clock. It is expected that a peer is either synchronised via the Network Time

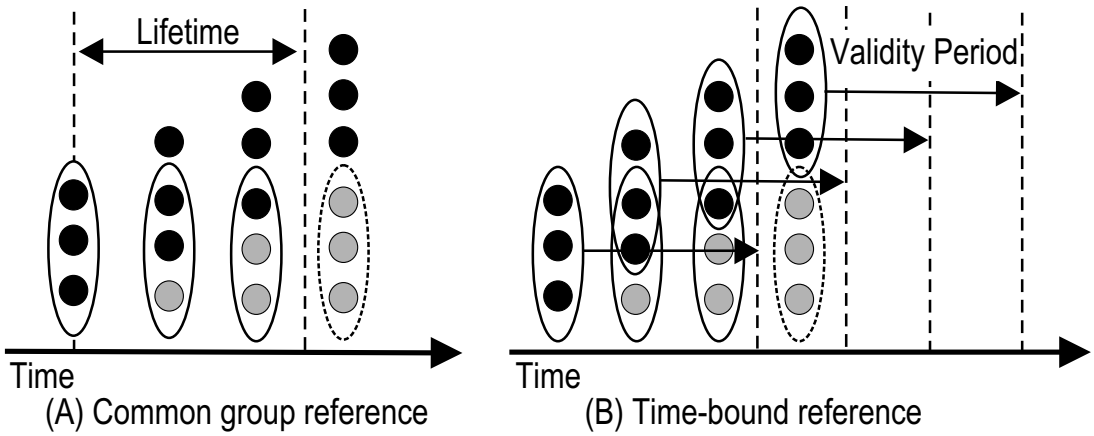



Fig. 3.3: Basic concept of time-bound object references

Protocol (NTP) [Mil92] or a similarly accurate time source. In fact, the need for accuracy of the time source strongly depends on the length of a validity period. Values ranging from several minutes up to several days are reasonable. The length of the validity period mainly depends on the redistribution rate of a decentralised adaptive service, but also on the number of peers. Therefore, the lifetime of an IOR is more or less service-specific and for this reason should be determined by the service. Two general update-policies are proposed.

The first one is called *on-update-or-expiration* policy. Each time the distribution changes (a part of the service migrates to another location; a new location is added; a peer crashes or simply drops support) a new IOR version is generated. This IOR is passed whenever it is needed either explicitly or implicitly (e.g., marshalling). Additionally, if the most recent IOR version runs the risk of expiration, a new version with a new validity period has to be generated. This should typically happen before at least one of the parties holding the IOR expects the invalidation. As a rule of thumb, one would suggest half of the lifetime of an IOR.

The second update policy is named *continuous* policy. Every time the IOR is requested explicitly or implicitly a new reference version with a new validity period is provided. Thus, this policy includes the on-update-or-expiration policy, as clients always receive a new reference that has the maximum lifetime. Assuming equal reference lifetimes for both policies, this period usually will be much longer than if an on-update-or-expiration policy is used. On the other hand, the continuous policy has the drawback that the object has a higher overhead to provide and maintain the corresponding information; furthermore it restricts the redistribution because every time a reference is accessed a new validity guarantee is given. Thus, it is more or less anticipated that the distribution changes continuously.

Neither of the two policies fits for all use cases. A developer has to decide, depending on the service and the environment, which policy to use and it is expected that sometimes intermediate actualisation schemes will be chosen. As earlier noted, the same applies for the lifetime of references. However, usually one would like to combine the on-update-or-expiration policy with a long reference lifetime. This combination has a very low overhead and the seldom generation of new references does not matter for the clients because of the

overall long lifetime of references. However, this combination is only suitable for services with a low rate of distribution changes. In contrast the combination of the continuous policy and short reference lifetimes is suitable for dynamic environments with high change rates. This gives clients every time they receive a reference, a maximum availability guarantee and imposes, despite the continuous policy, low restrictions on distribution changes because of the short reference lifetime. The drawback is a higher overhead to provide the references and the distribution changes that can be performed at the same time are restricted by all valid references.

### 3.4.4 Secure Binding of Distributed Objects

Time-bound references assure a reliable service binding as long as the validity period of a reference has not yet timed out. However, in the scope of dynamic environments, such as EDAS, where services are distributed over a changing set of peers that may belong to various independent parties, techniques for a secure service binding are necessary to prevent attacks at binding time. More precisely, two threat scenarios in dynamic environments are described and how to enhance the basic concept of time-bound object references to cope with them.

#### Secure Binding Despite Overtaken Communication Addresses

The first threat scenario is given by a form of *man-in-the-middle attack* [Sch95]. An attacker takes an address of a peer that supported or still supports the service. The attacker acts as the peer and creates a service reference that only includes peers that belong to the attacker. If a client binds to such a reference, the attacker can intercept and manipulate all requests for a service. This might be possible if a supporting peer has a dynamically assigned internet address, disconnects from the network and an internet service provider reassigns the free address to the attacker.

This form of attack and other ones, such as internet address spoofing are usually prevented by using public key cryptography [DH76] to clearly identify nodes and services. Consequently, every node generates a public-key pair and includes the public key together with its address into the IOR. The associated private key is kept secret by every peer. This way, each peer can be identified using the public key included in the reference by a secure handshake. Additionally, this offers the use of encryption at transport level.

However, in a first step, the IOR has to be passed via a trusted party to the client because the reference passing party provides the initial contact to the service, thus it has always the chance to replace the whole reference. This is known as the *first contact problem* in the context of peer-to-peer systems like Freenet [CSWH00], which aims at providing an anonymous information system. A signed public key and a well-known certificate authority solve this issue, but introduce additional complexity and overhead, as everybody has to know and trust the certificate authority. Therefore it is assumed in the context of this thesis that the client acquires the initial reference via a trusted party.

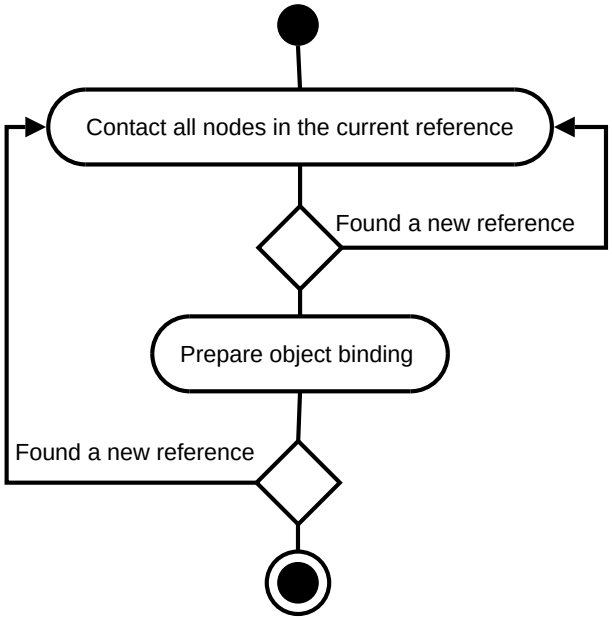



Fig. 3.4: UML state diagram of a secure binding process

#### Secure Binding Despite Attacks by Former Members

The source of the second treat scenario is the decision of a service manager to no longer take the resource of a certain node or a group of nodes. The reasons for such a decision might be security problems of the resource provider (e.g., missing critical security patches) or social problems. Thus, such nodes pose a security risk to the service.

A simple solution to the problem might be to redistribute the service as fast as possible according to the guarantees given by the valid references, and assume that the peers behave well until they are not longer needed. Depending on the lifetime of references this can be a very long time. Thus, a faster process to address exclusion of nodes is demanded. The ultimate solution would be to be able to exclude all nodes hosting the service immediately and do a complete redistribution in one step. However, this is not possible, as it would invalidate the whole concept of time-bound references; a valid reference might no longer be usable to connect a service because a service manger excluded all nodes of the reference. Even worse, the excluded nodes are now able to attack the service binding by pretending still being valid supporters of the service.

Consequently this thesis proposes an intermediate solution to enable a fast exclusion of possibly dangerous nodes and to preserve the guarantees given by valid time-bound references. The basic idea is to assume that a reference does not only give the guarantee that at least one node is accessible until the validity period expires but also that this node has not been excluded from the service. Conversely, older still valid references can include nodes that have been excluded from the set of service providing nodes by the service manager. It is assumed that excluded resource provider might try to attack the service or the binding process of the service.

Thus, the second threat scenario addresses attacks to the binding process by excluding

nodes that formerly supported a service. The authentication of nodes is not sufficient to prevent attacks by former supporters, as they can be included in valid references and, therefore, are able to inhibit the binding process by pretending to support and host the currently bound service.

A solution to the second threat scenario is offered by the introduction of a public-key-signed checksum. On creation of a service, a public-key pair is generated. Then a secure hash (SHA-1) [DEJ01] is built over all profile components and some general reference components of the IOR, like its object id. This hash is signed by the private key that has to be kept secret by the manager of the service. The checksum and the public key are added as new components to the profile. This enhanced time-bound object reference now enables a secure binding process that is outlined in the following.

If the IOR could be acquired via a trusted party and the service holds the guarantees given by the validity period, the following protocol (see Figure 3.4) ensures a reliable and safe binding of a service:

As long as the validity period of an IOR is not expired, at least one of the peers is still hosting the service. To resolve the most recent version of the reference, every accessible peer included in the IOR is asked to provide its currently held version of the IOR. It may appear that one or more peers provide a newer version of the reference. Each time, the checksum and the signature of the checksum are verified. If the checksum and the signature are valid, all accessible peers referenced by the new IOR are contacted. It is assumed that an attacker cannot provide a new version of the IOR, as she does not possess the private key. This process continues until a stable state is reached where all accessible peers provide the same version of the IOR. Then this most recent version of the IOR is used to connect to the service. Depending on the structure of the service and the communication mechanisms there is still a possibility to become a victim of a former service member that attacks the binding process. Between the search of the most recent IOR and the actual binding of the service, the service distribution and thus the set of supporting nodes could change. Therefore, after connecting the service and before submitting requests, the most recent IOR is evaluated again. If the IOR is identical, the binding process is finished, otherwise the connection is aborted and the service restarts connecting peers on the base of the new IOR version. Afterwards the reference is checked again. This goes on until a version of the IOR before connecting to the service is identically to the version after connecting to the service. If there are reference changes after this point, it is assumed that excluded peers immediately drop the connection to all currently bound clients, thereby triggering a restart of the binding procedure. Thus it is assumed that a node resigns properly and then might become an attacker. Otherwise not only the binding process has to be secured but as well the service has to be Byzantine fault tolerant. This is not considered in the scope of this thesis.

If there are many distribution changes in a short time the search for the current version of the IOR possibly never ends. In practice, this shall not be an issue. If all peers participate only for very short periods of time such a high fluctuation rate requires very short validity periods. Such small periods reduce the benefit of the overall concept and also reduce the usability of the service. Therefore, high node fluctuation should be generally avoided. If the

service is hosted by a quite stable set of peers and there are additional peers that change rapidly, these additional peers should not be included into the IOR, if possible. Finally, the usage of the location-service extension described in Section 3.4.5 also provides a solution for services that frequently change their distribution.

#### Autonomous Reference Updates in Context of a Secure Binding Process

In Section 3.4.3 two possible update schemes were proposed. These schemes also apply for the two proposed public-key–based extensions of time-bound references. If former supporters of a service could turn into attackers, none of the hosting peers should possess the private key and therefore have the possibility to provide arbitrary versions of the reference. Instead, the service manager as an outstanding party holds the private key (e.g., on a smartcard). Depending on a membership policy, she signs a new version of the IOR and publishes it via an administrative method of the service. This procedure has the drawback that the key holder has to be available every time a new reference version is needed. Thus, she represents a single point of failure outside of the service. This is a not justifiable restriction as replicated decentralised adaptive services should be able to change their distribution autonomously without external intervention.

By taking a closer look at the second threat scenario, it turns out that only excluded supporters of a service are considered to become attackers. If a service manager passes a membership policy to a service that is *static* — in a way that it leads always to the same decisions and as long as such a policy is valid — no nodes that have supported the service based on this policy will be excluded and therefore can turn into attackers. This condition holds even if nodes drop their support for example due to a host crash or maintenance reboot as they can rejoin the group of service-hosting nodes after recovering. If one or more nodes could drop their support and attack the service without a policy change the service needs to be Byzantine fault-tolerant. Basing on this assumption it is possible that a service holds a static policy and according to that, changes its distribution and reference. A good example for a simple policy might be that all nodes within a certain IP range are allowed to support the service. At this point, it is assumed that the network hosting the IP range is secured and the IPs are not dynamically assigned.

The critical point of this approach is the change of the membership policy by the service manager. This might result in an exclusion of formerly supporting nodes that afterwards might try to attack the service binding.

Taking this into account not every single version of a reference has to be signed by the key holder but only versions that exclude nodes due to a policy change have to. This leads to two kinds of references, *unsigned* and *signed* ones, resulting in an extended binding process.

Despite a policy change, every non-expired reference at least includes one peer that still belongs to the service that provides the most recent valid version of the IOR. However, after a policy change there might be non-expired unsigned references that contain some nodes that have been excluded according to the new membership policy. These nodes might try to attack the binding process by providing their own unsigned references if asked for the most recent reference version. This is possible if a service starts the binding process with

a slightly outdated non-expired reference that contains excluded nodes and that precede the signed reference, which excluded them. In the first place, a valid reference cannot be distinguished from a malicious one, as both of them are unsigned. Consequently, there might be several contradictory reference histories. These reference histories can be compared to branches. To distinguish the attackers from the true members of the service, one has to select the right reference history branch. This can be done by collecting the reference histories of all branches and scanning them for a signed reference. This is performed by connecting all nodes of reference and asking them for the latest reference version as done by the previous binding process. There has to be a signed reference, as this is the only reason a branch can occur as only a policy changes can be the cause for exclusion of nodes that try to attack the binding process. Furthermore, the branch with the signed reference has to be the right one as attackers cannot provide a signed reference, as they do not possess the private key of the service.

The process of detection of a signed reference is followed up by detecting a branch. This can possibly occur multiple times depending on the number of policy changes and the length of the validity periods. The branch that includes the latest signed reference is the right one and points to the service that can then be bound in the same way as for the basic secure binding process.

### 3.4.5 Integration of a Location Service Extension

Although the validity period provides good support to keep track of distribution changes of a service, there might be situations leading to an expired IOR. For example if all peers of a service change rapidly and the validity period is comparatively short or in cases where a node leaves the internet longer than expected. Aside from these cases, there might be service implementations other than replicated decentralised adaptive services that do not provide the necessary client-side mechanisms to refresh the IOR information. These services might miss update mechanisms due to the necessary implementation effort or the used internal communication protocol (e.g., a simple streaming protocol that does not offer additional control messages). Of course, this might have security implications, which have to be considered. Putting these three examples together providing additional mechanisms enabling a service binding despite an expired reference is desirable.

A solution offers the integration of a location service. This can be achieved by adding a supplementary *location-service reference* component to the proposed IOR profile. This component is simply another (embedded) IOR that references the location service of the depending service. If the IOR of a service contains a reference to a location service, this service is bound at first by the Aspectix profile manager (see Section 2.4). The location service fragment is then handed over to the local service fragment. The fragment implementation has now the choice to use the peers provided by the original IOR to start the connection procedure or to ask the location service for the latest reference version.

### Application Interface of the Location Service

The location service could have any interface because the fragment implementation receives a normal object from the profile manger during the binding process. The API outlined in Figure 3.5 is applied in the context of this work, but a developer is not limited to it.

```
interface LocationService {
    void registerIOR(in string ior, in boolean observe)
        throws AlreadyRegistered, InvalidFormat;

    void updateIOR(in string ior)
        throws InvalidFormat, NotAuthorized;

    string getCurrentIOR(in string ior)
        throws UnkownObject, InvalidFormat;
};
```

Fig. 3.5: Location service interface

The proposed location-service interface supports a push- and a pull-based update of IORs. If the default fragment implementation at the client-side is not able to keep track of IOR changes the push-based update is suitable. Regarding replicated decentralised adaptive services the pull-based approach is applied.

On creation of a service, its IOR is registered at the location service via the `registerIOR()`-method. This method takes one parameter that indicates whether the location service should observe distribution changes of the register service or not, according to a pull- or push-based update strategy. If the service was previously unknown to the location service and the IOR string is well formatted the object is registered, otherwise an appropriate exception is thrown.

At any point in time, the IOR of a registered object can be updated via `updateIOR()` under the precondition that the provided version is newer than the already registered one and has a valid signature.This method is usually only needed for a push-based update strategy. For securing the update of non-signed references additional support is required that is not further described as replicated decentralised adaptive services use a pull-based approach.

If the default fragment implementation of a service (e.g., a replicated decentralised adaptive service) keeps track of all distribution changes of a fragmented object there is no need for the service to announce these changes to the location service. Instead, the service advises the location service to observe distribution changes by setting the observe flag parameter handed over to the `registerIOR()`-method. The location service only has to bind the reference and the locally bound fragment implementation keeps track of all distribution changes and consequently reference changes.

If a client binds an outdated or even stale reference that includes a location service component, the locally instantiated fragment implementation can determine the most current version by using the location service. This is achieved by calling the `getCurrentIOR()` method of the location service, which requires a previous version of the reference.

### Possible Implementations of the Location Service Extension

At the first glance it seems like a location service extension introduces a well-known service that originally has been proposed to be avoided. However, either this service itself should have a time-bound reference as it can be implement as a replicated decentralised adaptive service so there is no fixed set of nodes introduced or by using a public peer-to-peer network.

As described in Chapter 4, peer-to-peer services can be easily wrapped as fragmented objects and therefore seamlessly replace a plain CORBA object or a replicated decentralised adaptive service. Such a peer-to-peer based location service simply distributes all reference information starting from the last service-manager signed version up to the most recent one in the scope of a peer-to-peer network.

This approach entails two possible threat scenarios. The first one is the possibility that nodes of the peer-to-peer network could modify and republish unsigned references and thereby attack the binding process. The second one relies on the fact that nodes that are responsible of hosting information about a reference could drop this information and therefore prevent the binding process.

The second issue is tolerated, as peer-to-peer networks usually comprise many nodes ranging from hundreds to tens of thousands. Thus, the chance to inhibit a certain piece of information is quite complicated. This is especially the case for frequently changing data that is republished quite often.

The first issue is severe and has to be eliminated. Attackers might simply publish modified references that cannot be distinguished from the true references of a service. This is especially problematic as the approach to control the history chain detailed in Section 3.4.4 is no longer sufficient as a client may start the binding process with an expired reference. Accounting this fact, a peer-to-peer network has to store a reference chain that starting with the latest expired signed reference is followed up by a number of unsigned expired references and continues with the valid unsigned references up to the most recent version that has been published so far (see Figure 3.6). At this point an attacker is still able to lookup a reference with a partly expired reference chain. She uses this information to publish an alternative reference chain that starts immediately after the signed reference including several manipulated expired unsigned reference up to still valid references that point to nodes owned by the attacker. So far a client cannot determine which reference history chain is the right one as for expired references there is no peer that is guaranteed to still belong to the service.

To prevent this attack the concept of a *self-signed time-bound object reference* is introduced. Each replica of a service including a new member signs the hash of the reference and includes this signature into the IOR. If a node crashes, the remaining nodes create a new version and again sign the hash of the reference. This way a reference history cannot be modified, as there is a signing chain starting from the service manager signed one to the most recent, bridging the border between the invalid and valid versions of an IOR. Figure 3.6 outlines this context. A client may hold an expired reference that is required to find the last published manager signed reference and the reference history since then up to the last valid unsigned reference.

New members cannot be attackers because the current set of nodes included the new peer according to the policy. Former members shall not attack the binding process because the policy has not been changed. If it is changed, the responsible service manager will have to create a new signed reference and publish this in the peer-to-peer network. External attackers cannot create a new reference, as none of the current or former members of the service would sign a modified reference, which would be needed to be accepted by a client.

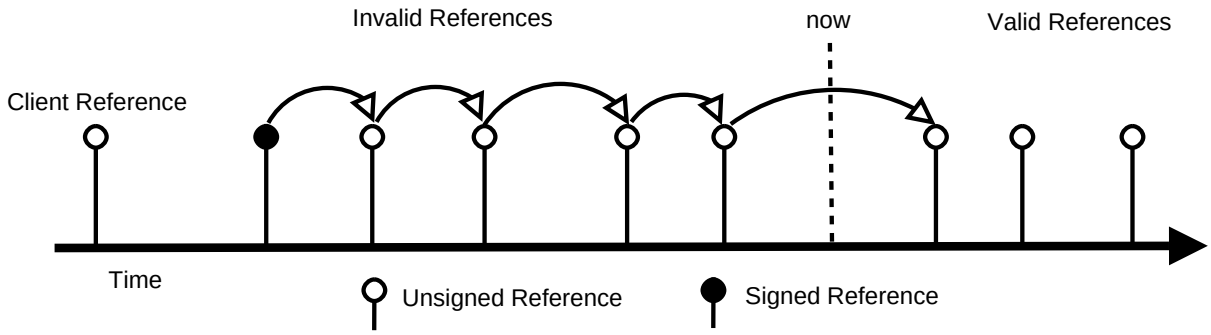



Fig. 3.6: Reference version history of a time-bound object reference

### 3.4.6 Use Case: Reference Management of a Decentralised Version Control System

To clarify the realisation of time-bound object references and the associated binding process, a brief example is given that outlines how a distributed, fault-tolerant source code repository, like the one described in Section 3.10, handles the reference management.

As the service is a replicated decentralised adaptive service its functionality is split into two fragment implementations: A client-side access fragment, which provides an intelligent stub hosting a copy of the repository, and a replica fragment implementation that realises the replicated repository storage. An instance of the service is, for reasons of fault tolerance, usually composed of at least three serving fragments and depending on the demand of an arbitrary number of access fragments. State changes to the repository are propagated from an access fragment to one of the serving fragments and then broadcasted via the group communication system to all replicas (see Section 3.3).

If a new serving fragment is added, an existing one is shut down, or a serving host crashes, a group communication system following the *virtual synchrony model* [Bir97] as used for replicated decentralised adaptive services sends a *view-change* message to indicate a group membership change. This membership change is further on controlled to be in tune with guarantees given with previous, still valid IOR versions. If a valid IOR version is at risk to expire before the validity period is over, further leaving and changing of serving fragments is inhibited. After each view change, every serving fragment creates a new version of the IOR. If there are no view changes for a long time and the most recent IOR version is at risk to invalidate, one of the serving fragments, usually the coordinator of the group communication infrastructure detects the imminent invalidation and announces a new validity period via the group communication. On reception of such an announcement, each serving

fragment creates a new version of the IOR with the extended validity period. This update process is equivalent to the on-update-or-expiration policy. The access fragment caches the IOR to keep the connection to the service. If the IOR is at risk to expire, the access fragment asks the replica fragment, it is connected to, for the current version. If the IOR is requested from the client or the middleware the access fragment also asks the replica fragment for the most recent version.

In context of the example, one could assume a validity period of an hour. Further, we assume five replica fragments. Up to two of them could fail during a validity period. As one replica fragment has to be available for the whole validity period, a maximum of two nodes can be exchanged during the validity period of a reference. Thus, a complete redistribution of the service takes at least more than two hours. Of course, using the proposed location service extension and a shorter validity period can reduce this time.

If the service manager connects the service to change the membership policy (e.g. by removing a domain from the list of supporting institutions), it binds to the service as any other client and passes a credential to the service to get access to the administrative interface. Next, the following three steps have to be taken: First, the member policy is changed. This results if necessary in the shutdown of replicas at the excluded domains and the set-up of new replicas at other locations. Finally, the new reference version is signed. In a worst-case scenario where all replicas of a service are hosted by an excluded domain, the redistribution takes at least until the most recent reference version expires.

### 3.4.7 Related Approaches

The concept of time-bound object references could be compared to leases [GC89]. Middleware systems like Jini [Mic01] or .NET [Ric02] use leases to de-allocate resources after a certain period of time. If a lease times out and is not renewed, the server can safely reassign the resources to other clients. It is in the interest of a server to provide leases that are only valid as long as the client needs the resources. In contrary, time-bound object references are not used to assign resources but to keep contact information up-to-date. Moreover, it is in the interest of all parties to provide long-durable references.

Various middleware systems offer mechanisms for object mobility or support the mobile agent paradigm. Emerald [JLHB88] and SSP-Chains [SDP92] try to prevent the invalidation of references by leaving forwarding information at previous locations of the object. If this fails, Emerald will contact registry services and, if this does not help, an exhausting search over all nodes will be performed. Thus, Emerald uses the two techniques, request forwarding and a location service that have been outlined at the beginning of the section. Both approaches primarily target the mobility of non-replicated services but can be applied in a straightforward way to replicated services. Still these systems rely on one or more nodes that have to be accessible at all time and miss a secure binding process to address changes of the supporting node set.

Essentially the same applies for the handling of references as defined by the FT-CORBA specification. It is assumed that a service cannot change its domain and consequently it is not considered that a node of a domain can be excluded or might even become a malicious party after the service has moved to a new location.

The FIPA [Fou04] has specified the *agent naming reference model identifier* to access mobile agents. Such references consist of a set of physical addresses at which the agent could be located. If this is not the case, the reference provides additionally contact information of resolving services. The specification assumes statically available resolving services and offers no support for distributed objects, because a FIPA-compliant agent is always located at one physical site. Again, only basic mechanisms like a location service are provided and a static set of nodes is assumed but no extended approach for a secure binding is supported.

### 3.4.8 Summary

This section targeted the problem of outdate and stale object references due to service migration. Common basic approaches, like request forwarding, or using a location service rely on the assumption that there is a static set of nodes that either are given by the previous locations of an object or the nodes hosting the location service. It has been argued that there is no such static set in case of dynamically distributed systems. This especially applies for EDAS and replicated decentralised adaptive services.

As a consequence of this lack of an appropriate mechanism for reliable referencing replicated services in dynamic environments, the concept of a time-bound object reference was proposed. Such an object references gives, apart from the necessary information how to access a service, a time-bound guarantee for a successful binding of the service despite distribution changes. The basic concept was then extended to enable a safe object binding, as replicated decentralised adaptive services are not executed in closed environments and not all nodes in an open distributed environment can be trusted. The extension of the concept ensures a safe binding despite former supporters of the service that have been excluded from the group of resource providers try to attack the binding process.

Whereas time-bound object references and the extension for a secure binding process offer a good solution for binding services in a dynamically distributed system on the downside migration and redistribution of services is restricted by the availability guarantees given by the references. To alleviate these restrictions validity periods can be shortened. This demands for the proposed mechanism of an integrated location-service facility that itself can expose a time-bound reference or be implemented on top of peer-to-peer network thereby omitting the need for a central well-known service.

## 3.5 State Transfer

Reliable referencing and binding is only one issue that has to be handled when providing decentralised adaptive services that are dynamically distributable. Even more important and fundamental is the support for the actual state transfer process that is required for initialising stateful service replicas. The state transfer provides the base for setting up replicated services, replacing failed replicas and supporting dynamic service distribution in general.

Previous fault-tolerant CORBA implementations and their underlying group-communication frameworks offer only limited support for state transfer as the integration of new replicas

is considered a seldom task that is only needed at creation time of a service and to replace crashed replicas. This is not the case for replicated decentralised adaptive services as they dynamically adapt their distribution making state transfer a much more frequent task. Three limitations of current group-communication frameworks and frameworks for fault-tolerant object replication have been identified. First there is limited developer-support for serialisation of the in-memory application state, especially for heterogeneous environments. Another deficit of previous work is the fact that it does not consider large application states. Recent fault-tolerant service replication frameworks require an application to provide the whole state to the group-communication framework as a single unit (e.g., as a byte array) that is forwarded by the framework as one or more chunks of data to the joining replica. This limits the state size due to the available process memory. Third and most important are unnecessary long state transfer operations especially for larger application states. This decreases the availability of a service, as requests have to be delayed during at least parts of the state transfer operation. Furthermore, a long state transfer reduces the fault-tolerance of a replicated service, as it takes longer to recover crashed replicas.

Summarising, extended support for state transfer is missing at this point. Accounting this fact this thesis aims at providing better support for state transfer that explicitly speeds up the state exchange and supports arbitrarily large application states. First, this is achieved by offering an extended stream-based interface for state transfer, enabling the handling of arbitrarily large object states. Second, this is enabled by providing extended support for state serialisation by using IDL and especially CORBA value types. This allows to describe the in-memory state of a service in implementation language independent format. In contrary to a manual implementation of state serialisation methods this is less time consuming and error-prone for large object states. Third and most important, the process of state transfer itself is accelerated and improved by providing support for non-blocking and parallel state transfer among other techniques such as utilising synchronisation mechanisms.

The following section outlines the state of the art of state transfer for non-replicated stateful entities and replicated stateful services. This provides the background for the proposed extended mechanisms, starting with the proposal of a stream-based state transfer interface. Then the format of the transferred state is discussed and a platform independent solution based on CORBA value types is suggested. To provide the necessary basis for the proposed parallel state transfer algorithms, Section 3.5.4 summarises existing approaches for parallel data transfer. Section 3.5.5 presents the main contribution of this section; two adapted algorithms for parallel state transfer. Further on it is discussed how synchronisation techniques can improve and speed up state transfer. Section 3.5.7 proposes an optimisation for a non-blocking state transfer in the context of large application states. The prototype of the implemented state transfer support is outlined in Section 3.5.8 followed by an experimental evaluation.

### 3.5.1 State of the Art

The migration of a non-replicated stateful entity (e.g., an object, an agent, or a process) as well as the creation of a new replica rely on state transfer. Thus, the state of the art in both areas is briefly summarised to provide a basis for the proposed techniques.

### State Transfer of Non-Replicated Entities

To successfully migrate a non-replicated stateful entity from one node to another, several steps are required. First, at the origin location the entity has to reach a consistent state. Usually, this is achieved by delaying or even blocking external inputs and wait until all ongoing activities inside this entity have finished. If such a consistent state is reached, a snapshot of the entity state can be taken. This snapshot is serialised and transferred to the new location. At the target node the snapshot has to be deserialised. Finally, the execution process of the entity is restarted and blocked external inputs are redirected to the new location so they can be processed there.

In principle, the transferred state of a stateful software entity can consist of program code and the execution state. In context of this section, only the execution state is considered. However, one cannot assume that in dynamic systems like EDAS the program code of an arbitrary service is available at every target location. Therefore, Chapter 6 addresses the issue of dynamic loading of platform-dependent code.

The execution state of a stateful software entity can be captured at different levels. If the whole execution state is taken as this is for example done by process migration systems [MDP+00] or approaches that support the migration of threads [SH99] it is called *strong migration*. If only the application state is captured without system runtime information, it is called *weak migration*. The latter approach is used by most mobile agent systems [MLC98, BRST02, LO98] as it provides sufficient support at application-layer and can be implemented on top of standard middleware systems or provided as a service by the platform. Strong migration in contrary requires deeper support by the operating system or at the level of a virtual machine.

Apart from providing easy to use migration support that is transparent to clients, migration speed has always been an issue, as a stateful entity usually cannot respond to requests during migration. For systems, supporting strong migration a snapshot can be taken that is transferred to the new location while still processing requests at the old location. As the snapshot is outdated by the time it is initialised at the new location one cannot immediately switch over but has to update the snapshot by transferring small deltas capturing the state changes since taking the snapshot. Such an approach has been evaluated for process migration [Rou96] but also for the migration of whole virtual machines [CFH+05]. This does not speed up the state exchange but improves the availability of the migrated application. A slightly similar approach is presented in the following section for replicated services. In context of agent system it has been evaluated how to modularise an agent by phases to reduce the state that has to be migrated [Gei01]. However, this assumes that only parts of the agent state are required at a target location. Thus, this approach cannot be applied in the context of replicated services.

### Approaches to State Transfer for Replicated Services

The basic workflow of migrating a non-replicated stateful software entity can be compared to the integration of a new replica. First, all replicas have to reach a consistent state. This is achieved by delaying requests. Then a state snapshot is taken. This snapshot is transferred

to the newly instantiated replica that is initialised using the snapshot. Finally, the request processing is restarted and all blocked requests are executed. This can be named a *blocking* state transfer as all replicas stop request processing for the whole state exchange.

Although there is a multitude of group communication toolkits and object replication frameworks, state transfer is often treated in a similar way. Cabañas and Mestras [CM00] give a brief overview of existing approaches. Early systems such as Arjuna [PSWL95] and Electra [Maf95] do a state transfer automatically when a new member joins a replication group, thereby blocking the whole system as outlined by the basic workflow description. Whereas the former requires a state transfer in one step, the latter provides support for multiple steps, enabling a transfer of larger object states. In Phoenix [Mal96], only the members involved in the state transfer are blocked. However, this system also requires a state transfer in one step, offering no support for large object states.

All three systems block at least some of the group members for the time needed to acquire the state from an application and the entire duration of the actual state transfer. This is sufficient for transferring small application states but leads to long response times for larger ones [Bir97]. To avoid such long response times, recent systems such as JGroups, Eternal [NMMS01] and also Birman [Bir97] provide solutions in which the system (apart from the joining node) is blocked only for the time of the state acquisition from the application. Throughout the text, this is called a *non-blocking* state transfer approach.

The prototype implementation of the framework for replicated decentralised adaptive services is based on the JGroups framework. In JGroups [Ban98b] a node can join a communication group and then in a second step request the state. As Figure 3.7 shows, first a `take_snapshot` message is send to all members including the requesting node. On reception of this message, all members start queuing further incoming messages. Furthermore, all fully functional members process all requests that have been received before the `take_snapshot` message according to their order. When the application is in a fully quiescent condition a snapshot is requested and store it in memory. Long running operations can delay the state transfer process and therefore should be avoided. After acquiring the application state, the members proceed processing the queued messages. Thus, all fully functional members are blocked only for the time to get the state from the application. The requesting node proceeds queuing incoming messages and requests the state from one of the members (usually the oldest one) by sending a `get_state` message. This replica returns the state. On reception at the joining replica, the state is passed to the application, which is now initialised and ready to process requests. Afterwards the message queuing is stopped and all enqueued messages are forwarded to the application. In a final step, the new node sends a `GC` message to all nodes to indicate that it is now save to destroy the state copy. This protocol requires a total ordering of messages, as the `take_snapshot` message has to be delivered and processed by every member at the same logical time. Otherwise, the requesting node could start queuing messages before a state-providing node and later on forward messages to the application that are already included in the application state, resulting in an inconsistent state.

To give a more complete overview about state transfer approaches the checkpoint-based state transfer is outlined in the following. Usually applied for passive replication, it also

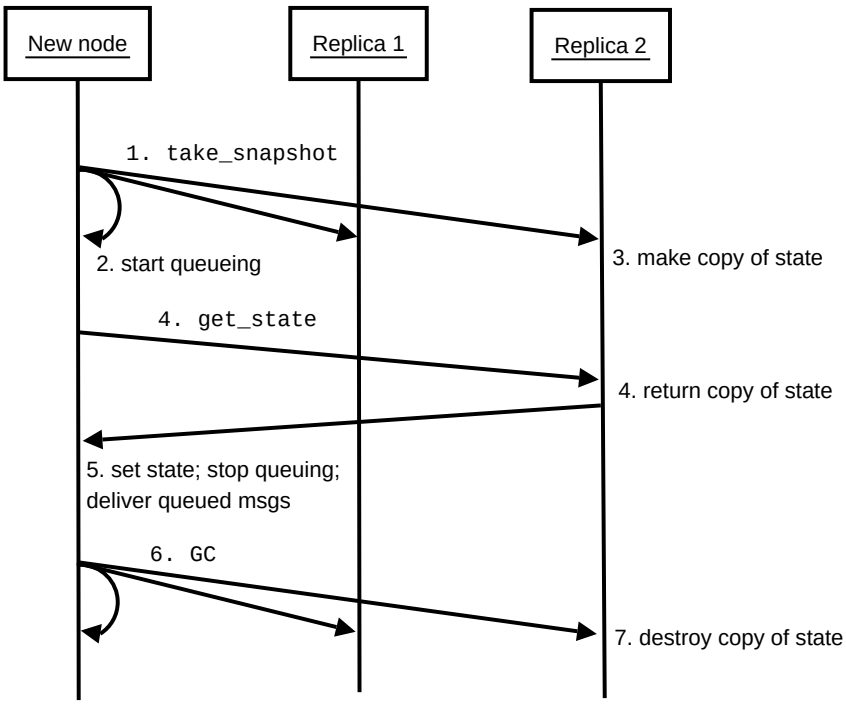



Fig. 3.7: Basic non-blocking state transfer protocol

can be used for actively replicated objects. An early description using checkpoints for state transfer can be found in Mishra et al. [MPS93]. Castro [Cas01] uses checkpoints for state transfer to initialise new replicas in a Byzantine fault-tolerant setting. In the following, a basic variant of a checkpoint-based state transfer protocol for actively replicated objects is described.

Every replica writes all incoming messages to a message log. If the log reaches a certain size, every replica saves a checkpoint to a stable storage and truncates the log. A joining node sends a `take_snapshot` message to all members including itself. On reception, all fully functional members write the message to their message log as a marker and possibly delay log truncation until the sending node signals the successful state transfer. Receiving the `take_snapshot` message the requesting node starts logging all further incoming messages and sends a `get_state` message to one of the group members. This member returns the latest checkpoint together with the message log including all messages before the `take_snapshot` marker in the message log. Then, the new node initialises the replica by the checkpoint and forwards the messages included in the message log provided by the replica. Finally, it forwards the messages from the local log to the application. To finish the state transfer the new node sends a message to all members indicating that the state transfer has been successfully completed. If the message log has reached or further on reaches the maximum size the replicas can safely make a new checkpoint and truncate the log.

In contrast to the basic non-blocking state transfer protocol, checkpoint-based state transfer approach offers the possibility of an immediate transfer without any delay of requests, which means that the state does not need to be acquired from the object. However,

this is not for free as the message log has to be maintained and the periodically taken checkpoints introduce additional overhead. Taking a checkpoint requires the same time as acquiring the state from the object in the basic non-blocking state transfer protocol. More severe the number of checkpoints might overweight the number of joining nodes by far. Thus, a checkpoint-based state transfer system has an overall higher overhead. To reduce the overhead it would be very useful to distinguish state mutating messages from non-mutating ones, as the later need not to be included in the message log. This results in less logged messages and consequently less checkpoints. In context of the prototype implementation, this call context information is already provided by the annotation of read-only operations as described in Section 3.7. Finally, a checkpoint-based state transfer protocol supports the reintegration of temporary disconnected nodes (e.g., due to system reboot). If a replica was only disconnected for a short period, the replication group has probably not taken a new snapshot and only the missed messages have to be extracted from the message log of one of the remaining replicas and provided to the rejoining member.

Figure 3.8 summarises the three presented state transfer approaches. The following sections target the improvement of non-blocking and blocking approaches of state transfer. Some of the proposed techniques are also relevant to the checkpoint-based state transfer approach but this is not detailed any further.

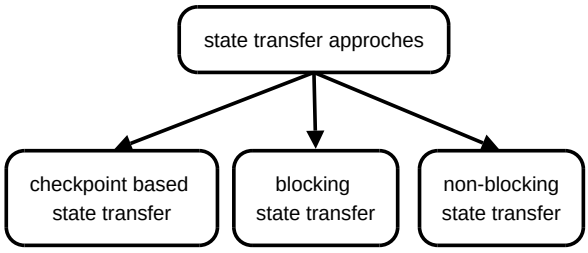



Fig. 3.8: State transfer approaches

In the domain of replicated database systems, existing work covers the recovery of replicas using the coordination support offered by group communication frameworks [KBB01, JPPMA02]. Unlike the approaches discussed above, these systems primarily target the recovery of replicas by using system properties of databases. Thus, the proposed concepts cannot be directly applied to object replication systems.

### 3.5.2 State Transfer Application Interface

To support the state transfer of arbitrary large application states an appropriate interface that standardises the state exchange between the application and the replication framework is an essential prerequisite. Currently, most systems provide a simple array-based state transfer interface that limits the state size to available process memory. In contrary, this thesis proposes a stream-based interface that is suitable to transfer arbitrary large application states.

### Simple Array-based State Transfer Interface

The FT-CORBA specification as well as most group communication frameworks (e.g., JGroups) provide a simple state-transfer interface that has to be implemented by a fault-tolerant service to enable the creation of replicas. Such an interface usually offers two methods, `get_state()` for obtaining the state of an object and `set_state()` for setting the state of a new or rejoining replica (see Figure 3.9). Called by the middleware these methods pass the state as a single byte array from a replica to the group-communication framework and vice versa on the receiving side. This limits the state size, as the state has to be stored entirely in memory during the transfer. Sometimes even multiple copies of the state exist as the state is copied between the different layers of the middleware and the affiliated group-communication framework.

```
typedef sequence<octet> State;

interface Checkpointable {
    State get_state() raises(NoStateAvailable);
    void set_state(in State s) raises(InvalidState);
};
```

Fig. 3.9: FT-CORBA interface for state transfer (IDL)

### Stream-based State Transfer Interface

This work proposes a similarly simple but *stream-based state transfer interface* (see Figure 3.10), which has to be implemented by every replicated decentralised adaptive service. If a new replica is instantiated, the framework for replicated decentralised adaptive services advices the group communication framework to join the communication group of the service and request the service state. After successfully joining the group but before requesting the state, the group communication creates an input stream. This stream is then handed over by the group-communication for replicated decentralised adaptive service to the new replica by calling the `setState()` method of the state transfer interface. This way the group communication can immediately forward junks of the state to the new replica as they arrive. This has two benefits, first the state does not need to be stored entirely in memory and second the state can be deserialised by the new replica in parallel to the transfer.

On the state-providing side, a similar process takes place. The group communication framework notifies the framework for replicated decentralised adaptive services about a state request and provides an output stream. This stream is passed to the replica via calling the `getState()` method of the state transfer interface. The replica then serialises the state and writes it to the stream. The group communication framework reads the stream and transmits the data to the new replica. Thus, also on the state-providing side only small parts of the whole state have to be stored in memory, as the stream enables the on-demand serialisation and transmission of data.

```
public interface StateTransferIfc {
   void getState(OutputStream out) ;
   void setState(InputStream in);
}
```

Fig. 3.10: Stream-based state transfer interface (Java)

### 3.5.3 Transfer Format

Independent of the state transfer interface in use, the FT-CORBA specification leaves the state transfer format to the developer. Therefore, a developer has to manually implement state transfer methods for each object implementation, offering full control how to serialise and deserialise the state of an object. The current prototype of the framework for replicated decentralised adaptive services supports this approach because it is simple, lightweight and flexible. However, implementing serialisation methods on a per-object basis is costly in terms of implementation effort and error-prone in case of large and complex object states. Therefore, two transfer formats supporting the automated serialisation and deserialisation of object states are discussed in the following. Whereas the first approach analyses the use of language-supported mechanisms, the second approach discusses the usage of CORBA values types. The first approach is straightforward and applied by many systems requiring state transfer not only enabling active replication but also to support migration (e.g., for mobile agents). The second approach is novel, as it proposes to specify not only the interface but also the state using IDL. This ties the state transfer support neither to the implementation nor to the used implementation language.

#### Using Language Provided Serialisation Mechanisms

A straightforward approach would be to use native language serialisation mechanisms provided by the implementation language. In case of Java, the implementation language of the prototype, a class only has to implement the marker interface `java.io.Serializable`. All objects extending this interface can easily be serialised and deserialised to a stream. On the downside, native language serialisation formats usually tie the state of an object to a certain implementation and the implementation language itself. Additionally minor implementation changes may result in incompatible versions. In case of the Java programming language, the consequence is a serialisation error on the receiver side, at the new replica. An example for such a change is the deletion of a field. To prevent these kinds of errors one could generally suggest using the same implementation on all participating nodes. This might be complicated in large installations comprising hundreds of nodes as not only the same application code has to be provided at every place but also identical dependent third party libraries and, in case of Java, the same Java class library.

One way around this problem might be the reduction of the transferred object state. If a certain field is not needed to provide the actual service functionality, as this might be the case of tuning parameters and other implementation-specific variables it should not be included in the transferred object state. This reduces the probability of incompatible changes

and serialisation errors due to implementation evolution. A good example is the `Hashtable` class of the Java class library, which includes numerous internal variables, but the relevant state that should be transferred comprises only the key-value pairs of the object instance. Java offers two solutions to solve this issue. There is the keyword `transient`, marking variables that should not be serialised and a developer can provide custom serialisation methods. The basic idea behind the `transient` keyword is to exclude certain variables from serialisation that cannot be automatically transferred, such as socket connections or file handles. It allows excluding variables from application classes, but does not provide a solution for classes provided by third-party libraries or the Java class library. The second approach requires a developer to manually implement serialisation methods for each individual class. Such an approach has for example been taken by JGroups: all communication messages provide custom serialisation routines. This is time consuming to implement and as error-prone as writing serialisation methods from scratch, as proposed by the FT-CORBA standard.

#### Serialisation Based on CORBA Value Types

To loosen the coupling between object state, object implementation and implementation language, automated support for state transfer via value-types based on previous work [KSH05] offers a better solution. Instead of just defining the interface of an object, a replicated decentralised adaptive service can be defined as an IDL value type, which specifies, besides the service methods, the object state in terms of IDL data types. In standard CORBA, value types can, on the one hand, be automatically passed by value, which allows automated state transfer; on the other hand, they can be activated and become remotely accessible like any other CORBA object. In short, value types support object-by-value semantics. If a service is specified as a value type, object-specific serialisation code will be generated similar to standard CORBA value type code. For the developer, object state transfer is transparent, as she does not need to implement special methods. Instead, the defined state is serialised and deserialised on demand.

Yet, it is expected that decentralised adaptive services possess not only in-memory state but also state that is stored on hard disk. This *disk-dependent* state is typically not captured by the value type description. Instead, the object has to provide additional custom serialisation methods by implementing the state transfer interface outlined in Figure 3.10 that are executed after serialisation and deserialisation of the value type state. The advantage of this twofold approach is that the generic state definition in IDL allows an easy state exchange between heterogeneous replica implementations and still custom serialisation methods can be provided to serialise the disk-dependent state of a service. Of, course this approach can be also used in conjunction with a language-dependent serialisation format.

Using value-types for state description allows not only the state transfer between different versions of an implementation, but offers in principle the opportunity to support the state exchange between implementations that have been implemented using different programming languages. In previous work [KSSH06] it has been validated how to modify and adapt objects specified as value types to provide adaptive mobile objects but so far this approach has not been introduced to replicated decentralised adaptive services. However,

this kind of state adaptation would offer a basis for online software updates as it allows transferring state between different value types. Finally, implementation-independent state description might be beneficial in a Byzantine fault-tolerant setting, if multiple independent replica implementations are provided to tolerate implementation faults.

### 3.5.4 Parallel Download Approaches

Considering the previously outlined approaches to state transfer, usually every member of the object group acquires the application state, but only one dedicated member is actually involved in the transfer. This is sufficient for small application states but causes high load for the providing member in case of larger application states, comprising hundreds of megabyte. Using parallel data transfer for state exchange balances the transfer-load over all fully functional members and usually reduces the overall integration time. Below a brief overview about existing approaches to parallel data transfer is given. Basing on these former results it is described how to provide parallel state transfer for actively replicated objects in the following section.

Rodriguez, Kirpal and Biersack [RKB00] propose two methods for parallel download named *history based TCP* parallel access and *dynamic TCP* parallel access. Both approaches require one dedicated unicast connection from the client to each of the providing servers. They assume that there is no common bottleneck in the network path. The first approach suggests adapting the transfer packet size depending on the available bandwidth of the accessed servers. The bandwidth information is gathered from a database that stores information about earlier transfers. According to the authors, history based TCP parallel access produces good results if the network and server conditions are constant. If these conditions change rapidly, the information provided by the database is outdated and consequently leads to poor performance.

Therefore, the authors propose *dynamic TCP* parallel access, which omits an external information source. A file that should be downloaded is divided into $N$ blocks of equal size. Then the client requests a different block from every server. If the client has completely received a block, it requests another one that has not already been downloaded from the same server. This simple approach assigns more blocks to faster servers, but fully loads all servers. The authors give some guidelines considering the number of blocks and block size. There should be far more blocks than servers. Blocks should not be too large, otherwise the approach is no longer dynamic and slow server will increase the download time. On the other hand blocks should not be too small so a block transfer takes considerable longer than the time to request the next block. Finally, the authors propose an approach to speed up the endgame of downloads by requesting a few last missing blocks of a file from multiple servers.

Rodriguez, Kirpal and Biersack already recognised that a server has an idle phase between the end of the transmission of a block and the reception of a succeeding request (see Figure 3.11), Rodriguez and Biersack discuss this in more detail in [RB02]. They suggest request *pipelining* to avoid these inter-block idle times. A new block should be requested at least one round-trip-time (RTT) before the current block is fully received. They evaluated the speed-up by using the average RTT on a per server basis of previous downloads. Fur-

thermore, they suggest to overestimate the RTT for the request pipelining as it eliminates the inter-block idle times and does not degrade performance.

Vazhkudai [Vaz04] proposes similar parallel access approaches like presented in [RKB00] and [RB02] but in contrary does not focus on clients accessing small and midsize documents but targets downloads of big data sets in a grid infrastructure. As a very simple approach, he proposed for comparison reasons *brute-force co-allocation*. Thereby a file is divided in $n$ equal parts, with $n$ corresponding to the number of providing servers, which are downloaded in parallel. This approach takes advantage of all servers but the time to transfer the whole file depends on the slowest connection and server. Another scheme proposed as *predictive co-allocation* matches the *history based TCP* approach of Rodriguez and Biersack [RB02]. Vazhkudai also proposes two dynamic approaches taking different server and network conditions into account. *Conservative load balancing* can be compared to dynamic TCP by [RB02] without pipelining. Lastly, he proposes *aggressive load balancing* that records the maximal transfer rate of all servers. Each time a server exceeds the maximal transfer rate its assigned block size is doubled. If the rate of a server is significantly less than the maximum rate, it is excluded from the download process.

### 3.5.5 Adaptation of Parallel Download Approaches for Parallel State Transfer

Two variants of the previously discussed algorithms were implemented and adapted to fit the needs of parallel state transfer in the context of actively replicated objects. Thereby the naming convention was taken over from Gkantsidis, Ammar and Zegura [GAZ03]. They presented the basic algorithms for parallel download of files that were extended to fit the requirements of a parallel state transfer.

*Static equal* assigns equal shares to all state-providing peers and uses smaller blocks to enable a continuous data flow. *Dynamic* can be compared to dynamic TCP [RKB00] but this work introduces a different approach for runtime optimisation. Before explaining the algorithms in detail, basic conditions and assumptions considering parallel state transfer are outlined.

In opposite to a typical download of a file, the state size is not known before a state transfer. An application provides the state as a stream to the group communication framework and the transfer protocol immediately starts to transfer the data. A possible solution would be to completely acquire the state from the application, determine its size, and afterwards start the transfer. In most cases, it is not possible to store the whole state of an application in memory, as it might comprise a large amount of data such as a whole database. Instead, one could temporarily store the data on disk and then read it for the transfer. However, this approach is time-consuming and slows down the state transfer.

Several of the presented algorithms for parallel file download do not only download a file from a set of nodes, but also either preselect the accessed servers depending on RTT and bandwidth or exclude slow servers during the download process. Both approaches are unnecessary for parallel state transfer as an object group, especially if actively replicated it comprises only a few nodes that should be well connected. In fact, if a node is highly loaded and therefore unresponsive, it might be excluded from the object group by the group communication framework. Consequently, a preselection of nodes does not need to

be performed by the state transfer protocol.

Rodriguez and Biersack [RB02] discuss the costs of parallel access for setting up unicast connections, signalling overhead and managing the download process itself. These considerations only partly apply to parallel state transfer as the group communication framework already maintains unicast and broadcast connections to all group members that can be used for the transfer.

Only a few algorithms for parallel download of files consider the transmission- and arrival-order of data. For state transfer this is a critical point, as the object replicas provide the data as a stream and the joining replica deserialises the stream to setup the in-memory state, which is needed to deserialise the persistent state to store it at the local disk. Jitter can be tolerated to some extent, as there are no strict timing constraints. Furthermore replicated decentralised adaptive services target not very resource-limited execution environments such as mobile devices where memory consumption is a critical factor. Thus, requesting the same data from several nodes, like done by Nebat and Sidi [NS06] to rule out jitter and possible node failure for the reliable transmission of multimedia streams, is not necessary.

A protocol for parallel state transfer has to signal each of the group members the part of the state it has to provide. Let $S$ be the set of servers providing a data $D$ of size $|D|$ for download. A data request is described by a tuple $< s_i, start, end >$, where $s_i \epsilon S$ and $start$ represents the first and $end$ represent the last byte of a requested byte sequence. A server $s_i$ returns a data reply defined as $< start, B >$ where $start$ determines the absolute position in the state and $B$ represents a transmitted byte sequence further denoted by a block. Returning the end position is not necessary as it is implicitly available by $end = start + |B| - 1$. If the requested block is not available as $start > |D|$, the response is $B = \{\}$. This is possible as $|D|$ is not known in advance as described earlier.

In the following, the two adapted algorithms for parallel state transfer are outlined in detail.

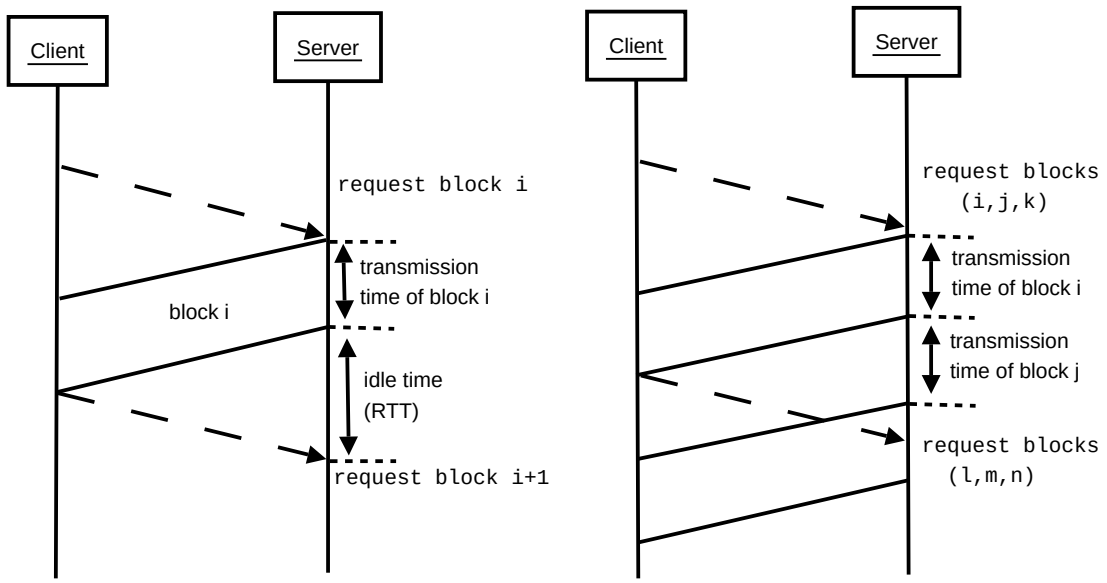
### Static Equal

A simple and easy request scheme proposed as static equal in [GAZ03] splits a file in $n$ equal pieces where $n = |S|$, hence every server has to provide one part of the file. This algorithm was adapted to fit the needs of parallel state transfer. As the state size is not known in advance, the state is divided in small blocks of static size. Each server should provide an equal amount of the state so the state transfer is organised in rounds. Each round every server is requested to transmit one block that has not been transferred so far.

The data requests of a round can be constructed as follows: $< i, nr + (i-1)|B|, nr + i|B| - 1 >$ where $n \epsilon \mathcal{N}_0$ and $i = 1, ..., |S|$. Each round assigns $r = |B||S|$ of data.

As the state size is not known in advance, we define $start_{max} = +\infty$ as an initial start value. If a response $a$ with $D = \{\}$ is received, $start_{max}$ can be computed by $start_{max} = min(start_{max}, start_a)$. All requests $start = nr + (i-1)|B| > start_{max}$ are discarded. As there might be out-of-order transmissions, one has to wait for all pending requests $start \leq start_{max}$.

On the state-providing side, the first request $b$ that arrives with $start_b \geq |D|$ causes the




(a) Basic request scheme

(b) Request scheme with static equal block assignment doing pipelining and batching

Fig. 3.11: Parallel-access approaches

server to send a response $B = \{\}$. All subsequent messages requesting data behind the end of $D$ can be ignored. The server still has to continue participating in the state-transfer protocol, as requests for blocks before the end position might arrive out of order.

Requesting small blocks is expensive in terms of control messages, as for every block a request message has to be sent. To reduce the number of control messages *batching* has been introduced. Instead of requesting one block at a time, several blocks are requested at once. Thereby the blocks match the assigned slot of a server in a configurable number of consecutive rounds $p$, called *batchsize*. There might be two alternatives to this approach: one would be to enlarge the block size so fewer requests are needed; the other is the introduction of a static request policy. As a continuous stream of data is required to reduce resource consumption at the receiver side and to support the deserialisation process of the state, a larger block size is not an option. A static request policy assigns each providing server a certain block in a round. Therefore, every server knows in advance that every n-th block in a round has to be transferred. This would reduce the number of control message to one at start-up. Although a static request policy seems to be very attractive, it might lead to problems if there are one or more servers that are considerable faster than other ones. These would provide their part of the state too fast. This leads to unnecessary buffering and resource consumption at the receiver side, as the other servers have not provided their part of state to generate a continuous stream.

Actually, control messages are not the key problem and batching allows already a reduction of the signalling overhead. In Section 3.5.4, it has already been mentioned that if a succeeding request is issued only after a previous request has been fully answered, this results in an idle time (see Figure 3.11(a)). To eliminate this period one can use pipelining

as suggested by Rodriguez and Biersack [RB02], by requesting further data before a server has finished the current transfer. For static equal this can only be done to a certain degree. For example, one waits for a subset of blocks of a requested batch and then sends another batch request. To enable a continuous ordered data stream some requests have to be delayed until all servers have provided their subset of blocks for the current round. Of course, this introduces the need to buffer data inside the group communication framework at the receiving node. However, the required memory footprint is rather moderate compared to an approach that omits rounds like earlier discussed. In fact the current prototype implementation starts a new round if less than batchsize packets are missing to complete a round. Assuming there are no out of order transmissions maximal two rounds can be active at a time. This leads to a worst-case memory demand of less then $2p|B||S|$ byte.

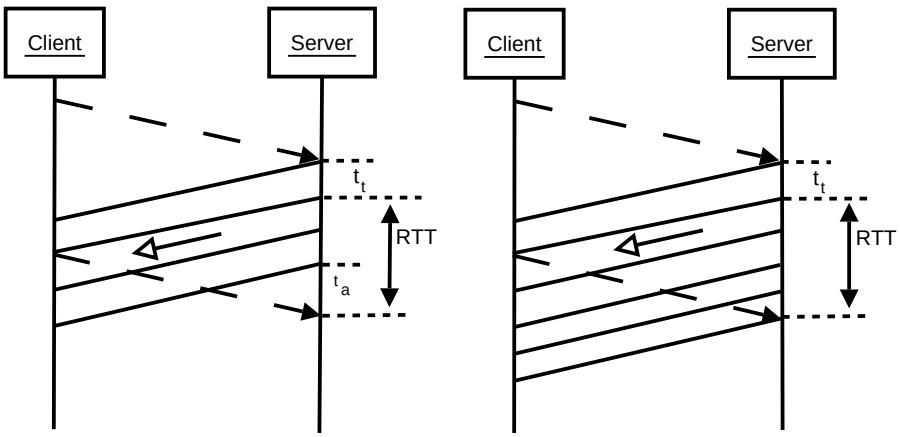
### Dynamic

Whereas the static equal algorithm assigns every server an equal part of the work, dynamic as proposed by Gkantsidis et al. [GAZ03] adapts the request strategy at runtime, taking network and server conditions into account. The presented adapted algorithm can be compared to dynamic TCP proposed by Rodriguez, Kirpal and Biersack [RKB00] but adds support for runtime adaptation of pipelining. Finally, the original dynamic algorithm is adapted to the needs of parallel state transfer by introducing batching.

The basic idea is to request a new block from a server each time the previously assigned block has been completely received. This ensures that servers which are less loaded and have a better connection (e.g. higher bandwidth and smaller RTT) transmit more data. This way the overall transfer time no longer depends on the slowest server like it is the case for static equal. To reduce the signalling overhead but gain a continuous data stream requests are combined to batches like done by static equal.

Instead of setting a fixed batchsize and a number of transmitted blocks as a mark for submitting the next batch request as done by static equal, dynamic adapts the batchsize $p$ depending on the server and requests a new batch immediately after the first block has been successfully received as shown in Figure 3.12. The key idea is to request not too much data as this might lead to bad performance if a server or connection for some reasons slows down. On the other hand requesting only one block at a time or requesting only a small batchsize leads to idle times at the server. The optimum would be to compute the batchsize in a way that a new batch-request reaches the server when the last block of the previous batch has been fully transmitted. As this is not possible due to the unpredictable behaviour of the network and the server load an overestimation is suitable. Rodriguez and Biersack [RB02] suggest to estimate an upper bound of the RTT and using this as a mark for submitting the next request. However this is a rough rule of thumb that can be replaced by an adaptive approach like described in the following.

Starting with Figure 3.12(a) the batchsize $p$ can be adapted to avoid idle times like illustrated by $t_a$ and in the optimal case match or at least underestimate the point when the server runs out of work (see Figure 3.12(b)). $t_a$ can be computed by $t_a = RTT - t_t(p-1) = RTT - \frac{b}{C}(p-1)$ with $b$ denoting to blocksize, $C$ to bandwidth and $t_t$ to the emission time of a block. In the optimal case, we assume $t_a = 0$, so we can compute $p = RTT\frac{C}{b} + 1$. Thus $p$




(a) Sub-optimal batchsize (p=4) causing idle time

(b) Optimal batchsize (p=5)

Fig. 3.12: Dynamic request scheme with adaptive batchsize

depends on runtime conditions. The bandwidth can be determined by measuring the time $t_t$ and compute $C = \frac{b}{t_t}$. The RTT can be measured in a straightforward way. As both values depend on runtime measurements that might temporarily fluctuate the exponential moving average is used to eliminate outliers and to include previous values but give earlier ones more impact. Under usual conditions, the computed batch length will be very short (e.g., 1 or 2) so the benefit of batching vanishes. To compensate this fact we introduce a configurable minimal batchsize (e.g., 3). However there are situations when the bandwidth and RTT is high (e.g., RTT 12 ms, bandwidth 76 MBit/s, blocksize 16 kb, batch size ~7) causing a larger batchsize.

As in the case of the static equal algorithm the group-communication framework has to buffer data as some nodes might provide their data faster than other ones. In the extreme there could be one state-providing replica that is much slower than all others, leading to a lot of buffered state. However, this case can be addressed by delaying the assignment of new batch requests until the left behind node keeps up. Alternatively, one could assign the missing batch to another node.

### 3.5.6 Synchronisation Approaches

Sometimes a complete state transfer is not necessary, instead only the changes between the current and some previous state snapshot should be exchanged. This might be the case if a node rejoins an object group or to reduce the request delays blocking state transfer approach is used as. The later is addressed in the following section.

Originally, tools like diff [HM76] or libraries like XDelta [MHS98] are used to compare two versions of the same or related files and output their differences as deltas. Usually, these deltas are much smaller than the file and the older version can be updated to the newer one by applying the deltas. Therefore, these comparison tools and their underlying algorithms are widely used to reduce data transfer in distributed system. A popular use-case is

updating software to patch critical security holes or simply to install software extensions. Although well known in the domain of distributed systems, synchronisation mechanisms have gained little attention in the context of state transfer of replicated services. For plain state transfer two approaches for delta-based data synchronisation were identified. Here, the core idea is not primarily to reduce data transfer, but to decrease request queuing due to a state transfer and to speed up the integration of a new replica. First the two approaches are outlined then it is investigated how they can be combined with parallel state transfer.

#### Snapshot-based Synchronisation

The first approach aims at reducing the request queuing time when using a blocking state transfer protocol. A blocking state-transfer can be useful as all replicas delay the request processing and restart the processing at the same time. This way there is no node that is left behind, as this might be the case for a joining replica when using a non-blocking approach.

For this approach it is assumed that during each state transfer the state-providing nodes write a snapshot to disk. If a new node joins the object group the state transfer procedure is not immediately started instead the new node first requests the snapshot from the previous state transfer. During the transmission of the snapshot, no messages have to be queued or delayed. Then the actual state transfer procedure takes place by queuing the requests, waiting for quiescence of the replicas and acquiring the current state from the application. Instead of transferring the whole state, one simply generates the deltas between the snapshot and the current state, and transfers just those. This approach is further on called *snapshot-based synchronisation*. It it assumed that the delta generation and the transfer of the deltas demand less time than transferring the whole snapshot. If for some reasons the application state has completely changed since the last state transfer this might be no longer the case. In this case the application could indicate that the state has changed since the last state transfer so the group-communication system can switch back to the standard state transfer protocol. This should happen when the state snapshot is requested from the application.

Snapshot-based synchronisation offers no benefits if using a non-blocking state transfer protocol as neither the overall transfer time is decreased or the amount of transferred data. In fact, the amount of transferred data is usually higher as a previous state and the deltas to the current state have to be transfer.

#### Remote Protocol Synchronisation

The second approach targets the faster integration of former members independent whether a blocking or non-blocking protocol is used. This is achieved by utilising the remote synchronisation protocol (rsync) [TM96]. Whereas diff and XDelta require both files locally, rsync provides a three step synchronisation remote protocol. First the node that has to be updated splits the last data into small parts and generates two kinds of checksums a weak and strong one. These checksums are submitted to the node that hosts the up-to-date data that in turn uses them to generate the deltas between local and the remote data. As the

weak checksum can be generated very fast it is first used to detect matching blocks. In case of match this is verified using the strong checksum. All blocks that did not match, along with some instructions how to apply them to update the data, are returned to the requesting node in step three that applies the changes.

This three-step synchronisation protocol has previously not been used to improve the state transfer for actively replicated objects but can be applied without major modifications. If a replica leaves the object group its state is written to disc and the checksums are generated. On a rejoin the basic state transfer protocol as outlined in Section 3.5.1 can be executed with one modification. The get_state message that is used to request the state includes the checksums of the old snapshot. The receiving replica evaluates the changes and submits them instead of the whole state to joining replica that in turn updates its old snapshot to the current state of the service. This approach is further on named *remote protocol synchronisation*. Similar to the snapshot-based synchronisation approach the snapshot of the rejoining node could be rather outdated reducing the benefits of this approach. However, if a replica recovers after a short period (e.g., a reboot of the node) this should usually not be the case.

#### Combining Remote Protocol Synchronisation with Parallel State Transfer

Both synchronisation approaches can be used in conjunction with parallel state transfer. As only the remote protocol synchronisation approach has benefits in the context of replicated decentralised adaptive services as a non-blocking state transfer is supported this approach is further on investigated. A basic combination of the remote protocol synchronisation and parallel state transfer would simply let the rejoining node send the checksums to all replicas. Each replica generates the deltas that are downloaded in parallel using one of the two proposed parallel transfer algorithms.

### 3.5.7 Writing Partial States

At least for a non-blocking state transfer, the state, if exceeding a certain size, has to be temporarily stored on disk during the state transfer; otherwise it might top the available memory. As the proposed support for state transfer of decentralised adaptive services targets the transfer of arbitrarily large states, it is expected that bigger parts of the state are usually files stored on disks. This leads to the point that during the state acquisition from the replica, the replica and the state transfer protocol concurrently read and write to shared disks. The replica reads the disk-dependent state and provides it to the group communication framework via a stream as outlined in Section 3.5.2. At the same time the state transfer protocol buffers the provided state on disk before its send to joining replica. Usually both operations will access the same disk. The question is, if and how this can be prevented or at least diminished. The easiest solution would be to enforce the usage of different storage devices. However this is not always possible.

Another simple approach alleviating the concurrent access is to assign every replica only a part of the state that it has to store. Of course this has to be coordinated with the parallel state transfer method. In case of the *static equal* approach this is very simple as every

replica has to provide a certain part of the state depending on the number of fully-functional replicas. Assuming, a blocksize $b$ and $n$ as the number of state-providing replicas. Every replica has a unique $ID$, defining a unique order over all nodes. Every replica knows the number of state-providing nodes and their IDs. On a state request each replica knows the first block it has to store by its position in the order and can determine all further blocks that have to be stored via $((n-1) * round)$.

Using the same approach when the state is transferred via dynamic is slightly more difficult. In this case there is no fixed rule which blocks are acquired from a replica instead this is determined at runtime. So writing only the $n$-th fraction of the state if the replica has $n$ full-functional replicas is simply unpractical and degrades the algorithm to static equal. However a more relaxed rule can be applied, e.g., by letting every replica write the half state. This assumes that there is no node that is faster (meaning having a smaller RTT and better bandwidth) than all other $n-1$ nodes together. If all participating nodes possess fairly equal network conditions this can be tightened by letting every node write only a third of the state and so one. Of course, the protocol layer at the joining node should know this rule to request only data that is available at a target replica.

As already described, partial state writing can easily be combined with parallel state transfer. The same applies for remote protocol synchronisation approach. A node preparing for a shutdown has to save the whole state otherwise a remote synchronisation approach does not work. During a rejoin the deltas first have to be determined for the whole state by each node, but then partial state writing is possible.

Concluding partial state writing offers an opportunity to alleviate concurrent disk access introduced by a non-blocking state transfer. Albeit simple such an approach has not been considered so far by related approaches. Further on it can be used in conjunction with parallel state transfer and it can be combined with the remote protocol synchronisation approach.

### 3.5.8 Implementation Overview

The proposed algorithms and mechanisms have been implemented as a protocol layer of the JGroups [Ban98a] group communication framework. JGroups possesses a modular protocol stack that is configured at start-up time. An application accesses the framework via a *channel* that provides a socket-like communication endpoint. A channel provides a locally unique address and enables an application to receive and send unicast messages to single members or multicast messages to all members connected to the channel. As described in Section 3.3.1 replicated decentralised adaptive services do not directly access the channel but via the generated fragment framework classes. Figure 3.13 shows a reduced protocol stack including the implemented *distributed state transfer protocol (DSTP)* layer. Each protocol can be configured via properties during the stack initialisation. There are essentially two kinds of transmission units named *events* and *messages*. Events represent a signalling mechanism for corresponding protocol layers. Messages are application-dependent transmission units.

The message sequence diagram in Figure 3.14 outlines the basic signalling protocol of the non-blocking variant of the distributed state-transfer protocol. Initially, an application

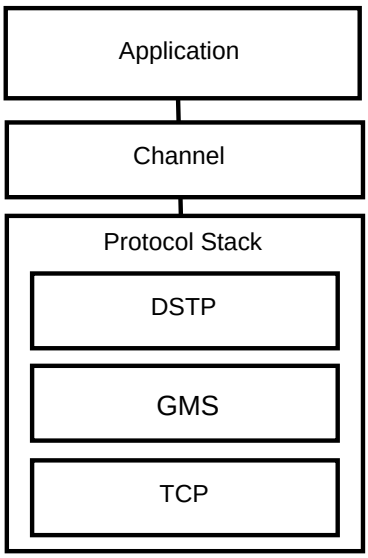



Fig. 3.13: Architecture of JGroups

requests its current state via GET_DSTATE. The DSTP-layer returns a Java InputStream to the application to immediately deserialise the received state. Next, the DSTP-layer sends a NEED_CURRENT_STATE event to all members including the local instance. This causes all members to enqueue all further messages. Furthermore, all previously received requests are fully processed according to their order. Then, assuming a non-blocking state transfer is configured, a GET_APPLSTATE message is forwarded to the application. This message includes a Java OutputStream, which the object uses to serialise the state. All members of the group that reply by sending an event named STATE_VIEW. If there is already an ongoing state transfer this event and all other actions are suppressed. The joining node will be informed by STATE_TRANFER_DONE that a previously initiated state transfer is finished and can restart its state request by resending NEED_CURRENT_STATE. If there is no active state transfer the requesting node will receive the STATE_VIEW events of all group members. Collecting these events provides the information about all fully-functional nodes to compute the request strategy. This is necessary as for certain state transfer configurations (e.g., a non-blocking parallel state transfer with partial state writing activated) each node provides only particular parts of the state. This has to be taken into account for the transfer. After reception of the STATE_VIEW events the joining node can request the state according to the request strategy by sending dedicated DATA_REQUEST messages that are answered by DATA_RESPONSE. If the requesting node has received the whole state a STATE_TRANFER_DONE message finishes the state transfer.

If a node rejoins a channel and possesses a former state the remote synchronisation protocol approach can be used. This introduces one further step in the previous protocol. Before actually requesting parts of the state the joining node sends a SYNC_SIGS message to all nodes. This message includes the checksums that are needed to compute the deltas. These deltas are then requested as any other state via DATA_REQUEST. On the receiving side the deltas are applied to the old state snapshot and the result is handed over to the replica.

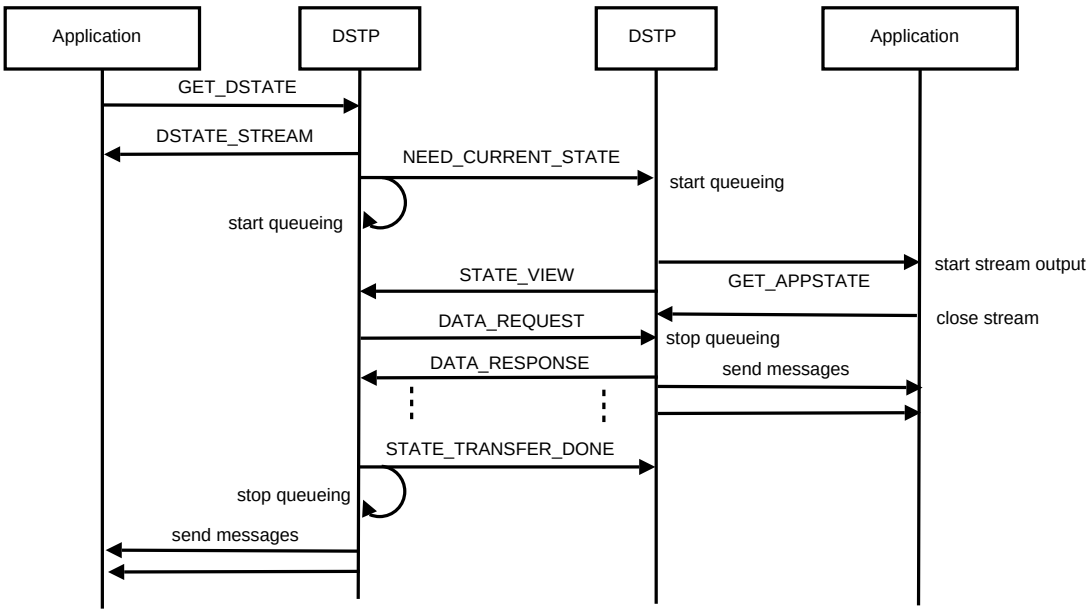



Fig. 3.14: Message Exchange of the Non-Blocking Distributed State Transfer Protocol

### 3.5.9 Experimental Evaluation

This section evaluates the parallel state-transfer strategies discussed in the previous section in a homogenous LAN environment and a heterogeneous Wide Area Network (WAN) setting. Finally, the impact of a non-blocking state transfer on service availability is investigated.

#### State Transfer in an Homogenous LAN Environment

Group communication and active replication of objects often takes place in an homogenous cluster environment. The same is in general expected for replicated decentralised adaptive services. Thus, the following measurements have been made on a set of PCs with a AMD Athlon 2.0 GHz CPU and 1 GB RAM, using Linux kernel 2.6.17, SUN Java SDK 1.5.0_09, and connected by a 100 MBit/s switched Ethernet network.

The first experiment measured the elapsed time for state transfers of state sizes between 0 and 200 MB for replication groups using the static equal and the dynamic protocol. As the impact of parallel state transfer depends on the number of state providers the group size has been varied from one to four state-providing nodes. In all experiments, a fixed batchsize of 10 and a block size of 16 kB has been used. In order to compare the prototype implementations with existing state-transfer protocols, I did the same measurements with two state-transfer protocols provided by the JGroups group communication framework. The first variant, implemented by JGroups version 2.3, supports a non-blocking state transfer that requires the application to provide the state as a byte array that is transferred to the joining node. The second variant has recently been released as preview of the future JGroups version 2.5. It offers an API similar to our prototype provided by this thesis and supports a blocking streaming state transfer.

Figure 3.15 show the results of the measurement. The old state transfer protocol of JGroups 2.3 is not suitable for transferring states larger than 50 MB. The JGroups 2.5 state

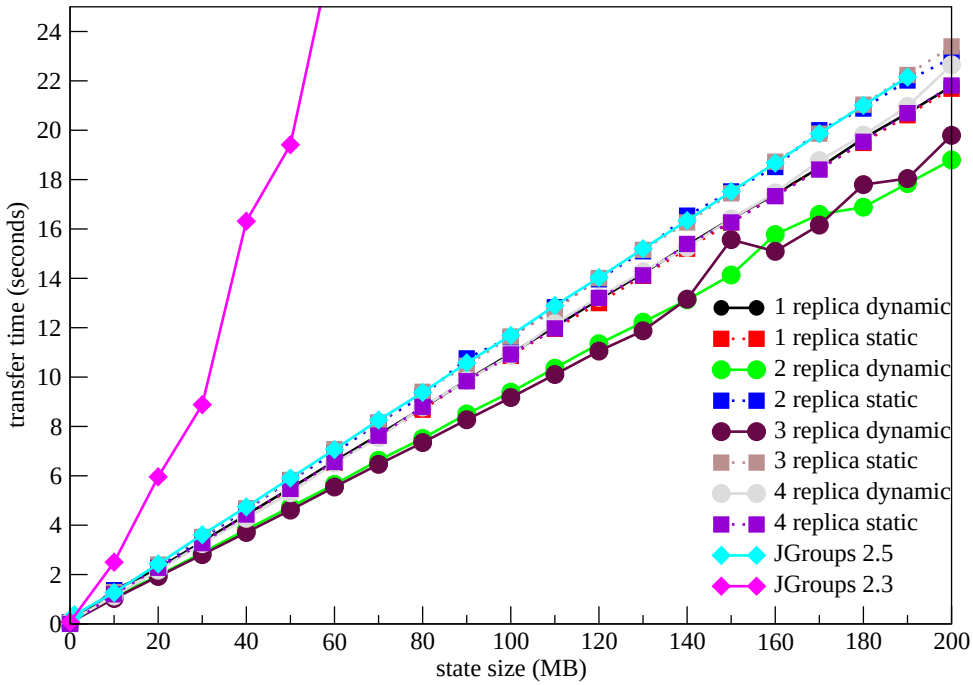



Fig. 3.15: State transfer in a LAN environment

transfer protocol implementation scales better, but is not more efficient than one of the presented parallel state-transfer variants. The static equal parallel transfer produces very similar results for 2 and 3 state providers, and is slightly faster with 1 and 4 replicas. This circumstance is emphasised by Figure 3.16 that shows the transfer time for 100 MB. The reason for the slightly longer transfer time for 2 and 3 replicas is founded in the restrictions of the equal algorithm not to start a new round before a certain amount of data from previous rounds has been received. Thus, if there are slightly faster replicas these are delayed by the slowest one. This effect is noticeable for 2 and 3 nodes and vanishes if more replicas provide the state or if more rounds can be executed in parallel. The last fact has been validated by further experiments.

Dynamic state transfer from 1 and 4 replicas produces similar results. The fastest state transfer is achieved with the dynamic algorithm and state provided by 2 and 3 replicas. The reason for the slightly longer transfer times for 4 state-providing replicas in contrary to 2 and 3 seems to be a network saturation at the link to the target and the overhead for sending requests to an increasing number of state providers.

All streaming state transfer variants produced good results that are close to each other. The dynamic variant performed slightly better than the static one, but the difference is very small. This matches the expectations, as a static equal distribution of state-transfer tasks on all nodes should be well suited for the given homogeneous environment.

In practice, a LAN or cluster environment usually is not dedicated to a single application. Thus, in a second experiment the impact of CPU load at one of the state-providing replicas has been evaluated. A simple load generator has been implemented to produce predictable and reproducible load. During the whole experiment, the system CPU load of the burdened

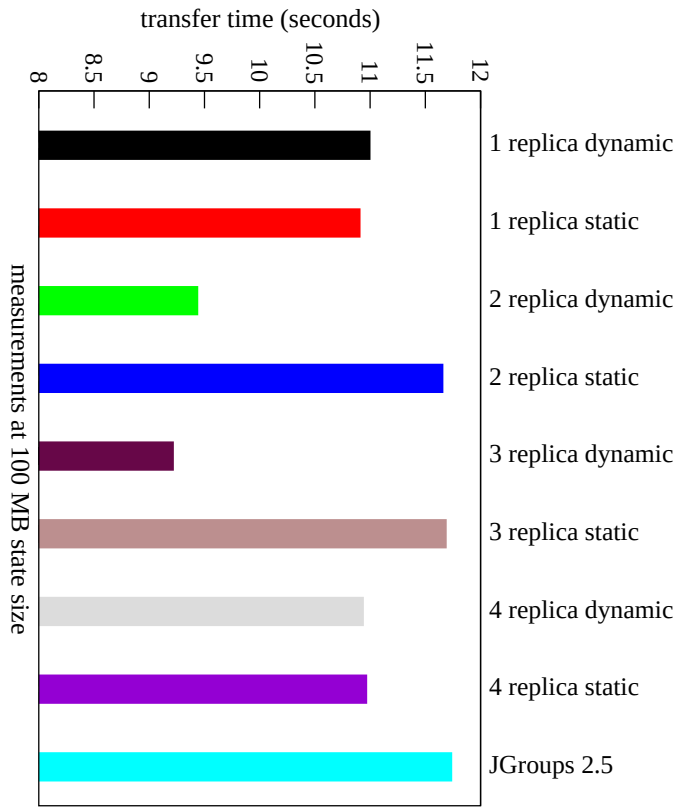



Fig. 3.16: State transfer in a LAN environment measurements at 100 MB state size

node oscillated between system load level 2 and 3. Thereby three replicas provided the state and a fourth node joined the group. This time the state size has been raised from 0 to 200 MB in steps of 10 MB.

Figure 3.17 shows the strong impact on the state streaming JGroups implementation. The state transfer time roughly doubles in comparison to an unloaded system. Both the dynamic and the static implementation perform better, as the state transfer is split among all state-providing replicas. The dynamic variant in general outperforms JGroups 2.5 and static equal state transfer.

In summary, the two experiments have shown that the introduction of parallel download techniques accelerates the transfer of large application states. While the benefit is only small in an idle environment, a significant speed-up is obtained in an environment with high CPU load. In both cases, the proposed dynamic state transfer algorithm outperforms the streaming state transfer offered by JGroups 2.5 and the parallel static equal algorithm.

### State Transfer in a Heterogeneous WAN Environment

For evaluating the proposed techniques in a heterogeneous WAN environment, a set of four different nodes has been chosen; each node possessing slightly different hardware and running a different operating system. Two nodes are located in the same sub-network at the FAU Erlangen-Nuernberg, a third node *faui00a* is located in a different sub-network also at the campus of the FAU. Finally the fourth node *schirk* is located more distant at Ulm

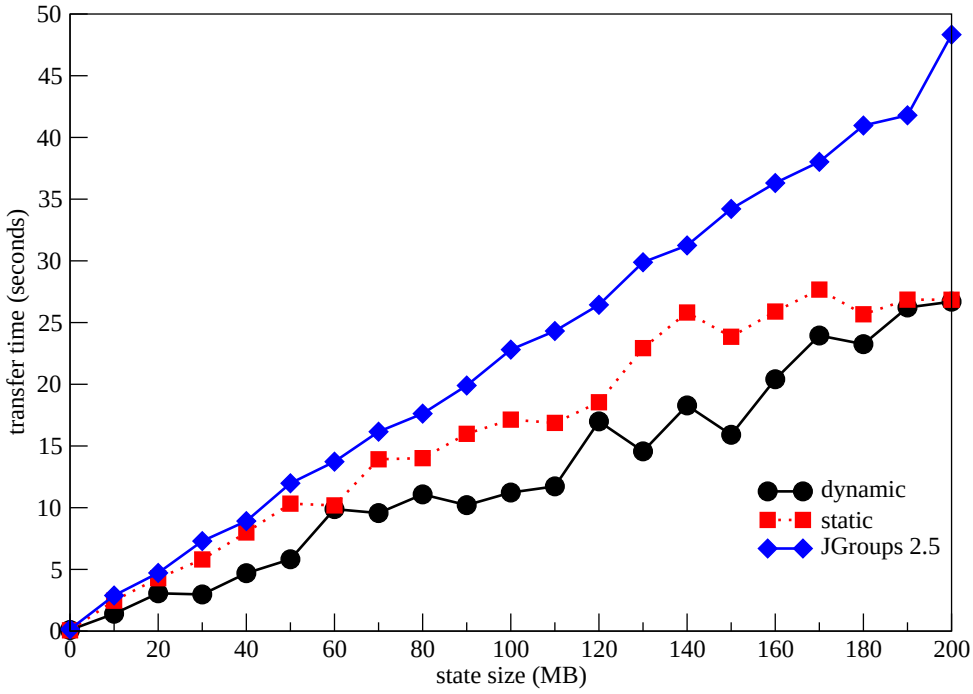



Fig. 3.17: State transfer in a LAN environment with load injection

University.

In the experiment a group of three replicas has been set up and a fourth node joined the group. Two scenarios have been chosen: One time faui00a at FAU joined the group and another time schirk, the node located at Ulm University, entered the group.

As Figure 3.18 outlines the state transfer protocol of JGroups 2.3 had scalability problems. The reason seems to be the memory consumption of the protocol this was especially the case when the distant node joined the group. The JGroups 2.5 protocol produced better results than the implementation of JGroups 2.3 and, as expected, requires more time for state transfer if the node at Ulm University joins the group. The transfer values of static equal are very close together, independent of the location of the joining node. Static equal is in general better than JGroups 2.5 if the distant node joins the group, but slower if the joining node is located at FAU. This is to be expected, as static equal waits for the slowest node to start another round. The dynamic parallel state transfer performs best regardless of the location of the joining node.

### Non-blocking State Transfer

This experiment does not target the reduction of the state transfer time, but instead evaluates the reduction of service unavailability caused by a state transfer.

A replication group of two nodes has been formed. One node sending probe messages every 100 ms to all group members. Every node that receives a probe message immediately replies to the probe and the sender records the round-trip time. Again, a third node repeatedly joined the group and the state size has been raised step-wise from 0 to 200 MB. During this process the joining replica recorded the time to acquire the state and the state-

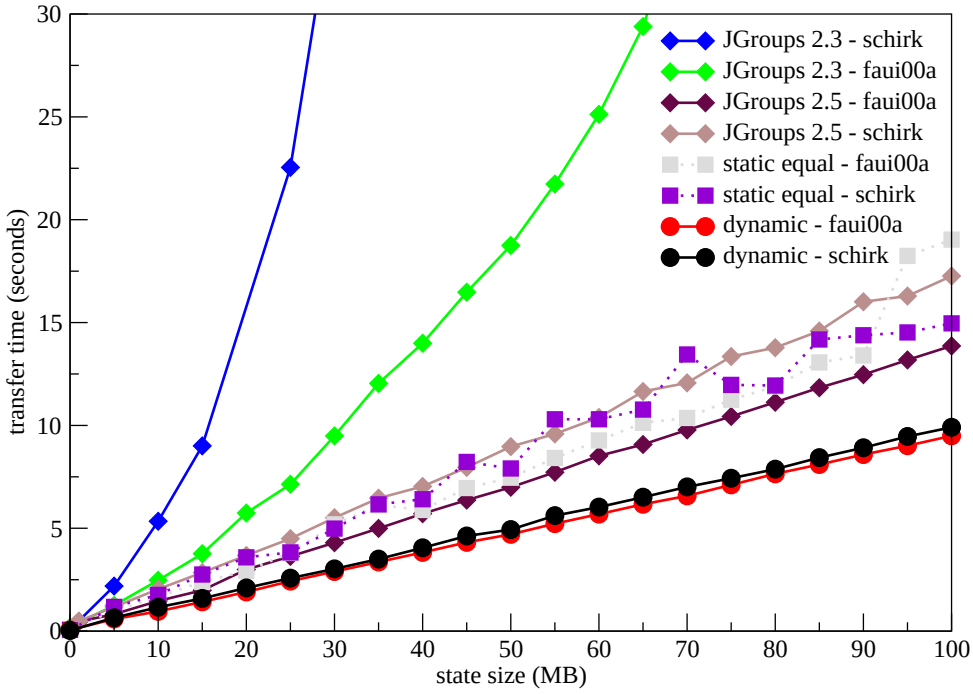



Fig. 3.18: State transfer in a WAN environment

providing nodes logged the time to hand over the state to the group-communication framework. As Figure 3.19 details by the strong red and black lines, far less time is required to provide the state to the framework than to transfer the whole state. This is achieved as the state is temporarily saved on disk. Directly after the state provision, the application is able to respond to requests, as the second set of curves shows. Here the fine black line outlines the message round-trip time from the sending node to a state-providing node. The fine red line depicts the round-trip time between the sending node and the state requesting one. As can be seen by the curves the request round-trip time varies between less than one second and values that exceed the state transfer time. Furthermore the maximal request round-trip time of the state-providing node is always lower than the one of the state requesting node.

### 3.5.10 Summary

The presented concepts provide a significant improvement for state transfer in actively replicated objects. Due to the stream-based state exchange interface, object states of arbitrary size can be transferred in contrary to most of all existing group-communication frameworks and frameworks for fault-tolerant service replication. The integration time for new replicas has been speeded up by introducing parallel state transfer algorithms. These algorithms are extended variants of algorithms for parallel data transfer adapted to the requirements of state transfer in actively replicated objects. So far there is no other system known to the author employing similar techniques. Starting from a non-blocking state transfer protocol the request queuing time in which an object renders unresponsive was additionally decreased by partial state writing. Finally, it was discussed how to reintegrate former group

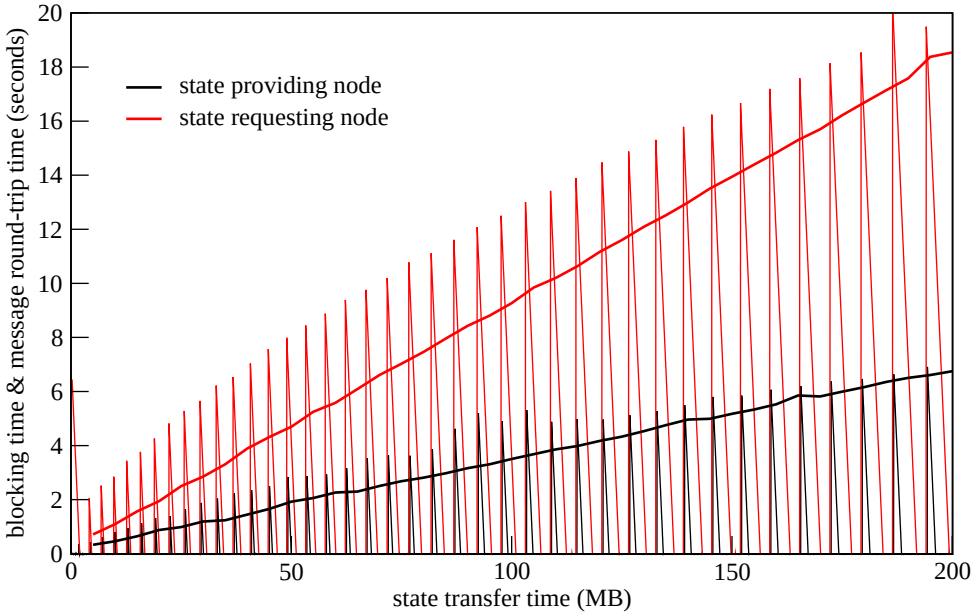



Fig. 3.19: The impact of a non-blocking state transfer on blocking time and message delay

members of an object replication group. An approach named remote-synchronisation protocol was proposed that should reduce integration costs in terms of data transmission and time. Summarising, each of the three presented extended mechanisms in conjunction with the stream-based state transfer interface builds a new contribution.

Even though the current prototype implementation supports parallel downloads for blocking and non-blocking state transfer and the integration of snapshot-based and remote protocol synchronisation has been partially implemented there remain some open issues. For non-blocking state transfer it might be possible that the state-providing replicas are very far ahead before the joining node. This might lead to high memory consumption, as many messages have to be buffered until the joining replica keeps up. To prevent such a problem one can use a flow-control protocol that is already provided by the JGroups framework. Such a protocol delays the submission of messages if one or more nodes cannot keep up with the faster ones.

Second and more important is the issue of node failures during the state transfer. In the ideal case the state transfer should simply progress as all state-providing nodes initially possess the whole state and crashed nodes can be tolerated. If using partial writing this is no longer the case and in a worst case scenario the state transfer has to be restarted from scratch including the acquiring of state from the application and the deletion of a partially loaded state. The latter demands for explicit interaction with the state reading service instance.

Despite these two open issues the provided prototype offers good support not only to replace crashed replicas by new ones but also to migrate replicas due to load or resource management reasons. This last issue is especially important for replicated decentralised adaptive services as it enables the adaptation of a service distribution according to resource demands imposed by the service itself and the hosting environment.

## 3.6 Deterministic Multithreading Support

Active replication that builds the core technique used by replicated decentralised adaptive services requires a deterministic execution of all state-modifying actions. If multiple threads are allowed to access a replicated object in parallel, the order in which threads access shared data may vary between replicas; this can lead to an inconsistent object state.

Usually, parallel unordered access is inhibited by letting only a single thread enter the object, like done by OGS [FGS98] and GroupPac [FMLF97]. However, allowing only a single thread to enter and modify an object has several drawbacks. For example these systems are not able to handle *nested remote invocations*, where one replica group requests the service of another replica group that in turn requests the service of the first replica group. If such a request would be handled without taking additional actions the consequence is a deadlock. This problem can be solved by tagging nested invocations with context information that identifies the originating logical thread. To avoid non-determinism the first thread representing the original request remains blocked during the execution of the nested invocation and is resumed after the additional second thread has finished. This was first provided by the Eternal system [NMMS99].

Although this basic approach solves the problems caused by nested invocations it demands a restricted programming model missing the support for condition variables like used in most multithreaded client/server applications. Thus, a non-replicated service has to be manually adapted to the requirements induced by the replication infrastructures following this approach. Furthermore such a restricted threading model causes performance degradation due to decreased concurrency.

Accounting these facts the framework for replicated decentralised adaptive services provides support for deterministic multithreading based on the ADETS-SAT [DHRK06] algorithm that is similar to the one proposed by Zhao et al. [ZMMS05] but provides improved support for condition variables and for the native synchronisation mechanisms of Java. Thus traditional non-replicated Java-based client/server implementations can easily be converted to replicated decentralised adaptive services. This follows the objective to provide an implementation model for replicated decentralised adaptive services that is similar to a client/server implementation. The support for deterministic multithreading and the briefly summarised support of the ADETS-SAT is not a core contribution of this thesis but is presented to build the basis for the performance evaluation presented in Section 3.9.

### 3.6.1 Deterministic Thread Scheduling

As described in Section 3.3.1 the framework for replicated decentralised adaptive services provides a pluggable scheduling component. In context of the current prototype implementation a plug-in of the ADETS-SAT algorithm was implemented. It offers a non-preemptive, deterministic mechanism for thread scheduling. A new thread is only created and an existing thread is only resumed, if all other existing threads have reached a safe state. This is the case if they have terminated or have blocked waiting for a mutex, for a condition variable notification, for a timeout, or for a nested invocation reply. All decisions are fully deterministic, and consequently remain consistent among all replicas. Lock requests, lock releases,

and condition variable accesses need to be passed to the thread-scheduling algorithm.

Whereas the support for mutex and condition variables can be implemented in a straightforward way the handling of time-bounded wait operation requires additional actions and is therefore briefly summarised in the following. First an internal timer is scheduled. As soon as this timer expires, the replicas send special time-out messages to all group members. The arrival of the first message cancels the local timer if it is still active, and thus may suppress the emission of further time-out messages. If there are multiple timeout messages only the first one is processed. The waiting thread is not resumed immediately, as this could result in nondeterministic behaviour. Instead, time-out messages are processed by schedule only if all threads are suspended or terminated. In this situation, the thread may either still be waiting in all replicas, resulting in a deterministic resumption by the timeout, or it may have been resumed by a notification operation.

### 3.6.2 Code Transformation Support

To avoid the modification of the execution environment (i.e., the JVM), but to allow implementing synchronisation in the replicated object with native Java mechanisms (`synchronized` statements, etc) a tool is provided. It transforms all synchronisation statements into appropriate calls to the deterministic scheduling component. This approach requires that the synchronisation of a replicated decentralised adaptive service is fully encapsulated within the replica fragment. Meaning it is assumed that lock object instances used by the replica implementation are not used for synchronisation in code outside the replica fragment, but within the same JVM. Developer code within the access fragment has no direct access to the object state and thus does not require synchronisation.

### 3.6.3 Summary

The outlined scheduling mechanisms together with the support for code transformation enable an easy porting of non-replicated service implementations to become replicated decentralised adaptive services without the need to manually change the implementation or the execution model (e.g., in the case of nested invocations).

The proposed support for deterministic multithreading is essential to replicated decentralised adaptive services as it enables a straightforward implementation of services that are concurrently accessed by multiple users that may frequently modify the service state. Allowing only a single thread to enter and modify the service state would impose an unnecessary burden to the service developers and force them to provide custom solutions that might be error-prone or inefficient. The latter has been evaluated by the implementation of a counter that is concurrently modified by a set of clients (see Section 3.9). Despite the used ADETS-SAT algorithm provides already sufficient support for deterministic multithreading there exist several algorithms that enable extended deterministic multithreading [BWKI02, BKI03, RHD+06] that have been evaluated in the context of replicated decentralised adaptive services and the Aspectix project but this is not detailed any further. A comprehensive description can be found in the Ph.D. thesis of Reiser [Rei06].

Albeit the benefits of supporting condition variables, their careless use can have serious

consequences for decentralised adaptive services. If a client issues a request that blocks inside the service, as it waits for a condition to become true, this inhibits a state transfer as all running requests have to be finished before a consistent snapshot can be taken. Thus, such a scenario should be generally avoided. However even without having condition variables available a request may take arbitrarily long (see Section 3.5.1). Both problems can be solved by interrupting such long-running operations at certain safe points that have to be defined by the service. In case of a blocked request waiting for a condition variable this would mean to wake up the waiting thread in a consistent order on all replicas. Once activated the thread determines that the condition is not true but that the request has to be aborted because of a state transfer. Thus, the object is put back in a consistent state by the thread and the request is finished. This approach is feasible but not transparent to the developer.

## 3.7 Semantic Annotation Support for Customised Implementation

According to the objectives, replicated decentralised adaptive services can be developed similarly to common CORBA client/server applications, starting with the specification of an IDL interface. This interface is passed to an IDL compiler that generates client- and server-side marshalling and service-specific fragment code. Finally, the developer has to provide the object functionality by implementing the replica fragment methods. In contrast to the traditional development model it is furthermore possible to annotate the service interface for a custom code generation process that enables among other things to provide service-specific method implementations at client-side. This offers great flexibility to implement services with non-standard demands as demonstrated at the end of this chapter by providing a decentralised version control system.

With IDLflex [RSH01], the Aspectix project provides an IDL compiler that is capable to generate custom client- and server-side code based on a configurable XML-based mapping specification. There are already two mapping specifications for the Java programming language, one for standard CORBA and another for supporting fragmented objects. The mapping for fragmented objects provides, as detailed in Section 2, basic support for the general object model but misses any marshalling and communication support. This is left open to the developer and offers great flexibility, as any internal communication structure is possible. If a certain internal object architecture is frequently used this is leading to the recurrent implementation of transport and marshalling code which is time consuming and might be error-prone. Therefore one would usually only use the plain fragmented-object support for special cases, where the provision of tool support is not profitable. As fault-tolerant replication is a frequent requirement and as the internal architecture of replicated decentralised adaptive services is fairly complex the standard mapping for fragmented objects was extended to generate client- and server-side code that provides marshalling and glue code to support the architecture described in Section 3.3.1.

On top of this basic support for replicated decentralised adaptive services, the IDLflex compiler was extended to support semantic annotations at the operation level. These are expressed as `#pragma annotate` statements. Within the extended mapping, these anno-

tations are evaluated and used to control the code-generation process. This allows the utilisation of semantic knowledge, provided by the developer, to improve and customise the replication mechanisms in three ways. First of all, it is possible to gain performance improvements as shown in Section 3.9. Second and even more important, it allows the flexible introduction of code at client side which preserves the benefits of the underlying fragmented object model while still providing comfortable implementation support (e.g., no need to write marshalling code). Third, it enables to introduce access control.

The semantic annotation of interface definition languages has previously been investigated in various projects. In the following, a short overview about the most prominent approaches is provided. Then, the introduced annotations are described in detail and finally the impact on the code generation process is explained.

### 3.7.1 State of the Art

Specifying the remote interface of a service in a language other than the implementation language of the client/server application for generate marshalling code has various reasons. The most important one might be the fact that the implementation language does not provide enough information to automatically generate marshalling routines for any kind of parameter. Interface definition languages solve this problem by a more strict type set, e.g., by avoiding pointers or allowing only size-bounded arrays. Another important aspect is platform-independence. Interface definition languages are usually only used to specify an interface and miss any operational constructs. This way they avoid any form of platform dependencies. Of course parameters can have certain requirements, e.g., the value space of an integer. However this does not tie the language to a certain platform but has to be handled by the transformation process from the interface definition to the real marshalling code that demands for a defined mapping. Thus an interface definition can be used to generate marshalling code for clients and servers that possess different system properties. Finally the usage of an interface definition language allows the integration of additional knowledge about the distributed application provided by the developer and therefore an optimised code generation. Simple examples in the context of CORBA IDL are the declaration of the marshalling directions of parameters with the keywords `in`, `out`, and `inout`. If a parameter is declared as `in` it only needs to be transferred to the server and can be omitted by the marshalling process for the response. However this is only a very limited form of representing developer knowledge at the level of the interface description to optimise and extend distributed application.

Krämer et al. [KHK96], and also Krämer and Jacobsen [JK98] have evaluated how to extend standard CORBA IDL by semantic properties to automatically synthesise code that instruments this information. Their main objective was to support security and safety conditions, such as synchronisation constraints or guards and an implementation-independent way to specify these constraints. This way the same tests and guards that can be used to evaluated different implementations for different platforms. As an extension to this work Jacobsen and Krämer [JK00] developed an XML-based framework to specify IDL extensions and generate custom and extended client/sever stub code. Their framework can be compared to the IDLflex compiler, used for supporting semantic annotations in the scope of this thesis.

Despite the generic support for IDL extension the authors identified demand for various kinds of extensions such as security, quality of service, caching and others.

The DADO [WJD03, WJD04] system offers advanced features to adapt client/server stub code by using concepts of aspect-oriented programming (AOP)[KLM+97] at the level of IDL descriptions to insert so called *adaplets*. The authors evaluated the insertion of crosscutting-concerns off-line but also online. Whereas this goes certainly beyond the annotation of IDL descriptions that allow only a static integration of developer knowledge it is in principle bound to customisation of the request handling process. For example requests can be cached and logged or request compression and encryption can be introduced. Thus, DADO enables similar support as interceptors.

### 3.7.2 Semantic Annotations

For replicated decentralised adaptive services three different types of semantic annotations have been identified as valuable. First, semantic annotations to speed up request handling. Second, extended support for client-side computing. This is essential to develop complex distributed applications and offer the opportunity to use a framework-based replication support. Third, support for access control is provided. Access control is mandatory in open distributed environments and the provided support offers a suitable solutions.

#### Semantic Annotations to Speed Up Request Processing

To alleviate the impact and overhead of active replication if possible two modifiers for performance improvement are provided:

- `readonly`: A method marked as `readonly` does not change the replica state. Instead of executing this method at all replicas, it is sufficient to invoke it on one available replica.

- `parallelisable(methodlist)`: A method marked as `parallelisable` with respect to a set of other methods can be executed in parallel with the specified list of other methods. This allows true multithreading and loosening the restrictions of a deterministic scheduler

Both annotations require special request handling by the proposed deterministic scheduling component. A thread may be created or resumed not only if all other threads have reached a safe state, but also if it is marked as `parallelisable` with all other threads that have not terminated. Special care needs to be taken for methods labelled as `readonly`, which are only executed in one replica. These methods are not allowed to use wait/notify operations on condition variables, as this could lead to an inconsistent scheduling of other modifying methods. Furthermore the framework for replicated decentralised adaptive services does not support the nested executions of read-only operations. Possible solutions to this problem have been proposed by Reiser [Rei06].

#### Semantic Annotations for Custom Client-Side Computing

Up to now a classic client/server code generation support for replicated objects is provided, comparable to other frameworks supporting the development of fault-tolerant services (see Section 3.2). To provide support beyond this traditional client/server partitioning, and to take advantage of the fragmented object model, three additional annotations are supported:

- `local`: The implementation of a method marked as `local` will be placed in the client-side fragment. This way, methods that need no access to the replica state can be executed locally at client side, while still being conceptionally part of the distributed object. Furthermore it is possible to execute other methods of the service, especially methods marked as private.
- `private`: Methods marked as `private` will not be included in the public interface of the service but can be called and executed from any client-side part of a local or intercepted method. The marshalling code is still generated, thus a private operation resembles a normal remote method that is not accessible to the outside. This offers more flexibility for object-internal client/server programming.
- `intercepted`: A method marked as intercepted will execute custom code at the client-side before and after invoking the remote method at the replica group. This mechanism can be used for local preprocessing, for caching, or for the accumulation of multiple client invocations into one remote invocation to the replica group. Along the lines of methods marked as local it is possible to access methods labelled as `private`.

#### Supporting Access Control based on Semantic Annotations

Restricting the access to a service is another important requirement. It is solved via an own annotation:

- `restricted`: Methods marked as `restricted` are access protected. Before executing such a method a client has to provide credentials to the local fragment.

### 3.7.3 Customised Code Generation and Request Handling

This section targets the effects and realisation of the proposed semantic annotations in terms of request handling and code generation. First the exclusively performance-relevant annotations are discussed, second the handling of the structural annotations are described and lastly the support for access control is outlined.

If at least one method is marked as `readonly`, the generated code for the gateway (see Figure 3.2) component of the replica-side fragment will examine all request invocations. All read-only requests will bypass the totally ordered group-commutation and are directly forwarded to the upper layer to locally execute them. Within the scheduling component, the thread handling is controlled by the deterministic thread-scheduling algorithm, which is described in Section 3.6. If a method is either marked as `readonly` or as `parallelisable`, the scheduling component of the replica fragment is informed about this property to appropriately handle them.

Specifying a method as local causes the implementation of the method to be placed in the access fragment instead of the replica fragment. This is achieved by generating an appropriate empty placeholder at the access fragment. Conceptually two kinds of `local` methods are possible: state modifying and static ones. A static method is typically used to validate client data against a fixed set of rules, to provide factory methods or any other kind of static service. In a state modifying case of a local method, the method calls other methods of the replicated decentralised adaptive service. This might be public methods but more often methods marked as `private`.

For methods annotated as `intercepted`, an abstract method is created in the access fragment and marshalling code on client- and server-side. The actual method implementation on client as well as on server side has to be provided by the developer. Methods marked as `private` do not require extended code generation despite the marshalling code but to omit their signature from the fragment interface, so they are not accessible via the interface of the service.

Finally, methods marked as `restricted` require the generation and inclusion of access control code. The access fragment offers a special method to hand over credentials that are checked, by the inserted code, before the execution of any access-restricted methods.

### 3.7.4 Summary

Especially the proposed semantic extensions targeting performance improvements and custom client-side programming offer extended support for efficient and complex replicated services that are currently not provided by any other systems. Felber et al. [FJRS01] suggested to extend IDL interfaces by semantic knowledge enabling an intelligent protocol selection for distributed component interaction and especially replicated objects. They also proposed the annotation of read-only and parallelisable operations, but missed to provide an evaluation. Brügge and Vilsmeier [BV03] recently evaluated the impact of caching and pre-fetching of call results in the context of a non-replicated CORBA application but did not evaluate the impact on replicated services. Finally, they provide only specialised support for these two features but also miss a generic support for client-side computing. Other earlier discussed related work [JK98, WJD03, WJD04, JK00] proposed some form of generic support for semantic annotation and integration of developer knowledge but does not consider client-side computing as proposed in the context of this thesis. In Section 3.9 the performance relevant annotations are evaluated and, even more important, Section 3.10 demonstrates how the annotations support can be used to implement a decentralised version controls system that has various non-standard demands.

## 3.8 Run-Time Infrastructure

To support the creation and management of replicated decentralised adaptive services a minimal run-time infrastructure was developed. Along the lines of other fault-tolerant middleware infrastructures it provides two core components: A *generic factory* that is able to create replica fragments, and a *factory finder* component to search for factories. Unlike

```
interface GenericFragmentFactory {
  APXObject create_initial_replica(
        in Key k,
        in Credentials cred,
        in Criteria the_criteria)
    raises (NoFactory, InvalidCriteria, CannotMeetCriteria);

  void create_replica(
        in APXObject fo,
        in Credentials cred,
        in Criteria the_criteria)
    raises (NoFactory, InvalidCriteria, CannotMeetCriteria);
};
```

Fig. 3.20: Interface of the generic fragment factory

other approaches the creation and management of a replicated decentralised adaptive service is not handled by an external entity as it is the case for FT-CORBA that specifies a replication manager. Instead a service represents a self-managing entity, thereby reducing the complexity of the infrastructure.

The creation and start of a new replicated decentralised adaptive service demands for several steps. First, a generic factory has to be searched by querying a factory finder. Such a factory finder basically represents a small database that hosts information about possible places of execution in the scope of a certain area [Obj02]. In a minimal infrastructure setting, the factory finder is realised as a common CORBA object, well-known on every node within a domain. This way, a node can register its local factories and replicated decentralised adaptive services can lookup factories from all other nodes. To provide a fault-tolerant and scalable solution an extended decentralised factory finder support was implemented in the context of the EDAS infrastructure as detailed in Chapter 5.

The generic factory offers basically two creation-methods (see Figure 3.20): one for setting up an initial replica fragment and another one for setting up further ones.

First, one or more factories are looked up using the factory finder. One of them is requested to instantiate the first replica using `create_initial_replica()` (see Figure 3.21 (a)). This results in the creation of an initial replica fragment and the activation of the service. Then, the service is returned to the requesting by handing over a local access fragment to client. In the following the service controls the creation of additional replicas.

A replicated decentralised adaptive service provides a management interface to adjust the number of replica fragments (see Figure 3.21 (b)). If the number of demanded replicas is increased, the existing service will be triggered to add the necessary number of additional replicas. First, one of the replica fragments queries the factory finder for additional factories. Then, the service reference is passed to one of the factories. At the factory side, the service is transparently bound by the Aspectix middleware, which instantiates an access fragment. Afterwards, the factory reconfigures the local fragment to be a replica fragment.

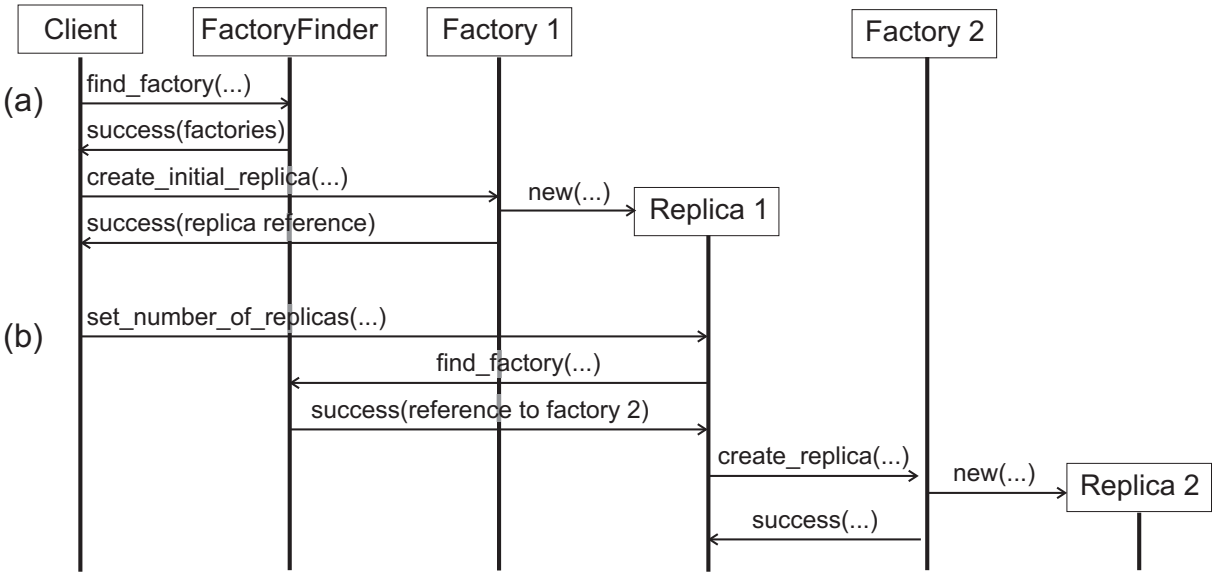



Fig. 3.21: Creation of first and additional replicas

This process is repeated until the desired replication level is reached or no additional factories can be found. A failed replica fragment is detected by the failure-detection mechanism of the group-communication framework. This triggers the set up of a new replica fragment.

## 3.9 Performance Evaluation

The performance evaluation is split into two parts: First the basic annotation constructs are evaluated then the benefits of the deterministic thread scheduler are shown.

All measurements were performed on a computer pool of 16 AMD Opteron 2.2 GHz servers running Linux 2.6.15, connected via a switched 100 Mbit/s Ethernet. The Aspecitx ORB [HKRS05] was used, which is internally based on the JacORB 2.2.1 [Bro97], together with JGroups 2.2.9.1 [Ban98a] as group communication framework. The JGroups stack was configured to use TCP connections and total ordering. Since the ORB and JGroups are implemented in Java all measurements were done with the Java server VM SDK-1.5 from Sun Mircosystems.

### 3.9.1 Evaluation of Efficiency Gains Using Semantic Annotations

In the first example, the differences in invocation time between a read-only, a local and a modifying method were measured. The local method is realised as an empty operation located at the access fragment. The read-only method invocations are not distributed via the group-communication framework, but sent directly to one of the replicas. Figure 3.22 depicts the average time per invocation, obtained from several runs with 5000 client invocations. A single client accesses a replicated decentralised adaptive service with the number of replicas increasing from one to five nodes. The invocation cost for modifying method is mainly caused by the request transmission via the totally ordered group communication. The read-only method performs much faster, than the modifying one, but the local method

is by far the fastest operation. This underlines the benefit from using semantic knowledge about object methods for building efficient actively replicated services.

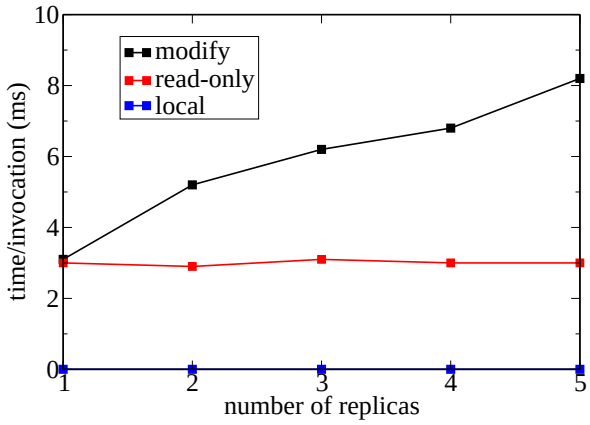



Fig. 3.22: Efficiency gains using semantic annotations

### 3.9.2 Evaluation of Deterministic Multithreading Support

Three different examples were implemented to evaluate the scheduling algorithm in comparison to a non-multithreaded approach. These measurements have already been published in [DHRK06]. The first measurement demonstrates the performance improvements that are obtained by the multithreading approach in case of nested invocations. Second, a simple producer/consumer example reflects the benefit that is obtained from supporting condition variables. The third example shows a time-restricted `wait()`-operation, which is usually very useful in combination with nested invocations as a weakly predictive response time might be required.

In the first scenario, two services $A$ and $B$ are created with each consisting of 3 fragment replicas. A varying number of clients call a method at service $A$, which in turn calls a method at service $B$. Internally, both requests and the reply from service $B$ to service $A$ are delivered via group communication. All clients are started simultaneously, and a total of 10,000 invocations is made at $A$ (and, in turn, at $B$). Figure 3.23 shows the average invocation time measured by the clients; to minimise JIT compilation effects, the first 200 invocations of each client are not included in the average. The solid lines (diamond and triangle symbols) refer to measurements in which the nested invocation returns immediately. Even in this situation, multithreading is increasingly better with a rising number of clients. In a second measurement (dashed lines with circles and squares), the method called at $B$ suspends for 2 ms before it returns. In this case, the benefit of a multithreaded approach (which allows accepting new requests at $A$ while the invocation to $B$ is still in progress) is enormous compared to a single-threaded execution.

The second example shows the possible speed-up obtained by using condition variables in a simple producer/consumer example. A replicated counter is iteratively increased by a `produce()` method and decreased by a `consume()` method. A consumer-call only succeeds if the counter value is positive. With the multithreading support, the example can be

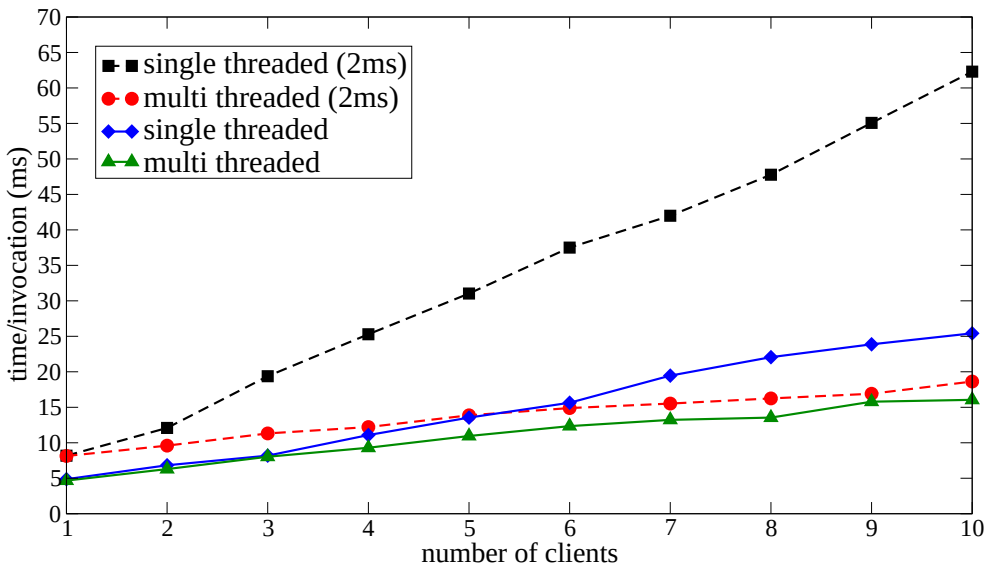
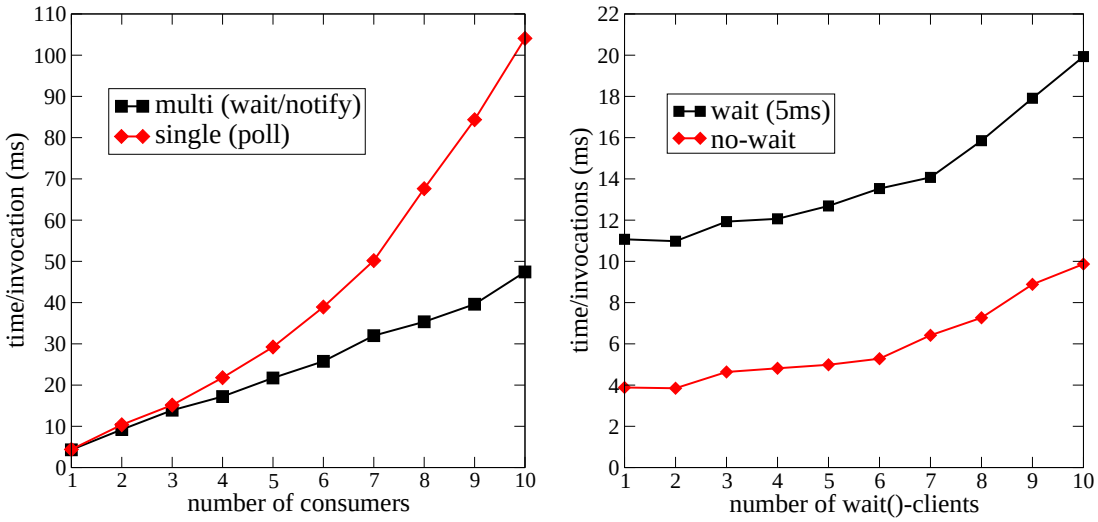



Fig. 3.23: Nested invocation example

implemented in a straightforward way using a condition variable, which is used to block a consumer until it is woken up by a producer call. Note that this implementation cannot be used in a single-threaded execution model. Therefore an alternative implementation was provided that aborts the `consume()` method with an failure indication. The client repeatedly has to call `consume()` until it succeeds. In the following the two implementations are compared with each other.

The counter object is created with five replicas on different hosts. One client is acting as a producer and a variable number of clients acts as producer, each of them on a separate node. In the experiment, the clients waited 1 ms between consecutive calls to the `consume` method. The measurements in Figure 3.24(a) show the invocation time of the `consume` method for an individual consumer, averaged over 500 invocations. The graph shows that the deterministic multithreading approach outperforms the single-threaded execution approach by 2%–119%. Whereas the single-threaded approach shows quite moderate performance penalties with only a small number of clients, the delay of the producer thread due to `consume` calls that fail gets more and more important as the number of clients rises. As expected, the multithreaded consumer approach scales approximately linearly with the number of clients, since blocked clients will only wake up if the counter is increased.

The last example evaluates the implementation of a time-bounded `wait()` operations. A replicated decentralised adaptive service offers two methods: `noop`, which returns immediately, and `delay()`, which performs a time-bounded `wait` with a timeout of 5 ms. Figure 3.24(b) presents the results from an experiment showing that the implementation of time-bounded `wait()` works with little overhead. One client continuously calls `noop`, while one to ten clients concurrently call `delay()`. The measurements shows the client invocation times averaged over 500 calls to a group of 5 replicas. The invocation time for the `noop()` operations rises from 4 ms to 10 ms due to the concurrency with other `delay` operations. The time difference between `noop()` and `delay()` exceeds the timeout duration 5 ms by




(a) Producer/Consumer

(b) Time-bounded `wait()`

Fig. 3.24: Producer/Consumer and Time-bounded `wait`

about 2–3 ms; this is due to the timeout message that has to be sent.

## 3.10 Case Study: A Replicated Decentralised Version Control System

To evaluate the framework for replicated decentralised adaptive services, a prototype of a version control system was implemented. Such a system is a key component for software development as it preserves the development history and enables the distributed and concurrent implementation by multiple developers. This is especially useful for all kinds of open-source software like the Linux operating systems and the Mozilla web browser that are concurrently implemented by hundreds of volunteers all around the world.

Currently most small and midsize open-source software projects either use the Concurrent Version System (CVS) [BF03] or Subversion [PCSF04] that is more or less an improved version of CVS. Both systems provide a central repository storage that stores the current version of each file and the deltas of all former versions together with a changelog. Thus, all information about previous source versions is stored at a central point and a developer has only its local working copy of certain source version and in case of Subversion the last version that has been requested from the central repository. Recently, several new systems like Arch [GNU], Darcs [Dar], Pastwatch [YCM06] and Monotone [HSS] have been implemented. These systems focus on a decentralised development model, where developers can have their own local repository that can include the whole version history and all source code versions of a project. This way offline work under local revision control is enabled. Later on, local changes can easily be reintegration into a central main branch usually maintained by a head developer. This approach is, for example, applied in the context of the Linux project.

However, some of these systems also allow the parallel usage of the centralised development model as the head developer could give write access to her repository and thereby allow authorised users to commit to a central server. Some of these next generation version control systems like Git [Bau], Pastwatch and Monotone use the benefits of cryptographic hash functions, like the secure hash (SHA-1), as they allow the reliable identification of files. Additionally, hash functions enable consistency checks and offer a lean and simple storage implementation.

For the prototype a reimplementation of the Git system was done that originally has been invented by Linus Torvalds as a consequence of a license problem with BitKeeper [Bit], the former version control system of the Linux project. Git has a lean storage model that can be implemented in a straightforward way. Another point in favour of doing a reimplementation of Git as a replicated decentralised adaptive service was the Linux project. As the prototype provides a binary compatible storage to the original Git implementation a copy of the Linux repository could be used for testing.

As described, Git already offers the decentralised version control of software. However this support primarily targets the local availability of a repository and the possibility to reintegrate local changes to a remote main repository either done by a head developer or if allowed by the developers themselves. A public main repository as in the case of the Linux project still represents a single point of failure. Once down, no further check-outs or check-ins of source code changes are possible. Thus, the centralised exchange of development progress is stopped. In addition, Git provides no out of the box support for mirroring or replication to achieve scalability and fault-tolerance. In context of the Linux project this is solved by considerable big hardware efforts (e.g. several Opteron multiprocessors, not only dedicated to the repository but also to provide binaries and snapshots) and replication via the file transfer protocol (FTP) and rsync [TM96]. For this reason the main goals of the reimplementation are to achieve fault tolerance and to some extend provided scalability without the need of external mechanisms and manual intervention. A version control system and especially a decentralised one demands for local file access and requires bulk data transfer for data-intensive operations. Both requirements can be met by replicated decentralised adaptive services and fragmented objects in general, as they allow to dynamically access the local file system via a local fragment and enable a custom object-internal communication. Summarising, a decentralised version control system builds an ideal test case for the proposed semantic annotation support to address the extended non-standard demands.

To take part in the development of a software project a developer checks out sources, modifies them, and finally commits her changes. Usually, these operations are performed using a standalone tool or by accessing a plugin of a integrated development environment. In context of this thesis a command-line oriented tool was developed but will not be further mentioned. Instead the focus is put on the internal management and operation of the service and the service interface. The latter can easily be mapped to the corresponding tool commands enabling also a simple integration in other tools and development environments.

In the following section the basic concepts of Git and its storage format is explained. Then, the architecture and the API of the decentralised adaptive variant of Git is outlined in two consecutive sections. Whereas the first one concentrates on local repository operations,

the second section discusses remote operations handling the management of the replicated object state.

### 3.10.1 Basic Concepts of Git

The storage model is composed of four basic object types that are stored as ordinary files on disk. Each object has a header that identifies the type and the size of the object. The file is named after the SHA-1 hashsum over its contents. This way any modification of the file can be easily detected, as the hashsum over the file contents no longer matches the filename.

The four basic object types are:

- Blob
- Tree
- Commit
- Tag

A *Blob* basically represents a file that is under version control. There is no option to reference another object from a Blob. A *Tree* object typically consists of a list of names referencing Blobs and other Tree objects together with metadata for each entry, like access time and permissions. This way a Tree object more or less maps a directory. Finally, there is the *Commit* object that uniquely identifies a certain committed version of the sources. A Commit object consists of a changelog message, name of the modified Tree objects, and the name of the direct ancestor Commit objects if there are any. A *Tag* object offers the additional support to uniquely identify and secure certain source code versions by a name and an optional certificate.

Taking the three main object types Blob, Tree and Commit the basic Git storage model is given as shown in Figure 3.25. The depicted example project consists of three files that are represented by Blobs. There is one Tree object that maps the project directory and one Commit object describing the initial version.

If a developer commits her changes to the repository all modified files hash to different values and therefore result in new Blobs. The depending Tree object changes, and consequently its depending hashsum. Consequently a new Tree object is generated. The Commit object documents the dependencies between the former version of the project and the new one. Figure 3.26 displays three check-ins of two files: the first one displays the initial version; the second revision has only one modified file; whereas in the third case all files have been modified. This structure enables a comparatively easy branching and merging of different development lines as only the modified files have to be stored and a new Commit object inserted that references the joint branches.

The Git storage model can be implemented in a straightforward way, and it prevents the undiscovered modification of files due to the rigorous usage of hashes. The drawback of the storage model is the introduced redundancy. Versions that differ only slightly result in

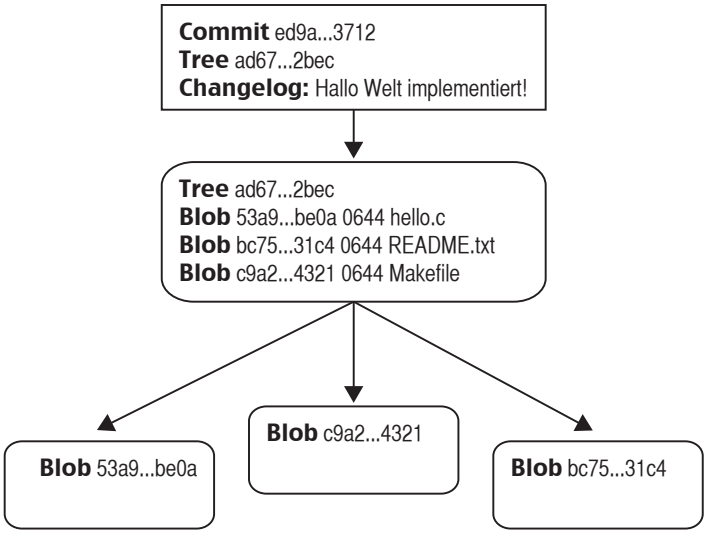



Fig. 3.25: Example of the basic Git storage model

completely new Blob objects. This leads to an enormous waste of storage. Common version control systems like CVS or Subversion usually store only the difference as deltas between two successive versions of files. To achieve a similar behaviour Git supports custom archives named *pack files*. A pack file is a collection of objects, individually compressed, with delta compression applied. Each pack file has an associated index file that references all packed objects. These pack files are used to reduce the repository size by archiving older revisions and to enable an efficient network transfer for data-intensive repository operations, like the initial check-out of a project.

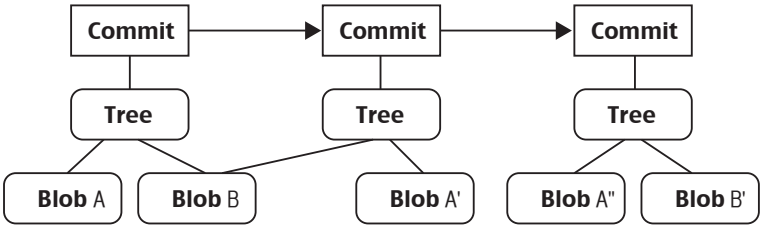



Fig. 3.26: Commit example

### 3.10.2 Local Repository Operations

Git features a distributed development model that requires that every developer has its own local repository. This is also the case for the replicated decentralised adaptive variant; every replica fragment but also every access fragment has a local storage. This storage is subdivided into two repositories one for local revision management mainly needed for access fragments and another one for the global revision management. The global part of the repository is replicated and managed by the replica fragments. At the implementation level both repositories share the same storage so objects that are known in both repositories

need to be stored only once. In this section it is examined how to do the local revision management (see Listing 3.27). At this point it is assumed that the client is already connected to the service and the local repository is populated by the sources of the global repository. How this is achieved is subject to the following section.

A developer can `checkout()` a certain source version from the local repository into her local working directory. New files have to be explicitly added to the repository via the `add()` operation. In contrary to other revision control system, `add()` marks not only a file for the next commit operation for addition to the repository but stores a copy of its current contents. If an added file is modified later on these modifications will not be considered for the `commit()` operation, which persistently stores a certain source version in the local repository. To include changes after the execution of `add()` the `refresh()` operation has to be called. The `refresh()` operation basically updates the earlier taken copies of added files but it is also used to store the current version of changed files that are already under version control. Thus, a developer has the option to collect distinct changes for the next commit while progressing in the development process.

To update the local working copy, `merge()` has to be called. If a certain source version should be used for further independent development this version can be tagged by the `branch()` operation.

All methods of this basic set of operations are annotated via the *local* keyword. This is required as all operations are handled by the access fragment that manages the local repository at client-side.

### 3.10.3 Distributed Repository Operations

This section targets the handling of remote and distributed repository operations. Several of these operations like the initial check-out of a repository and the update of sources are data intensive. Among many advantages of CORBA, one definite shortcoming is the missing support of efficient bulk data transfer. Similar shortcomings apply for the prototype implementation of replicated decentralised adaptive services and the underlying group communication framework. In context of plain CORBA this problem has been identified early and solved in various projects. Usually this has been achieved by providing frameworks [PHS96, KOS+00] or as in the case of Kurmann and Stricker [aTMS03] by a complete redesign of the communication stack and the ORB. Often specialised protocols are used for the data transfer and standard CORBA mechanisms for dependent control messages. Without having these solutions available, the fragmented object model and based on the support for client-side computing also replicated decentralised adaptive services provide an alternative solution, as a service developer has the freedom of choice to use arbitrary communication mechanism for the object internal communication. So a clear separation between control messages and file transfer has been made where it is possible. The implemented framework handles control and state modifying messages as standard method invocations whereas data is transferred via an adapted custom protocol taken over from the original Git implementation.

Before actually a developer can use the local repository a decentralised Git service has to be initialised and started. This is achieved by creating an initial replica and an associated

```
#pragma annotate(local)
void checkout(in string ref)
        raises (NonExistingBranch, NonExistingObject);

#pragma annotate(local)
string add(in string filePath);

#pragma annotate(local)
string refresh(in string filePath);

#pragma annotate(local)
string commit(in string ref) raises (NonExistingObject);

#pragma annotate(local)
boolean merge(in string ref) raises (NonExistingBranch);

#pragma annotate(local)
void diff(in boolean all);

#pragma annotate(local)
void branch(in string name) raises (NonExistingBranch);
```

Fig. 3.27: Basic operations to manage the local repository

IOR. Every service replica provides not only a complete copy of the repository but also an own TCP server for bulk data transfer that is started during the creation process of a replica. If an initial service replica has been successfully started, its IOR can be bound by any program wishing to access the service.

After binding the service, the local storage has to be populated by the sources of the replicated global repository before a developer can use the earlier outlined method set for local repository management. This is realised as a separate operation by the method `clone()`. Also implemented as local method this operation calls the *private* and *readonly* method `getRepositoryServers()` that provides all data server addresses. Then the access fragment selects one of the server addresses and requests the current repository state that is provided as a pack file by the TCP server of the requested replica. This pack file is used to set up a copy of the global repository. If the repository of the access fragment has already been initialised and should be updated this can be achieved by calling `pull()`. The `pull()` operation is also implemented as a custom local operation and requests the current state of the repository from one of the replicas. This replica returns meta data about all branches and their current head versions (the last commits). The access fragment compares this information with the entries of the global part of the repository and then determines the missing commit objects in an interactive process with the replica. Finally, all modified objects are provided as a pack file.

To commit local changes into the replicated global repository a developer calls the method `push()`. This method determines if there is a conflict with the global repository and generates a pack file that is handed over to the global state modifying method `pushPack()`. If the global state has been modified since the last comparison a conflict is signalled by the return value and has to be resolved before the changes can be committed to the global repository.

To set up a new replica a state transfer is required. Fortunately, every access fragment already possesses a local repository. However, this copy of the global replicated repository could be outdated as there might have been commits to global repository since the last invocation of `pull()`. To account this fact and minimise the state transfer the state transfer interface has been extended. The new replica submits revision information about its copy of the global repository to all state-providing nodes. In a next step this information is used to generate a pack file that is transferred to the joining replica. After this operation the new replica adds the address of its data server via `addRepositoryServer()` to set of state-providing replica servers.

```
#pragma annotate(local)
boolean clone(in string path);

#pragma annotate(local)
boolean pull();

#pragma annotate(local,restricted)
BranchArray push();

#pragma annotate(restricted)
BranchArray pushPack( in ByteArray pack,
                      in BranchArray oldBranches,
                      in BranchArray newBranches);

#pragma annotate(readonly,private)
StringArray getRepositoryServers();

#pragma annotate(private,restricted)
void addRepositoryServer(in string address);
```

Fig. 3.28: Operations to manage and synchronise the local with the global repository

### 3.10.4 Summary

The prototype implementation provides all basic features of a version control system and fulfils the initial goals, as it is due to the support of the proposed framework for replicated decentralised adaptive services fault-tolerant and to some extent scalable.

The current implementation makes intensive usage of the proposed semantic annotations to enable the local repository management. Moreover the opportunity to use arbitrary inter-

nal communications mechanisms turned out to be very useful to implement data-intensive operations like `pull()` and `clone()`. The aim to handle all data-intensive operations via direct socket communication has not been met as methods like `pushPack()` and the state transfer require the coordination and guarantees offered by the framework and the underlying group communication mechanisms. However, the service could be implemented without changing the framework or modifying the Aspectix ORB.

A possible extension for fast service migration and fault-tolerance beyond the offered active replication of the global repository could be achieved by running access fragments as mirrors that update their copy of the global repository on a regular basis. This way a quick set-up of new replica fragments is possible as less state has to be transferred.

## 3.11 Summary

This chapter presented an extended framework to provide replicated decentralised adaptive services. At its core the framework offers actively replicated objects extended by several unique features. First of all, the framework utilises the Aspectix middleware and its support for the fragmented object model, thus no modification to the core middleware is needed to support replication. Furthermore, the service replication is transparent to clients.

The proposed time-bound object references enable a secure service binding despite service mobility and the lack of central, always-accessible location service. If a location service is used a decentralised peer-to-peer network can be employed in a safe way.

To further support service mobility advanced techniques for state transfer were presented. This includes the proposal of a stream-based state exchange interface, the adaptation of algorithms for parallel download and the integration of techniques to reduce the transferred state.

To enable a flexible and easy development of new services and the port of existing ones basic support for deterministic multithreading is offered. This includes the support of native Java synchronisation mechanisms and the gracefully handling of nested remote method invocations.

More important, semantic annotations at the interface level have been introduced. In contrast to earlier work (see Section 3.7.1) the outlined annotations for locating custom code at client-side is novel. As demonstrated by the case study, the proposed semantic annotations support the implementation of complex distributed and fault-tolerant applications with non-standard internal interaction patterns.

Summarising, the presented service model is decentralised as it is able to tolerate a limited number of node faults and it is adaptive as service mobility is explicitly supported. Finally, replicated decentralised services represent the first answer of this thesis to the question how new service models for global-scale service infrastructures should look like.

# 4 Peer-to-Peer based Decentralised Adaptive Services

The previous chapter proposed replicated decentralised adaptive services as a fault-tolerant solution to implement services that are dynamically distributable and able to host information that can be concurrently accessed and modified by multiple clients. Albeit these features the proposed service model is not well suited for services that host large amounts of data that are accessed by a potentially vast number of clients but only modified by the information providers. Typical examples are lookup services, file sharing and instant messaging. These kinds of applications can be implemented using the replicated decentralised adaptive service model but their implementation will not scale above a certain degree. In addition, such applications have weaker fault-tolerance requirements that do not demand for active replication. In the average case avoiding a single point of failure is enough. This can save resources and enable more scalable service implementations as simpler mechanisms can be used.

Consequently, additional support is required to address these service demands. In the past few years numerous peer-to-peer algorithms have been proposed that offer scalable support for the lookup of data. These algorithms rely on the assumption that most modern computers are under-utilised and that spare resources can be spend for supporting additional services. Depending on the target application and the time resource providers share their resources, different algorithms should be used. For example, in a more static environment like a company, protocols supporting a structured network [RFH+01, DBK+01, RD01] perform very good and can be used for distributed file-systems, whereas for file sharing in the internet unstructured protocols [KM] or protocols supporting a so called super-peer architecture are better suited. Currently there is no best solution for all application and environment demands. Additionally, further protocols are still proposed. Thus, implementing a certain protocol and using this as the basis for supporting peer-to-peer based decentralised adaptive services is not an option. However peer-to-peer mechanisms are a solution to fit the earlier outlined service demands. Furthermore, they provide means to offer decentralised adaptive services as peer-to-peer protocols are designed to tolerate a limited number of node crashes and supply mechanisms to replicate data to account changing demand and environment conditions [RFH+01, DBK+01, RD01].

As detailed in Section 2.4, maximal flexibility is gained by using the fragmented object model, thus a peer-to-peer based object-internal interaction can already be implemented from scratch. On the downside, the implementation of peer-to-peer algorithms is not an easy task and the approach results in a high development effort, especially if needed in context of different service implementations. In the previous chapter, this issue has been alleviated by the proposal of an entire framework. This time a more lightweight approach was

taken. To facilitate a peer-to-peer based fragment-internal communication; JXTA [Gon01], a fully-fledged peer-to-peer middleware, was integrated in the Aspectix ORB by providing an own specialised profile manager.

Initiated by Sun Microsystems the JXTA project[1] can be considered as an effort to provide a generic and open infrastructure for peer-to-peer computing. Most current peer-to-peer applications share fundamental concepts, for example the search for information or the transfer of data. However, as earlier mentioned these applications usually implement common features in a different way and thus they are incompatible to each other, resulting in a set of isolated solutions. In order to establish a generic basis on which future peer-to-peer applications can be build, JXTA tries to standardise fundamental functions and concepts by the specification of six basic protocols [Soc05], whereby each protocol follows an asynchronous query/response communication pattern. This makes JXTA independent of a certain programming language, transport protocol, underlying network and hardware platform. More important it enables the implementation of different peer-to-peer protocols using a uniform set of mechanisms.

In contrast to current popular middleware systems like CORBA, RMI and .NET, JXTA offers only lower-level communication concepts like unidirectional, asynchronous connections, named *pipes*, and messages for remote communication. This is appropriate for developing peer-to-peer based protocols and services. However, JXTA misses higher-level support for concepts like an object reference and a remote interface as supported by current object-oriented middleware infrastructures. Furthermore, JXTA omits facilities for integrating JXTA services in general. However, this would be of high value as current distributed applications using traditional client/server interaction could be enhanced by a decentralised peer-to-peer communication to achieve fault-tolerance and as earlier outlined to improve scalability. Thus, the main contribution of peer-to-peer based decentralised adaptive services is to close this conceptual gap between the generic peer-to-peer infrastructure provided by JXTA and popular middleware infrastructures, in this case Aspectix and CORBA, by utilising the fragmented object model. In principle, the plain support for fragmented objects as already provided by the Aspectix ORB might allow implementing a fragmented object with JXTA-based internal communication without the support of a custom profile manager. However, this would be cost intensive as the JXTA platform would have to be setup during the binding process by the fragment implementation and this would even happen for every JXTA-based fragmented object anew which results in wasting a lot of resources. In contrary, the proposed integration based on a JXTA-aware profile manager offers a lean and easy integration of JXTA into the Aspectix ORB.

In the following section, a brief overview about related approaches is given. Next, the most important concepts and components of the JXTA middleware are explained. Not all of the provided information is necessary to understand how the Aspectix ORB has been enhanced by JXTA based services, however the section builds also the background for Chapter 5, Chapter 6 and Chapter 7 where peer-to-peer based decentralised adaptive services and JXTA are used to provide the EDAS environment and support for resource management. The actual integration of JXTA into the Aspectix ORB is achieved by a custom reference profile

[1] http://www.jxta.org

and a profile manager that are subject to Section 4.3. Finally, an example application is described that demonstrates the seamless integration of JXTA into Aspectix by using the fragment object model and the support of the newly provided profile manager.

## 4.1 Related Approaches

There is only few related work found that deals with the integration of peer-to-peer mechanisms in common object-oriented middleware infrastructures.

P2Pcomp [FHMO04], is an OSGi-conform [All05] peer-to-peer framework targeting at resource-restricted mobile devices. With the requirement to facilitate the development of distributed applications, the framework uses a peer-to-peer approach for communication and the model of *component-based programming* [Szy97] for reusing code. P2Pcomp utilises the OSGi concept of a *container*, in order to administrate the lifecycle of local components. To support a transparent communication between components of different nodes the concept of a *port* is introduced, which permits the use of any mechanism for communication between container instances, e.g., using JXTA communication. In contrast to the proposed approach, P2Pcomp focuses on Java interfaces for describing services that are provided by certain components and offers no seamless integration in a standard middleware, like CORBA.

To support future heterogeneous grid applications, Gridkit [GCBP05] illustrates the approach of a *deep middleware*. Gridkit has the goal to establish an infrastructure for the generic integration of different technologies at the network level, as well as on the middleware level. Therefore, a two-layered architecture is specified. The *interaction framework* enables an integration of interaction paradigms, as the RPC, using a plugin approach. In contrast, the underlying *overlay framework* permits a plugin integration of virtual network layers, e.g., Chord [DBK+01]. Thus, compared to the proposed approach, Gridkit focuses on communication mechanisms, integrating peer-to-peer mechanisms at a lower layer.

## 4.2 JXTA Overview

### 4.2.1 Peers and Peer Groups

A JXTA peer-to-peer network consists of peers, which syndicate to *peer groups*. Every *peer* is a uniquely identifiable node in a JXTA network, whereby several kinds of peers are distinguished [Mic05]. The two most important ones are *edge peers*, i.e., a common desktop computer, and *rendezvous peers* building the routing and lookup backbone of the JXTA network. In order to enable transparent communication with peers of variable connectivity to the network, JXTA offers with its *ID* concept a way to uniformly and location-independently address nodes.

Independently of its type, each peer belongs to at least one peer group. Peer groups permit the segmentation of a peer-to-peer network into diverse communities of peers. As centres of interest, a peer group provides a way to control the propagation of communication traffic in the common JXTA network. It is even possible to create private peer groups that are only accessible to trustworthy peers.

### 4.2.2 Advertisements

The availability of any network resource, e.g. peers, within JXTA is represented through *advertisements* [Soc05]. Basically, an advertisement is a programming-language independent metadata structure describing a resource, which is represented by an XML document. Additionally, advertisements are equipped with a globally unique ID of the described entity. Optionally additional information, e.g., human-readable names and descriptions are included. Accordingly, peers search for certain resources using their corresponding advertisements. Thereby, each advertisement is published within a certain peer group for a specified lifetime. During this time span, consumers can discover and use the resource via the JXTA resolver mechanism (see Section 4.2.4). A common peer operation is to purge its local cache from expired advertisements, which prevents the unnecessary storing of outdated information.

### 4.2.3 Messages and Pipes

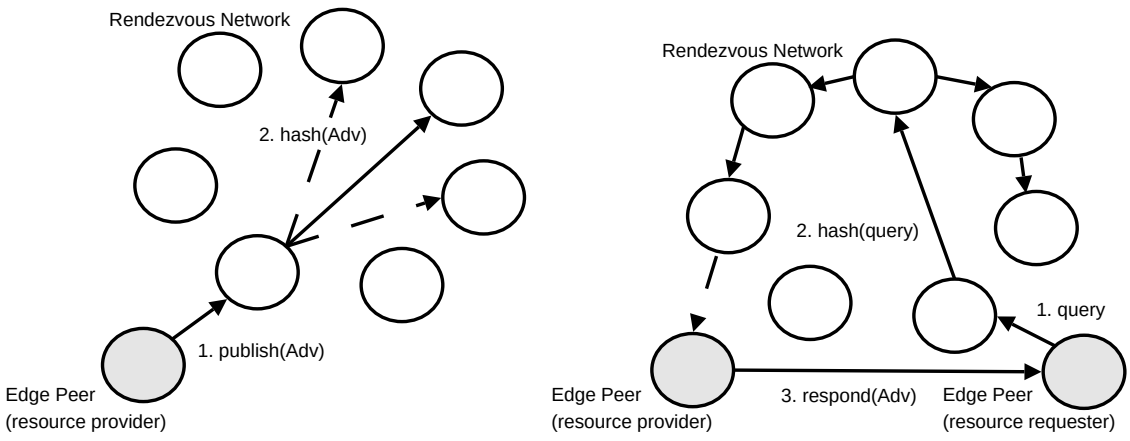
A *message* in context of JXTA designates a datagram, which is exchanged between peers and can take arbitrary data as payload. Besides, additional data like credentials or routing information can be embedded. Usually, a message is represented by a structured XML document with ordered name/value pair elements [Soc05].

JXTA enforces the abstraction of *pipes*, unidirectional, asynchronous and virtual communication channels, in order to transfer arbitrary messages between peers of the same peer group in an unreliable way. A transmitting and a receiving peer of a message do not have to possess a direct physical connection, nor does a pipe need to be bound to a special physical location. In addition, the endpoints of a pipe can be bound dynamically at run-time, even to different peers. Therefore, JXTA introduces two different kinds of pipes. On the one hand, the *point-to-point pipe* allows a communication between exactly two peers. On the other hand, the *propagate pipe* permits a multicast communication between a set of peers. In order to provide a bidirectional and reliable way of communication, higher communication constructs can be established like offered by the JXTA reference implementation by providing JXTA sockets.

### 4.2.4 Resolving Resources in JXTA

Based on unique identifiers and advertisements, JXTA offers a universal resource binding mechanism called *resolver* [TAA+03]. It is used to perform all kind of resource resolution operations such as the lookup of a peer name, a communication endpoint address or a service. Resource resolution is unified by the discovery of one or more advertisements. JXTA specifies only a basic protocol for resource lookup that can be implemented in different ways ranging from a centralised to a totally decentralised resolving process depending on the demands of the application and its domain.

Despite this openness, the JXTA reference platform offers a default resolving mechanism based on a rendezvous super-peers network. Every edge peer connects to a rendezvous peer and submits an index of all advertisements provided by the peer. All rendezvous peers build a network of super-peers. The rendezvous peers are organised in a ring-topology ordered




(a) Publish an advertisement

(b) Search for an advertisement

Fig. 4.1: Resolving resources in the JXTA rendezvous network

by the hash-value of their peer ids. Advertisement-indices provided by edge peers are redistributed in scope of the rendezvous network according to hash keys generated from the indexed advertisements. The index data is replicated over several nodes to tolerate node faults and appropriately redistributed to compensate peer changes. Figure 4.1(a) shows a simplified example of an edge peer that publishes one advertisement via its connected rendezvous node that propagates the information to the responsible rendezvous nodes.

If an edge peer queries the local resolver service the query is forwarded to its current rendezvous peer (see 4.1(b)). After hashing parts of the query it is routed to the responsible rendezvous peer. As the knowledge about the network of the query-forwarding rendezvous peer might be partially outdated the receiving node does not necessarily possess the requested information despite it is available in the network. Such situation might arise due to peer churn. To address this issue JXTA uses the concept of a *limited range random walker*. The responsible rendezvous peer that should hold the data but does not, sends requests to its neighbouring peer nodes. These peers either respond to the query if they host the requested information or forward the query along the ring. If the requested information is not available within a certain number of routing steps the lookup process terminates.

### 4.2.5 JXTA Architecture

The JXTA architecture is composed of three layers: core, service and application layer. The core layer offers basic functionality for peer-to-peer communication. In particular, the earlier described, necessary primitives for the management of peers and inter-peer communication are provided. The service layer is responsible for generic services that might be required for common peer-to-peer use cases, like file sharing or indexing. The highest layer, the application layer is reserved for applications developed by the JXTA community. In scope of this thesis all presented services belong to the application layer.

In order to offer a functionality in an easy and extensible manner, JXTA employs a generic module framework. The modules managed by the framework are responsible for providing

all aspects of JXTA's functionality. Thus, the concept of a module is essential in order to load and execute new services on the JXTA platform [Wil02]. A JXTA *module* thereby represents a distributable unit of functionality that can be initialised, started and stopped by a peer. Additionally, JXTA distinguishes a *service module* to represent a component used to run a service and an *application module* to act as an application. Using this approach, a JXTA application can be composed of several service and application modules. However, as mentioned at the beginning of the chapter there is no support for combining different modules in the sense of a component model or something similar.

To enable an efficient way to characterise and discover modules, the definition of a module is divided into three types of advertisements:

- Module Class Advertisements
- Module Specification Advertisements
- Module Implementation Advertisements

Whereas the *module class advertisement* announces the pure existence of a unique class of modules that can be compared to a category (e.g., file sharing services), a *module specification advertisement* determines a concrete variant of a service and its supported query/response protocol. This is achieved by including a link to a document specifying the supported protocol. As JXTA is platform-independent, there can be more than one module implementation that realises a certain protocol. Each implementation is described by a *module implementation advertisement*. The module specification advertisement misses information how the service can be accessed by an application. This is delegated to the code that is referenced by the module implementation advertisement

### 4.2.6 Summary

JXTA offers a set of basic primitives and abstractions to enable the implementation of peer-to-peer protocols on a common basis. On the downside the offered support for modules and services is limited to the management of life-cycle operations. Furthermore, modules provide only information about their network-direct communication and interaction via their module specification advertisements. What is essentially needed and missing is information how a module can be accessed by the application and consequently how several modules can be combined. Thus, an application that accesses a module needs a priori information about the module-specific interface.

## 4.3 Mapping JXTA Services on Fragmented Objects

In Chapter 2, the generic and flexible profile manager architecture of the Aspectix middleware was described. To combine the fragmented object model and the JXTA middleware to supply peer-to-peer based decentralised adaptive services, an own profile manager was developed. This required answering the following questions: First, how can information that is already standardised by the CORBA reference be mapped on existing JXTA concepts?

Second, which concepts of JXTA match best a fragment implementation? Third, which information need be included in a new JXTA IOR profile?

Every CORBA reference includes an interface and additional information to uniquely identify and bind a remote object. After successfully binding an object it can be accessed via its interface on behalf of a proxy. JXTA completely misses the concept of a remote object and consequently offers no support for remote references. Instead a service can be bound using its advertisements but as declared by the previous section it is not specified how to access the service by the application. Furthermore, service instances interact with other service instance supporting the same protocol. Thereby, it is usually not important exactly which instance is connected. Thus, a simple and direct mapping comparable to a bridge between one middleware and another one is not feasible. Instead, the support offered by the JXTA platform has to be adapted and mapped to meet the requirements of an object-oriented middleware.

A simple and straightforward approach would be to use JXTA only as communication infrastructure. Then method calls could be directly mapped to messages transmitted via a JXTA socket that provides a reliable socket-based connection mechanism on top of the already described pipe support. The remaining process of request handling and message marshalling can be left unchanged. Although using JXTA as communication infrastructure offers benefits like the comparably easy communication across firewalls and binding support despite changing physical communication endpoints of a server, it misses the key point of using JXTA, the peer-to-peer based interaction. Thus, this solution would neither offer improved scalability nor provide fault-tolerance.

Therefore a more intrusive approach was chosen that focuses on the original ideas of fragmented objects. As described in Section 2.4, a fragment implementation is not limited in its function set or capabilities. It can be anything ranging from a plain stub to a complete service implementation. For JXTA, similar flexibility is given, as an application or service can act as a pure client requesting information from other peers or provide a fully-fledged service that is accessed by all peers of a peer group. Apart from these general conceptual considerations, a useful convention has to be proposed to combine the fragmented object model and JXTA. Such a convention can be established by mapping a fragment implementation to a JXTA service. Both have similar degrees of generality.

This basic approach has various benefits. JXTA services and modules miss easy accessible application-directed interface (e.g., there is no information inside a module specification advertisement or the module implementation advertisement). By mapping a JXTA service to a fragment implementation this interface is implicitly given by the interface of the fragmented object. On the other side a fragment implementation usually needs to communicate over the network with other fragments to provide the object functionality. A JXTA service provides such a network support.

To characterise a JXTA service one can use its service specification advertisement. This advertisement declares how the service interacts with the network additionally it includes a pipe advertisement providing the necessary information to connect a service to other service instances. Thus, a service specification advertisement builds the ideal basis to identify a group of related service instances that can be be easily included in a CORBA reference as

a new profile. As previously mentioned it does not determine a concrete implementation. The last fact is considered in Chapter 6 as it allows dynamically selecting and loading of a platform-specific service implementation. Still, a fragment implementation is not limited to a single service or module implementation as the initial fragment can instantiate further modules and services as needed.

At the implementation level, a developer of peer-to-peer based decentralised adaptive services has to specify an object interface and then generates skeleton code for a plain fragmented object by using the IDLflex. Next, an appropriate JXTA-aware fragment implementation has to be implemented that maps method invocations of the object interface to appropriate messages that are transmitted via pipes or handled using other JXTA-provided mechanisms. Finally, the developer has to provide a basic service specification advertisement that omits a concrete pipe advertisement.

The pipe advertisement has to be generated at deployment time for a specific object instance; otherwise different objects could accidentally connect each other and break the encapsulation of the objects.

## 4.4 JXTA Profile Manager

During the initialisation phase of the ORB at start-up of an application, the JXTA platform has to be configured and connected to the peer-to-peer network. For easier interaction with the JXTA platform and to offer extended support for the dynamic loading infrastructure outlined in Chapter 6, beside the profile manager class `JXTAProfileManager` a custom wrapper for the JXTA platform named `JXTAHandler` is provided. Implementing the singleton pattern, this class connects the peer-to-peer network on creation and is accessed by the profile manager and all local fragments. This way it is ensured that only one instance of the JXTA middleware is instantiated.

The profile of a JXTA-based fragmented object contains a module specification advertisement. The profile manager offers with `insertProfile()` an appropriate method to create a JXTA profile. If a reference containing such a JXTA profile is bound, the included module advertisement can be resolved by the JXTA profile manager to an implementation, and loaded via the process described in Chapter 6 if it is not locally available. Afterwards, the fragment implementation is instantiated and has to be initialised. As this is a fragment-specific task, every fragment implementation provides a standardised constructor that is executed by the profile manager, passing the `JXTAHandler` instance and the module specification advertisement. Then, the implementation is free to bind the pipe advertisement included in the module specification and to use the JXTA platform as needed (e.g., to start further services or modules).

## 4.5 Case Study: Multi-User Chat Application

For demonstration purposes, a basic multi-user chat application was developed. In a standard CORBA environment, one would implement and set up a server providing at least two methods, one to register clients and another method to publish messages (see Figure 4.2).

```
module chat {
  interface chatListener {
    void receiveMessage(in string name, in string message);
  };

  interface chat {
    void registerChatListener(in chatListener listener);
    void sendMessage(in string name, in string message);
  };
};
```

Fig. 4.2: Chat application interface (IDL)

If a user publishes a message, all clients that are registered at the server, using a listener interface, are notified. This is a convenient and efficient implementation if there is a permanently available node that hosts the server. However, if this node crashes or is offline for other reasons, e.g., system maintenance, the service is no longer available.

A peer-to-peer based implementation using the proposed support for peer-to-peer based decentralised adaptive services eliminates this issue and potentially offers better scalability than the presented approach for replicated decentralised adaptive services. A developer has to implement the same service interface, as done in case of a traditional client/server implementation and additionally he needs to specify and implement the network protocol. To exchange messages with other users, a propagate pipe is used that multiplexes messages to all connected participants. As every client can send and receive messages to/from all other participants and the peer-to-peer network transmits and routes these messages, there is no single point-of-failure and no need for a permanently available server.

At deployment time of the service a pipe and service advertisement have to be created to provide the necessary object-specific information for lookup and to enable communication with other users. Finally, an object reference has to be generated, including a JXTA profile hosting the service advertisement. Following this approach, client applications do not need to be changed as they can bind a JXTA-based fragmented object as any other plain CORBA object. At the middleware level, the code is dynamically loaded as described in Chapter 6 and the first fragment is instantiated and initialised by binding the service reference.

## 4.6 Summary

As outlined at the beginning of this chapter, replicated decentralised adaptive services offer a solution for implementing dynamically distributable services, which serve data that is concurrently accessed and modified by multiple users. The drawback of the underlying active replication support is its limited scalability. To solve this issue for certain applications that primarily target the provision of read-only data this chapter proposes the usage of peer-to-peer mechanisms. This is achieved by combining the fragmented object approach and JXTA services thereby providing peer-to-peer decentralised adaptive services. This enables an exchange of a client/server interaction by more scalable and decentralised solutions.

As the basic example application outlines, it is possible to use a JXTA-based fragmented object instead of client/server-based solution without any modification at client-side. This has to be paid by the implementation of a JXTA service. However it is assumed that this introduces only moderate overhead and is legitimated by gained improvements. In contrary to the presented related approaches, peer-to-peer techniques can be used in context of a common object-oriented middleware.

Whereas peer-to-peer based decentralised adaptive services can be definitely considered as decentralised according to the definition given in the introduction, their adaptiveness by supporting dynamic distribution is not apparent. At this point it is assumed that a fragment instance given by a JXTA service has not to be explicitly migrated. The reason is that the underlying peer-to-peer mechanisms replicate data to address demand and environment changes. Thus, providing additional support for migration is not necessary. However, based on previous work in context of the CORBA Life Cycle service, migration support for JXTA services can be supplemented to the provided support [KSSH06, KSH05].

Finally, peer-to-peer based decentralised adaptive services give a second answer to the question how new service models for long-running services should look like.

# 5 Environment for Decentralised Adaptive Services

As motivated by the introduction, this thesis aims at providing a generic platform for the stable provision of long-running services. This chapter presents EDAS that provides the basis to distribute and manage decentralised adaptive services.

These services are usually run on behalf of a project or community. A typical example is an open source project that is aided by multiple research institutions, companies and individual users. To support the development process, such a project usually demands for several services: A webserver hosting the project homepage, providing information about the project; a source code repository for coordinating the development process; an issue tracker for reporting bugs; one or more mailing-lists for knowledge exchange and finally a solution for instant messaging. Currently, most open-source projects are hosted by *SourceForge*[1] or one of the few other central portals for open-source development that provide these services as a preconfigured suite.

Although the offered services are convenient and available for free, relying on a central open-source portal has several drawbacks: First, portals like SourceForge are usually owned by one company and if this one drops its support, the development process stands still until a new project ground has been found. Second, only a limited set of services is offered thereby restricting flexibility (e.g., it is not possible to use a different bug tracker). Third, resources usage is in general bound by the portal provider, posing a potential bottleneck.

The two proposed service models offer already an alternative to provide the demanded services in a decentralised and adaptive manner (see Section 3.10 for a replicated decentralised version control system and Section 4.5 for a multi-user chat application). This means, they can be dynamically distributed over multiple sites and change their distribution according to the requirements of the environment and the users. This eliminates the first mentioned drawback of using centralised open source portals, as services can be migrated to another location if a resource provider drops its support. The second drawback can be solved by simply allowing users to get more control on how the offered resources are used. This demands for extended platform support, e.g., enabling the dynamic deployment of custom code. The third issue has usually monetary reasons, as reliably providing an extensive set of resource is expensive. This issue can be solved by offering support for multiple sites to share and dedicate their resources to projects in an easy kind of way.

All three mentioned issues and their proposed solutions demand for an extended infrastructure that enables: the management of decentralised adaptive services, the support for providing custom project-specific services, and finally the utilisation of resources offered by multiple providers.

[1] SourceForge http://sourceforge.net/

This chapter focuses on the basic EDAS infrastructure that aims at virtualising resources provided by multiple supporting entities to be independent of a specific party, provide better scalability, and in general offer more flexibility as projects can host their own services. Thereby, EDAS is not limited to offer a decentralised management and execution platform for services for software development management, but can also be used to distribute all kinds of services.

The EDAS architecture is composed of two major conceptual components (see Figure 5.1): The *home environment* is provided by every peer that actively supports decentralised, adaptive services and the EDAS platform. It manages the resources of all peers belonging to the same administrative domain. The *service environment* is spread over a set of domains that support a group of decentralised, adaptive services run by one project or community. These adaptive services are dynamically distributed within the scope of their associated service environment.

Both new components were implemented using the two proposed service models as is shown in the remainder of this chapter. This emphasises the usability of the proposed service models and the way they extend the CORBA middleware platform. The basic EDAS architecture is enhanced in Chapter 6 by mechanisms for dynamic code loading to enable easy service deployment and in Chapter 7 by resource management support to enable a resource-aware service placement. Taking theses facts together, an infrastructure is provided that solves all mentioned three issues of current central open source portals.

Subsuming the contribution of this chapter, in contrary to common object-oriented middleware EDAS provides a basic multi-domain support to manage services distributed over several domains. Furthermore, EDAS builds another use case for the two proposed service models that are both required to provide the necessary functionality as it turned out that for some tasks peer-to-peer techniques are more suitable (e.g., lookup of a factory) whereas others required the use of active replication (e.g., fault-tolerant management of resource management).

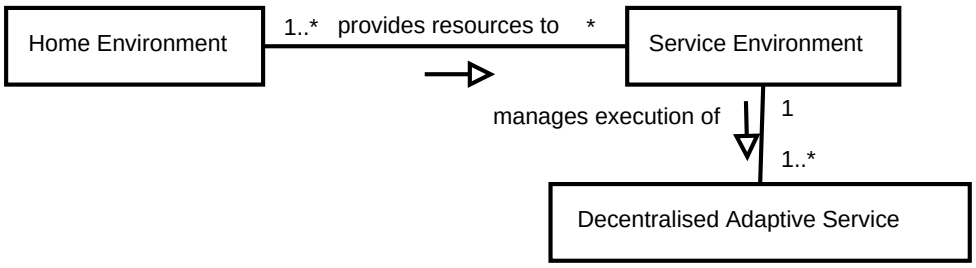



Fig. 5.1: The three core components of EDAS: home environment, service environment and decentralised adaptive service

The following section targets the state of the art of infrastructures enabling the execution and management of distributed applications. Then, the core components of the EDAS infrastructure are outlined. Next, it is described how decentralised adaptive services were integrated into EDAS. Finally, a system overview is given that combines the outlined components.

## 5.1 State of the Art

This chapter focuses on the basic infrastructure for executing and managing decentralised adaptive services. The idea of distributing applications across a set of nodes supported by an infrastructure has a long history. Consequently, a broad overview about related approaches is given and some more closely related systems are briefly present. Furthermore, this section addresses the general approach of these systems to enable distribution, namely the entities and techniques that are provided. Some of the outlined approaches offer extended support for code deployment or resource management. The first aspect is covered in Chapter 6, whereas the second issue is handled in Chapter 7.

### 5.1.1 Distributed Middleware

In the following, systems that provide some form of middleware supporting distributed applications and services are described. Thereby, middleware is considered as a layer, between the underlying operating system and the supported application that provides an abstraction of distribution aspects and platform heterogeneity.

#### Client/Server Middleware

Parts of this section have a close relation to Section 3.5.1 that discussed the issues of state transfer when migrating non-replicated objects or creating replicas of actively replicated objects and Section 3.8 that details the basic run-time infrastructure of replicated decentralised adaptive services. However, this section does not focus on migration issues but addresses infrastructure support for dynamic distribution at a broader scope.

Early RPC-based middleware systems [BN84, Gro88] offered only basic support for location transparency but missed extended mechanisms for dynamic service distribution. This changed with the advent of object-oriented middleware systems that provide additional means to support the life-cycle management of objects, including object mobility and support for persistence.

CORBA offers a set of so-called Common Object Services that provide generic functions to alleviating the task of developing distributed applications. Consequently, the OMG specified the CORBA LifeCycle Service [Obj02] to manage the distribution of objects. This is achieved by defining a set of interfaces and proposing the usage of certain design patterns. An object that is subject to life-cycle operations has to support an interface that includes methods to move, copy and delete the object. The methods have to be implemented by the object developer. Furthermore, there is the concept of a remote *factory* to create object instances at remote sites and a kind of registry for factories named *factory finder*. If an object should be moved from one node to another its `move()`-method is invoked and a factory finder instance is passed to the object. The object queries the finder for an appropriate factory. Once a suitable factory has been retrieved from the factory finder, this factory is invoked to create a stateless instance of the object. The original object is then responsible to transfer the state to the newly created object instance. The service specification is imprecise and thus leading to incompatible custom implementations, which prevented most ORB-vendors

to provide the service at all. Despite this problem, it has been used to solve computing intensive problems [PG00] and the author has shown that the service can be implemented in a platform independent way [KSH05] using only CORBA standard technology.

Java RMI and the .NET platform offer similar support for dynamic service-side instantiating of objects as CORBA and permit the provision of persistent object references (see Section 3.4.1 and Henning [Hen98]). This offers the basis for object mobility but both platforms miss its actual support. However, using the existing support they can be supplemented as done for Java RMI by Avvenuti and Vecchio [AV01].

The Web service technology [GGKS02] can be considered as an alternative to common object-oriented middleware solutions. A Web service is characterised by an interface that describes a collection of operations that can be accessed via the network through standardised XML messages. This is achieved by providing a service description, which gives all of the details necessary to interact with the service. As a generic basis for distributed application, Web services miss support for mobility or distributed service management. However, appropriate facilities can be implemented on top, as shown for migratable Web services [HL06].

The Ice platform is another general-purpose middleware similar to CORBA [Hen04]. For service management it offers additional support provided by the IceGrid-extension that consist of a location service named *IceGrid Registry* and the *IceGrid Node* daemon. The location service has to be preconfigured on every node and supports reference resolution despite location changes of objects. The IceGrid Node daemon is usually provided by every node of a domain. The daemon is responsible for service activation and monitoring. Applications can be dynamically deployed via IceGrid using XML-defined deployment descriptors. Subsuming, a lean and easy to use deployment and management infrastructure for distributed client/server applications is provided. The basic concepts match the proposed solutions in the CORBA-context [Hen98].

The FT-CORBA specification provides the standard to build fault-tolerant applications using CORBA. Various aspects of this specification were already discussed in Chapter 3. The actual architecture for service management was mentioned in context of basic infrastructure to instantiate replicated decentralised adaptive services even so the proposed solution follows a slightly different approach. The core of the infrastructure builds the *replication manager*. It is responsible for managing object groups that form a replicated service. To set up replicas, every node of a domain provides a *generic factory* that is capable to instantiate any type of object replica. Furthermore, every node hosts a *fault monitor* that checks the local object instance (e.g., via dummy calls). If a fault is detected it is reported to a *fault notifier* that in turn notifies the replication manager. If passive replication is used additional components are provided for logging of requests and recovering of failed replicas.

All of the mentioned systems are quite similar and provide some sort of factory that is located at remote nodes to enable remote object creation. These factories are registered at a location service that takes the role of a factory finder to enable the lookup of suitable location.

#### Middleware Supporting the Fragmented Object Model

As detailed in Section 2.4, the fragmented object model enables the implementation of distributed objects that can have an arbitrary internal structure and use custom internal communication. SOS [SGH+89, MGNS94] and Globe [HvDvS+95, vSHT99] are basic middleware environments that support the fragmented object approach or distributed shared object as it is called in the context of Globe.

Globe proposes an internal structure for local fragments (named local objects) that is composed of several sub-objects that are responsible for specific tasks like replication and communication. At the infrastructure-level, Globe provides a two-level lookup process to locate objects. There is a naming service that returns an object id if an object has been registered. This id is then passed to a scalable location service that returns the actual contact addresses of the object [vSHBT98]. Using the contact addresses, a local object can be instantiated using an implementation repository. Similar support but less scalable is offered by the SOS infrastructure that provides a so-called acquaintance-service, a fragmented object that is hosted by every node and acts as a location service.

Subsuming, both infrastructures support the creation, binding and lookup of fragmented objects. State migration issues are not explicitly addressed by the middleware but subject to the implementation of the fragmented object.

#### Distributed Runtime Environments for Object-Oriented Applications

All previously detailed middleware platforms offer at least basic mechanisms for object mobility and management of distributed applications. These mechanisms are exposed by some form of service interface offered by the middleware or services layer on top of the middleware. Systems like Orca [BT88, BBH+98] and Emerald [JLHB88] follow a different approach by providing a distributed execution environment and a specialised implementation language that manages distribution. Orca for example performs compile-time analysis and runtime statistics to decide about object distribution. In contrary to this approach, Emerald does not completely hide distribution but offers the programmer a fine-grained distribution interface that is embedded into the Emerald programming language. This way the developer has more control about the actual distribution of a program. Especially for the Java programming domain these ideas have caused the development of several systems that explicitly target the domain of high performance and grid computing [FJ02, vNMW+05, SH99].

Whereas custom language support and extended support by the runtime environment alleviate the task of developing distributed applications they restrict the flexibility to provide custom solutions and tie applications to a run-time execution environment.

### 5.1.2 Middleware Supporting Mobile Code

Up to now, only infrastructures were considered that rely on the assumption of a closed distributed system in which all nodes belong to the same administrative entity and therefore possess a common trusted code base. This assumption cannot be held for open distributed systems. Two general approaches active networks and mobile agents that rely on mobile code are briefly summarised in the following.

#### Active Networks

Active networks [TW96, WGT98] are an approach to dynamically update or replace current network protocols. The basic idea behind this approach is to augment network packets with code that can be executed in the network at routers and switches. This allows the dynamically reprogramming of a network to fit new demands. Albeit the idea is compelling, it has several drawbacks concerning security and resource consumption. Executing untrusted code is always a security risk and extended actions have to be taken, e.g., using a sandbox mechanism or a restricted execution model. It is furthermore unclear how the resource usage of active networks can be accounted and if resource consumption could be refunded by the code providers. Finally, at the infrastructure level active networks require code-aware network entities that scan packets and execute attached code. Especially, the last issue prevented the widespread deployment of active networks, as infrastructure providers did not offer appropriate facilities. Apart form this problems active networks attack a different problem domain than decentralised adaptive services.

#### Mobile Agents

Mobile agents [LO98, MLC98, BRST02, WHN+01] are another paradigm that relies on mobile code. Here the idea is to encode some form of decision process in an agent program that is mobile and executes on behalf of a user. Security and resource consumption have also been issues in context of mobile agents but could be solved in large parts. The more severe issue is the fact that all application problems that can be handled with agents can also be solved using standard middleware approaches. Thus, the benefits are usually too low to employ the mobile agent paradigm and therefore the use of mobile agents is restricted to some niches like mobile computing. Every mobile agent platform offers a service that is responsible for the actual migration process. This service handles the state transfer including the coordination issues to take a consistent state snapshot and the initialisation at the target platform. As an additional service, a platform usually offers some form of registry to enable the communication between user and agent but also among agents. In some sense, a decentralised adaptive service can be compared to a replicated mobile agent as it might decide on its own how to adapt its distribution. However, mobile agents are not suiteable to provide the targeted services (e.g., a replicated decentralised version control system).

### 5.1.3 Distributed Deployment Platforms

Distributed deployment platforms focus in contrary to the previously outlined approaches more on the aspect of deploying arbitrary applications than to provide a concrete implementation basis for distributed applications. Usually, the deployment is time-restricted as is the case for a compute-intensive job deployed on a high-performance computing grid.

#### Grid Computing

Grid computing [FKT01] provides services and mechanisms for distributed heterogeneous environments to combine resources on demand to solve resource-consuming and compute-intensive tasks. This is achieved by the proposal of a *virtual organisation* (VO) that can be distributed over multiple sites. In scope of a VO, resource-usage policies can be declared for groups of users or even single users. Grid infrastructures usually offer several services that support the lookup and management of resources that are for example exposed as Web Services [Fos05].

The concept of a service environment can be compared to a VO but the targeted applications are fundamentally different as EDAS focuses on long-running services that possess different resource demands (see Chapter 7).

#### General-Purpose Deployment Systems

The basic idea behind the PlanetLab [PACR03, BBC+04, PR06] is to offer a global-scale testbed for novel distributed applications, services and new network protocols. This is achieved by providing a customised Linux operation system that is executed by every supporting entity. On top of the operating system, additional support for virtualisation is offered that provides so-called *slivers* that represent an operating system sandbox that should fit the average demands of a distributed application. All nodes of the PlanetLab are managed by a central site that hosts a database informing about the status of all nodes. Researchers can contact the database either directly or via additional services (see Chapter 7) and request a number of slivers that are created on demand at the requested locations. This way, researchers can select a set of locations to deploy a new application or service. The actual deployment process is out of scope of the PlanetLab support. Concluding, PlanetLab can host any distributed application that is executable in the context of a Linux operating system.

The XenoServer [HHaIP03, Kot05, KMP+04] can be partly compared to PlanteLab but has even more general goals that are summarised by the idea of *global public computing*. Here the idea is to have multiple resource providers that offer nodes that host the XenoServer software which provides not only support for hosting services that can be executed in a Linux operating system but to deploy whole operating system images that are executed on top of the Xen hypervisor [BDF+03]. Thus, the provided support is even more generic as the deployed applications are not restricted to a particular operating system platform.

### 5.1.4 Summary

All of the presented client/server middleware platforms miss multi-domain-support that enables a seamless deployment of applications over several administrative entities. The same applies for the presented distributed execution environments and the two middleware platforms supporting the fragmented object model.

State of the art infrastructures for grid computing but also general-purpose deployment systems as PlanetLab and XenoServer support the distribution over multiple domains as these systems explicitly target the global-wide distribution of programs.

Especially the latter two platforms abstract from the hosted services. This offers the freedom of choice to use a standard middleware or any other custom application that can be run on top of the provided infrastructure. This is fine for a deployment of legacy applications but if applications should be aware of distribution issues or even manage their distribution, additional interfaces to the underlying platform have to be provided. Thus, applications and the hosting infrastructure have to be aware of each other.

Grid computing is specialised for the provision and management of large data- and compute-intensive applications. These applications usually have a workflow that is executed and managed by the grid if such a support is provided by the platform. Thus, the targeted application domain is different from decentralised adaptive services that run for long possible infinite time.

EDAS offers on top of a standard middleware extended support for decentralised adaptive services. Unlike standard middleware platforms, EDAS explicitly targets the support of multiple domains. Furthermore, EDAS is not dedicated to compute- or data-intensive applications that have different demands from long-running and user-accessed services. Finally, EDAS offers a uniform view on the platform and the services unlike the presented general-purpose deployment systems that provide a more generic basis.

## 5.2 Home Environment and Associated Services

As outlined at the beginning of this chapter the home environment represents a mediator between the peers of an administrative domain and one or more service environments, each running a set of decentralised, adaptive services (see Figure 5.1). Figure 5.2 shows three domains, each hosting a home environment spanning all peers of the respective domains. This is achieved by implementing the home environment as a peer-to-peer based decentralised adaptive service that creates an own peer group for a domain. Usually, every peer provides a set of resources. These resources are monitored and managed by the local home environment fragment as detailed in Section 7.2. Furthermore, the home environment fragment informs about important local system events. For example, if a peer is shut down, all affected service environments are notified and can react appropriately (e.g. migrate their service components to another location).

To place a service or a service component on a peer, every machine hosts a *generic peer factory* that offers an extended version of the factory interface outlined in Section 3.8. These factories are described by custom advertisements that are published in scope of the home environment peer group and can be queried via the home environment. Internally, the home environment utilises the default resolver support of the JXTA platform (see Section 4.2.4). Therefore, the home environment takes the role of a factory finder and offers an appropriate interface.

The generic peer factory is used to instantiate any kind of fragmented object regardless of if it belongs to a service environment or the home environment. If a requested fragment creation concerns a management facility of service or home environment, the fragment is locally instantiated. If the creation-request targets the instantiation of a service, a new virtual machine instance is set up and the service fragment will be started in scope of the

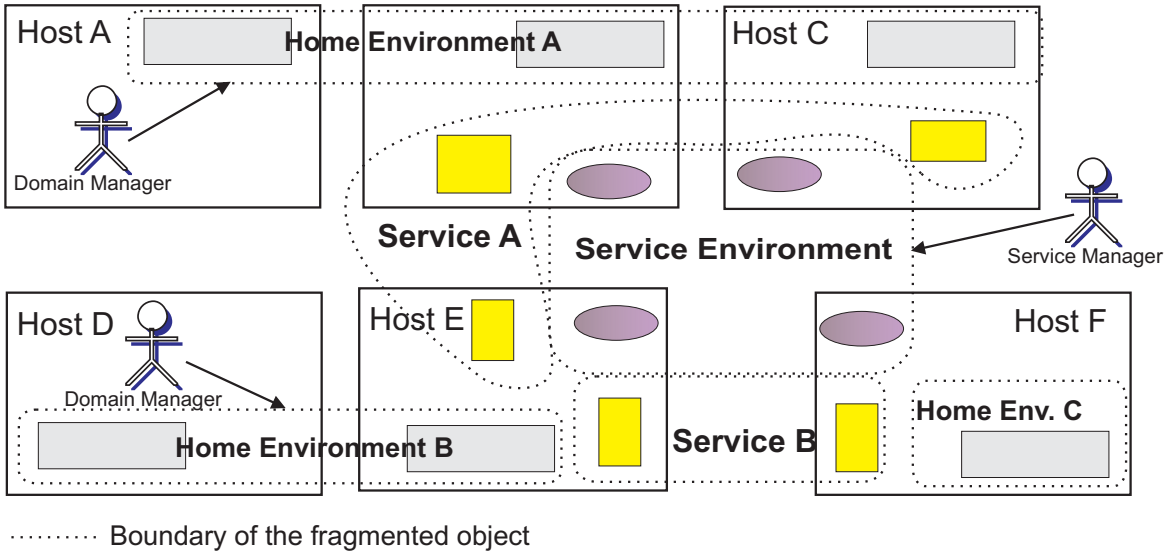



Fig. 5.2: EDAS Scenario comprising three home environments supporting one service environment that hosts two services

new execution environment. This has resource management reasons that are outlined in Section 7.2.

Each domain has a *domain manager*, a person who assigns and revokes resources to service environments. This information has to be reliably stored despite node faults and can be concurrently accessed and modified. Therefore, information about assigned resources cannot be hosted and managed by the home environment itself as JXTA and therefore the home environment provides no support for reliably storing information. Instead, resource grants and their management are performed by a supplementary service, named *home environment management service (HEMS)* that is realised as a replicated decentralised adaptive service. This service is initially placed on some nodes of the domain following the creation process outlined in Section 3.8. If one of the nodes that host a HEMS crashes or reboots, the service autonomously queries the home environment for generic peer factories and instantiates a new HEMS replica. This way, the service should be available as long as the home environment offers a sufficient number of peers. Summarising, conceptually the HEMS builds a component of the home environment that manages resource grants, but at the implementation level it is realised as an own service hosted on top by the basic support of the home environment.

If resources are granted to a service environment, the HEMS will create a *service environment service (SES)* that is also realised by a replicated decentralised adaptive service. To create such a service, the HEMS queries the home environment for generic peer factories and selects some of them to initiate the service fragments. An SES builds a stable access-point of a service environment in scope of a domain. The SES is conceptually a part of the service environment but at the implementation level realised as a separate service. Similar to the HEMS the SES copes with node reboots and node faults by querying the home environment for factories and the creation of new service fragments as needed. This way, it is ensured that a service environment reliably spans all supporting domains.

## 5.3 Service Environment

A service environment represents a scope of distribution for one or more decentralised, adaptive services. Usually, it is owned by a single organisation or community and managed by an individual called *service manager*. Such a manager can start, stop and configure services. Furthermore, she decides which resources offered by home environments are accepted and used. The main goal of the service environment is to support the seamless and easy distribution and management of decentralised, adaptive services. In most cases, a service environment is spread over more than one administrative domain as shown in Figure 5.2. One task of the service environment is to collect resource information offered by the supporting home environments. Another task is to manage the migration of services or service components, based on available resources, the needs of the services, and policies provided by the service manager. The migration of service components can be necessary for various reasons, like a peer shutdown, load balancing, or the growth or shrinkage of a service environment.

As earlier mentioned, a service environment owns an SES instance at every supporting domain. These SESs are all connected by binding the service environment that internally provides a JXTA peer group. In fact, not every single replica fragment of an SES instance binds the service environment, but only one. This replica fragment is determined by being the current coordinator of the framework internal group communication (see Section 3.3.1). If the coordinating replica fragment fails, the group communication framework automatically selects a new coordinator that will also bind the service environment. At the implementation level this can be realised by registering a handler at the group communication framework that is informed about membership changes. This way, there is always one replica of an SES that is connected to the service environment. As the connected replica fragment answers only queries from the service environment and no state changes are performed, object determinism is preserved. Queries might get lost due to node faults but JXTA does not provide a reliable broadcast communication in the first place. Thus, lost queries are not an issue, as they have to be reissued anyway if they are not satisfactorily answered.

If a new service component has to be created or needs to be migrated, the service queries the service environment via its factory finder interface for an appropriate location. The service environment forwards the request to all connected SESs instances. If a service environment owns enough free resources in scope of a domain, the local SES propagates the query in scope of its home environment to find nodes that provide sufficient resources and fulfil the system requirements of the service component. Furthermore, basic requirements have to be taken into account, like not to place replicas of the same service on the same node. This is addressed in more detail in Chapter 7.

The expansion or shrinkage of a service environment depends on the offered resources and trustworthiness of the resource providers. Each domain manager has the possibility to offer resources to a service environment by setting up an SES. The service manager can accept the offer and instruct the service environment to use the offered resources by letting the SES connect the service environment peer group. Such an approach can be implemented by using the JXTA membership service that has to be provided by every peer group.

It is obvious that there could be situations where the available resources are not sufficient or severe problems occur like a network partition. In these cases, where the service environment cannot autonomously solve the problem, the service environment notifies the service manager. The service manager can then fix the problem manually (e.g., accept resource offers by other domains).

## 5.4 Integrating Decentralised Adaptive Services

In EDAS, a decentralised, adaptive service generally matches a traditional service accessed by users. Such a service can be implemented by using one of the two proposed service models and is in general realised as a fragmented object. This object expands or shrinks in the scope spanned by the associated service environment depending on the service demands and for fault-tolerance reasons. Usually, every part of the object is mobile and can be migrated if necessary.

```
interface DAServiceMgtIfc{
    exception InvalidProperty{ Property invalid_property; };

    #pragma annotate(restricted)
    void start();

    #pragma annotate(restricted)
    void stop();

    #pragma annotate(restricted)
    void shutdown();

    #pragma annotate(restricted)
    void set_configuration(in Property prop_set) raises InvalidParams;

    #pragma annotate(restricted)
    Property get_configuration();
};
```

Fig. 5.3: Service management interface

Independently of its internal service model, a decentralised adaptive service possesses at least three different interfaces. There is a general management interface for the whole service. It offers methods for service configuration and life-cycle management enabling to start, stop and shutdown of a service. Figure 5.3 outlines a generic interface that can be extended accordingly to the demands a of specific service implementation, for example to provide additional methods for custom configuration. All methods of the interface are access-restricted and usually only accessible by a service manager. This is enforced via custom code generation triggered by method annotation as explained in Section 3.7.

Additionally, decentralised adaptive services provide a local management interface (see Figure 5.4). This interface allows the service platform to submit important system events to the service. Currently, these are only notifications about a scheduled system shutdown or reboot. The events include the actual shutdown time and in case of a maintenance reboot how long the node will be off-line. This gives a service the opportunity to decide whether to relocate an affected fragment or simply tolerate the down time, as all remaining services will be restarted on reboot. The third and last interface represents the actual service interface that is accessed by users.

```
valuetype RebootEvent : PlatformEvent {
    Timestamp shutdown_time;
    Timestamp duration;
};
interface DAServicePlatformMgtIfc{
    #pragma annotate(local)
    void receive(in PlatformEvent event);
};
```

Fig. 5.4: Platform management interface

## 5.5 System Overview

Taking all components together, a node usually hosts one virtual machine executing all management-related tasks and one or more virtual machine instances hosting services. Figure 5.5 outlines a basic scenario. The management virtual machine executes a home environment fragment that connects the node to all other nodes of the domain. It is accessed by a HEMS and one or more SES fragments. All of them are optional and might be placed on other nodes of the domain. The same applies for the service environment fragments that connect the SES fragments to the service environment. Apart from the home environment fragment the only mandatory component is the generic factory that is responsible for the creation of all optional management and service fragments. The figure also points out that the various components rely on the Aspectix support for CORBA, JXTA and group communication. It is even possible that a fragmented object relies on more than one communication system. The home environment fragment for example uses JXTA to connect other home environment fragments but also CORBA to inform services about system events via the home environment daemon, which is provided by every virtual machine hosting services. A service virtual machine is dedicated to execute one service or a service component in case of a decentralised adaptive service. Each service can access the service environment via a local service environment fragment.

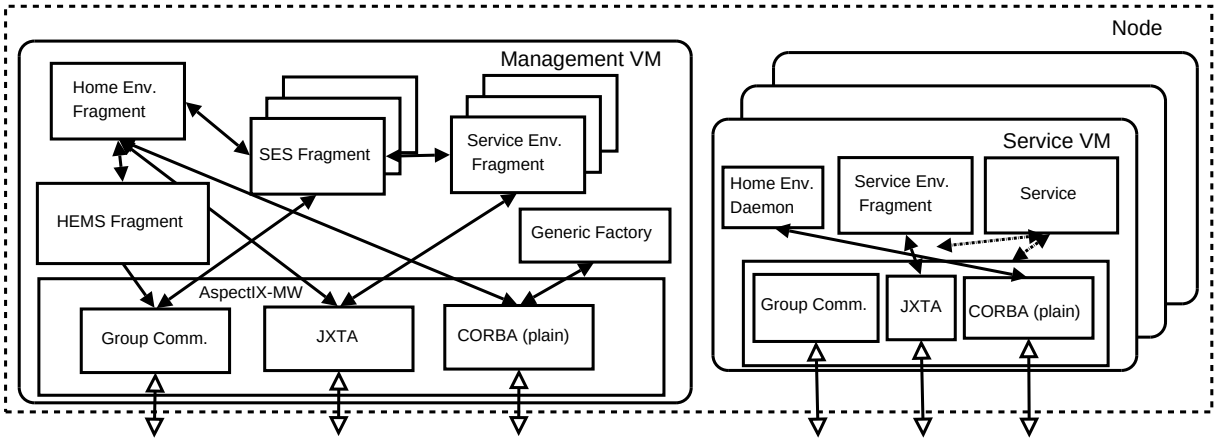



Fig. 5.5: EDAS system layout on a node

## 5.6 Summary

This chapter proposed the basic architecture of EDAS consisting of two main components. The home environment connects all nodes of an administrative domain and builds the communication back-plane for the service distribution in scope of a domain. The service environment provides a distributed execution platform for services belonging to one project or community. Both components including supporting services like the HEMS and SES can be implemented using the two proposed service models.

Summarising, EDAS builds the essential basis for distributed execution and management of decentralised adaptive services. Thus, EDAS answers the questions how the architecture of an infrastructure that is capable to support the stable provision of distributed and long-running services should look like. In the following chapter, this basic support is extended by services for dynamic selection and loading of platform-specific code, which is an essential prerequisite for the successful service distribution across multiple nodes. Finally, Chapter 7 extends EDAS by the necessary support for a resource-aware distribution of services.

# 6 Decentralised Dynamic Loading of Platform-Specific Code

When coping with decentralised adaptive services and fragmented objects in general, the object binding requires the instantiation of a custom and object-specific fragment implementation. Such a specific implementation usually cannot be dynamically generated using the interface of the service but at least parts of it have to be implemented by a service developer. This is the case for replicated decentralised adaptive services if their client-side programming support is used, e.g., a method is marked as `local` (see Section 3.7). For peer-to-peer based decentralised adaptive services, this issue is even more severe as there is no standard client-side fragment implementation. Thus, specific code is needed at the binding location to provide the demanded functionality, which cannot be automatically generated from an interface definition. In large distributed systems, consisting of numerous heterogeneous hosts, it is not feasible to pre-install all available implementations at every node. The reason is that most of the implementations are only needed on some nodes and these nodes are usually not known in advance. Sometimes, certain code modules might not be known at compile- or even at start-up-time of a service.

Consequently, support for the dynamic loading of code that is transparent to the binding entity is essentially needed for successfully providing and executing decentralised adaptive services in context of the EDAS infrastructure and as well important for clients to access a service.

In earlier work the author therefore proposed a dynamic loading service (DLS) that enables the dynamic loading of platform-specific code [KH03] that already addressed the issue of dynamic loading of code in context of heterogeneous environments. However, the DLS follows a classical client/server-based architecture and builds on a central component managing metadata of all know implementations and one or more services hosting the available implementations. One approach would have been to reimplement these services as replicated decentralised adaptive services. However, the code for a certain implementation is usually provided by single party and the metadata describing an implementation is fairly static. Thus, the support for a fault-tolerant and concurrent modification of data as provided by replicated decentralised services is not required. Accounting these facts and the aim to provide support for dynamic loading in a scalable manner a decentralised peer-to-peer based lookup, selection and loading process to support dynamic service distribution in scope of home environments and especially service environments is proposed.

The decentralised nature of the provided support and the ability to select an implementation that fits the requirements of an application and its executing environment distinguishes the proposed approach from all related approaches known to the author (see Section 6.1).

In the following, a brief overview about related approaches considering dynamic loading and deployment in homogeneous as well as in heterogeneous environments is given. As the problem of dynamic loading is a very basic one that occurs in most of all large distributed systems a general decentralised approach to dynamically search, select and load platform-specific code is described. Starting from this point, the facilities of the JXTA middleware for service lookup and integration are discussed and valued. Next, the general approach is mapped to JXTA by a prototype implementation of a platform-independent peer-to-peer based loading service. Then it is described how this support is integrated into EDAS. As a proof of concept the example application of Chapter 4 is extended by dynamic code loading.

## 6.1 State of the Art

As the execution of any software-based service or application relies on some form of code (e.g., machine dependent or interpreted code) its availability at a target system and how this is achieved is a very general but still important issue in distributed systems. In the most basic scenario code is copied on a node and then the application can be executed. This is further on considered as *static deployment* and requires an a priori knowledge where the application will be executed. If this information is not available and the application targets large distributed systems support for *dynamic loading* also called *dynamic deployment* is needed. Here, it is assumed that a service is available at a target site that is capable to load code on demand as an application is started or an object arrives at a system. In the latter case the service dynamically loads the code at the very beginning of the state deserialisation of the object. In the following related work mainly targeting dynamic loading is considered as motivated by the introduction this is essentially needed in context of EDAS.

The most common form of software deployment are package management systems. A simple system is the Red Hat Package Manager (RPM) [Bai97] used on Linux platforms. It allows the installing, updating, and verifying of software packages. To do so a package provider has to describe the software and its dependencies on other packages in a specific format. During the build process of a package additional parameters such as the host operating system and the machine architecture are automatically detected and noted in the package. There is only very limited support to specify additional custom system requirements like an operating system version or the address length of a machine architecture. However, the RPM system offers the possibility to specify tests as shell scripts, which are executed before the actual install process. This way custom requirements can be evaluated and if not fulfilled the installation process is aborted. To install a package a mounted repository is required. Thus, a centralised deployment is supported. Besides, all package-managing systems primarily target the static deployment of applications.

As Java is used for many distributed applications because of its platform independence, various projects target the support for dynamic loading of Java code. In fact the Java runtime environment already supports the dynamic loading of code by providing basic facilities for remote class loading offered by `URLClassLoader` a special class loader. It is capable to load classes from a remote location using the hypertext transfer protocol (http). In addition the Java runtime environment offers its own dynamic deployment mechanism named

Java Web Start [SM05] to provide whole java-based applications. This software deployment system uses the Java Network Launching Protocol and describes the code and the requirements of a Java application in a custom XML format. This results in applications that can be installed over the net via a special Java Web Start client (even system-dependent native libraries can be installed on demand). However, both approaches are highly Java-dependent and limited to install and in case of Java Web Start to update java-based applications. Furthermore, they feature a client/server based loading model.

The OSGi service platform [All05] defines an open run-time environment, enabling dynamic service integration. For the representation of a service's functionality, the concept of an OSGi *bundle* is defined. A special characteristic of such a bundle is the possibility to be able to dynamically add and remove it from the run-time environment. Compared to the similarly conceived proceeding in this work, a bundle offers extended possibilities, in order to formulate dependencies on other services. However, the OSGi approach misses sophisticated mechanisms to describe, remotely lookup and select code as outlined in this work. Further on, OSGi primarily targets code loading and sharing for the Java programming language, whereas the presented approach can be applied to other programming languages as well and targets the distributed lookup and the selection among various implementations.

Paal et al. [PKF05] proposed a distributed code-loading infrastructure based on multiple application repositories that can be dynamically queried by a custom application loader. In contrast to the presented approach, this system offers fine-grained code loading based on *class collections*, which are represented by class subsets of a Java archive. However, the system is limited to the Java programming language and application repositories have to be preconfigured at initial deployment time for enabling code loading. In later work Paal et al. [PKF06] proposed extended deployment support for components that targets the dynamic assembly of applications. This is beyond the scope of the proposed concepts but both approaches can be combined to enable the dynamic deployment of component-based software in heterogeneous environments.

In [KH03] the author presented the DLS, a CORBA service for the dynamic loading of program code. Similarly to the realised loading service of this thesis, the DLS permits to load remote code with consideration of the current run-time environment. However, the DLS follows the client/server paradigm and uses dedicated servers to host the program code portions and to offer specific information about available code. This is contrary to the code loading support proposed by this thesis that uses a peer-to-peer-network for both mentioned functionalities.

A peer-to-peer based architecture for remote loading of Java classes is described in [PC03]. This approach shows an alternative way to the standard Java class loader mechanism and is exemplarily realised using JXTA. Compared to the proposed solution, it lacks flexibility to describe and to search for suitable program code. Thus, the architecture neither permits a representation of loadable code with the JXTA concepts of module advertisements, nor it offers the generic possibility for the free choice of a transfer protocol.

None of the related approaches supports an application-specific selection of code in truly heterogeneous environments and offers a decentralised lookup selection and loading process.

## 6.2 Requirements and Properties for Implementation Selection

In order to bind a decentralised adaptive service or to extend a service by locally not available functionalities the dynamic loading of platform-dependent code is required. It is expected that every service or functionality might be provided by various implementations that offer the same function set but having different requirements and properties. A generic and decentralised selection process should choose the best-fitting one for a certain environment.

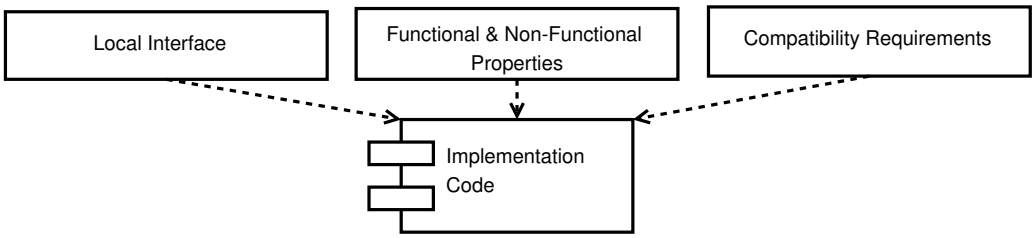



Fig. 6.1: An approach towards a generic code classification

Four categories of properties and requirements have to be fulfilled or at least be taken into account in order to select an appropriate implementation (see Figure 6.1).

**Local interface.** If an application requires a new locally not available functionality potential candidates can be identified by their interface, as an interface determines how the application can deal with the implemented functionality at the programming layer. Thereby, an interface should be defined in a generic interface description language to be platform independent. In context of this work, CORBA IDL has been used but other languages like the Web Services Description Language (WSDL) might also be suitable.

**Functional properties.** Functional properties express additional functional aspects going beyond the bare provision of an interface. For example, there might be various implementations of a network-dependent functionality, e.g., a remote stub, providing the same local interface but targeting different middleware infrastructures. For a successfully selection of the right implementation variant this information has to be available otherwise the stub implementation might not be able to connect to server object. In general it is hard to standardise all kinds of functional properties, however, this is useful for a generic selection process. To solve this issue, an infrastructure for dynamic loading should address well-known functional properties and delegate the evaluation of seldom and unknown properties to the requesting application. In scope of this work, functional properties concerning network capabilities are processed at the infrastructure level, but the processing of all further properties has to be handled by the application.

**Non-functional properties.** Implementations providing the same functionality might possess different non-functional properties that are essential to provide the requested behaviour for the calling application. Examples for such properties are timing behaviour, resource consumption and in general quality-of-service properties. Like functional properties, these additional qualities are hard to be generally standardised and evaluated at the infrastructure level and therefore usually have to be evaluated and processed by the application.

**Compatibility requirements.** As a fourth category, the concrete compatibility requirements that a certain implementation should fulfil have to be considered. This deals with a precise programming language or tangible execution environment issues. Such an approach considers the fact that some kind of functionality could be implemented in different programming languages, e.g., Java or C++, and could be optimised for a special run-time environment, e.g., Linux or Windows. Thus, implementations can realise exactly the same functionality, but require different systems. Compatibility requirements can be evaluated at the infrastructure level as there is a limited set of properties, outlined in detail in [KH03] that determines if an implementation is executable in the context of a requesting application or not. As the dynamic integration of a functionality usually requires some kind of initialisation and integrated functionality might be managed and controlled at the middleware level, compatibility requirements might additionally comprise platform-dependent interfaces.

## 6.3 Basic Infrastructure

The proposed infrastructure is composed of three components named *code provider*, *dynamic loader* and a *decentralised implementation repository*. A dynamic loader provides an interface to the application and the middleware in order to request a certain locally unavailable functionality. It searches, selects and integrates a suitable implementation into the address space of the requesting application. The lookup process of an appropriate implementation is thereby supported by the decentralised implementation repository that manages the information about all available implementations and possibly the implementations itself. Thereby, the repository is accessed by multiple code providers offering implementations and providing metadata descriptions. The scope of the implementation repository might be an enterprise, a multi-organisation-spanning grid or even a global-scale testbed like the PlanetLab [PACR03]. In context of EDAS, the scope of an implementation repository usually matches a service environment or a home environment but is not limited to that.

## 6.4 Data Structures of the Implementation Repository

Starting from the set of properties and requirements outlined in Section 6.2, it is possible to select the best fitting functionality for a requesting application. To provide a decentralised and scalable model for lookup, selection and loading of functionalities, all information about available implementations are collected and published as metadata descriptions in scope of the implementation repository. This enables an implementation selection without actually having to load the binary data of possible candidates. In fact, even the metadata description is split in several entities thereby omitting duplicated information and improving extensibility. Another benefit of such an approach is the opportunity for multiple independent entities to provide implementations and their associated metadata without having to coordinate modifying operations to the data structures. Four different metadata entities have been identified, along the lines of the requirement categories that have to be taken into account for selecting the best fitting implementation (see Section 6.2):

- Interface description
- Module description
- Extended functional description
- Implementation description

Each of these metadata entities is separately published in scope of the implementation repository as an own description document. The *interface description* document contains the fully-qualified name of the interface and the IDL interface itself. An interface might reference other interfaces and complex data types (e.g., named types, or exceptions). These are referenced by their fully-qualified names. This enables a dynamic lookup of referenced interfaces and data types by name. To cover all complex data types of an IDL module, they are combined and published in a *module description* an optional auxiliary metadata description. Interfaces defined inside the module are only referenced by their names. In the context of the dynamic loading process, implementations of a module description are not further discussed as their code should be available at the requesting platform or included in the code archive of the loaded implementation. Together, module and interface descriptions allow a complete representation of IDL modules. This offers the opportunity to use these descriptions to provide a decentralised interface repository.

The *extended functional description* specifies all functional and non-functional implementation-independent properties. Implementation-independent properties can be considered as attributes that are provided by various implementations and therefore should be used to select a group of equal implementations providing the same interface, possibly targeting different platforms. As mentioned earlier, it is complex to identify a concrete set of functional and non-functional properties that applies for a major number of applications. Instead, an implementation repository and an associated dynamic loader should provide a flexible interface that offers applications the possibility to introduce custom code for evaluating these properties. However, in scope of this thesis, the functional properties of a remote-dependent functionality have been standardised, given by the network protocol. A well-known protocol identifier like *http* for the hypertext transfer protocol and an optional link referencing its specification describes the protocol.

As the fourth metadata document, the *implementation description* describes a concrete implementation and its compatibility requirements. The implementation description includes a reference to the code and a description of the initially accessed implementation element. For object-oriented language like Java or C++ this would usually be a class name referencing some kind of factory class providing a default constructor.

## 6.5 Basic Workflow of Publication, Selection and Loading of Code

First, a code provider aiming at publishing several implementations of a functionality has to generate a set of appropriate metadata documents describing the offered code. As all implementations are identified by the same interface, one interface description is created. If there are multiple implementations of a functionality targeting not only different system

platforms but also providing support different network protocols, as this might be the case for a remote stub or a fragment of a fragmented object (see Section 2), a separate extended functional description has to be generated for every supported protocol. If omitted the right implementation cannot be unambiguously identified. To support the lookup process the extended functional descriptions reference the associated interface description. Finally, for every implementation, an implementation description has to be generated, describing the compatibility requirements and referencing the extended functional description and the concrete implementation. All these metadata documents are published via the decentralised implementation repository. Finally, the concrete implementation code has to be published in scope of the implementation repository or served via a separate service as done by the prototype implementation due to technical limitations of the JXTA platform.

If an application or a middleware infrastructure needs a locally unavailable functionality, it contacts the local dynamic loader by passing the fully-qualified name to identify the interface of the requested functionality. Additionally, it passes an optional handler for custom and maybe application- or domain-specific evaluation of extended functional requirements. The dynamic loader, first requests the implementation repository to lookup the interface description and, if not available, passes an exception to the calling application. On success, the repository is queried for all extended functional descriptions supporting the requested interface. If provided by the application, the results are passed to a handler, which returns an ordered list of suitable extended functional descriptions starting with the best-fitting one. Then, the dynamic loader asks the repository for each of the candidates for implementation descriptions and starts to evaluate them one by one. Depending on a policy, either the first fulfilled implementation description is taken or all are considered and the best-fitting one is selected. The best fitting implementation is considered as the one that fulfils the most specific compatibility requirements and, therefore, should have been explicitly implemented for the requesting environment. After having selected an appropriate implementation description, the code has to be loaded.

## 6.6 Dynamic Lookup and Loading of Services in JXTA

Section 4.2.5 gave already a brief overview about the service and module concept of JXTA. In principle it already offers the dynamic extension and loading of JXTA applications and services. However it has some conceptual shortcomings that prevent a direct mapping of the proposed generic lookup and loading process. This section targets a more detailed description of the lookup and loading mechanisms of JXTA and outlines its weaknesses.

### 6.6.1 Managing Lookup and Loading of Services via Advertisements

To enable an efficient way to discover modules, the definition of a module is divided into three types of advertisements:

- Module Class Advertisements
- Module Specification Advertisements

- Module Implementation Advertisements

As JXTA claims to be both language- and platform-neutral, the framework has to support multiple implementations of a given module, e.g., a module could be implemented in Java, C++ or any other programming language. In order to distinguish between several module implementations, each one is represented by a *module implementation advertisement*. Such an advertisement specifies all implementation-specific details, including the location of the code.

To cope with the common fact that software modules change over time due to software evolution, the concept of *module specification advertisements* is established to handle different versions of a module. As each version is supposed to have multiple implementations, the corresponding specification and implementation advertisements are linked loosely. Thus, a module implementation advertisement realises a specific implementation of a module specification. Accordingly, a module specification advertisement uniquely identifies a set of protocol-compatible modules. Additionally, a specification advertisement provides the necessary information to uniquely identify and to contact a service.

A *module class advertisement* is used in order to provide a higher view for referring to modules that provide a particular class of functionality, independent from a certain specification or implementation. Thus, a module class advertisement only exists to announce the pure existence of a unique module class (e.g. video on demand service). To spawn a module hierarchy, there exists a relationship between a module class and a module specification advertisement, as multiple module specification advertisements can relate to one module class advertisement (e.g. there might be several vendor-specific protocols for video on demand).

A module is initialised for a certain peer group. In return, the peer group could be used to retrieve its service. The latter is basically possible for the JXTA Java binding by the help of a special class loader, which offers additional functionality. The needed class files are defined with corresponding module implementation advertisements whereas the class loading mechanism should determine the suitability of the code using the embedded compatibility statements.

When a peer joins a certain peer group, an instance of all the services associated with the peer group should be dynamically created on this peer. Here, not all services need to be resident on the joining peer. It is intended that a peer can dynamically load new services using the information provided by the corresponding peer group advertisement referencing all the module specification advertisements of the peer group services.

### 6.6.2 Shortcomings and Weaknesses

Based on a set of protocol specifications, JXTA in general provides a network-centric and more or less network-directed support for peer-to-peer applications. This applies also for the dynamic lookup and integration of services.

Class advertisements just announce the availability of a general category of functionality. This might give developers a hint what to expect from a certain module specification and supports the selection process at a very high level. However, for an automated module

selection at application or middleware level, additional conventions have to be established. These conventions have to determine how to interact with the module, as the module specification targets only the specification of the network-directed interface of a module, namely the supported protocol. Therefore, the Java reference implementation makes implicit assumptions that a module implementation provides a certain interface that is neither specified by the protocol specification nor declared by advertisements. This interface enables to basically start and stop a module at the platform level. Furthermore, there is no support to determine and specify the interface of a module offered to higher layers like an application, making it hard to provide multiple implementations for the same platform but providing different properties. Thus, in the worst case a module can be loaded and executed by the platform but the application expects a different module interface and therefore cannot interact with it.

Last but not least, implementation specifications offer the opportunity to provide compatibility information but they are not standardised. The consequence might be that every JXTA implementation specifies its own format and parameters, which prevents the usage of a module implementation in context of different JXTA platform implementations.

To sum it up, the JXTA support for dynamic loading and integration of services leads to platform-specific implementations and does not support the dynamic loading of arbitrary code.

## 6.7 Decentralised Dynamic Loading Based on JXTA

As outlined in the previous section, JXTA already offers a set of advertisements enabling the publication and lookup of module and service descriptions, which supports the dynamic loading of code. Although this approach seems to be generic and flexible, it owns certain weaknesses and shortcomings so it cannot be directly applied as a generic and platform-independent infrastructure for dynamic loading. Therefore, the three JXTA service advertisements have been extended conforming to the proposal in Section 6.4 and to provide an own loading infrastructure on top of JXTA, offering extended support for dynamic loading.

### 6.7.1 Extended Advertisements

The *class advertisement* can be mapped in a straightforward way to an interface description document. The *name* field of the advertisement specifies the fully-qualified name of the most-derived interface supported by the described functionality. The *description* field of the advertisement is used to represent the whole interface. As the name field of the class advertisement is indexed in the JXTA network, an interface can be searched by its name.

The *module specification advertisement* is mapped to an extended functional description. This requires no fundamental changes; instead the protocol specification is considered as a functional property declaring if and how a functionality is network-dependent. Additional functional and non-functional requirements are encoded into the *description* field. If the specified functionality is a JXTA service or module, there is a pipe advertisement; otherwise the dependent field is left open.

Finally, the *module implementation advertisement* is extended with standardised compatibility requirements as defined in previous work [KH03]. These are stored in the *comp* field. In addition, platform-dependent interfaces have been added to the compatibility requirements allowing to explicitly specify how to integrate a certain functionality at the platform level. In contrast to the Java JXTA reference implementation that only allows loading a JAR-file from a web server specified by a *puri* field, extended facilities to reference and transfer code archives are provided. This is achieved by embedding a module specification advertisement in the *puri* element to describe a code provider. This flexible way enables the integration of arbitrary services for the dynamic code transfer as no transfer protocol is dictated. Furthermore it offers an additional recursive way to dynamically download code, as a requesting peer is able to fetch the code transfer handler that is specified by the *code* element dynamically over the peer-to-peer network. A code transfer handler should be offered via the HTTP-based code transfer support provided by JXTA, or our reference implementation of a basic JXTA transfer service that is described in Section 6.7.3. Thus, only one level of indirection is assumed.

### 6.7.2 Decentralised Implementation Repository

Section 4.2.1 introduced the concept of a peer group as a mechanism to group users or applications having similar interest. In context of the further on detailed prototype implementation a dedicated well-known peer group, called *Code Peer Group* is used as the starting point to publish and discover implementations. In context of EDAS this group usually corresponds to the peer group spanned by a service environment. A Code Provider, as described in detail in the following subsection, publishes all advertisements related to offered implementations in scope of this peer group.

Unfortunately, JXTA ties a module specification advertisement to a pipe that is bound to a certain peer group. The consequence is that this peer group is also used as the group to execute all the services. If this is not feasible an application creating a service instance has to lookup the dependent module specification advertisements, modify them by providing a group specific pipe advertisement, and finally republish them in scope of the Code Peer Group. During the lookup process the dynamic loader has to know the peer group of the requesting application and selects an appropriate advertisement. Both approaches can be used concurrently.

### 6.7.3 Code Provider

As outlined in the previous section, the JXTA reference implementation provides only restricted mechanisms for code transfer and sharing. Therefore, an own code provider service has been implemented, enabling code sharing and transfer via the JXTA network.

Before a concrete implementation and its dependent code archive can be published, the associated extended advertisements have to be generated, if not already available. The `CodeShareService` class offers the core functionality to publish and share a concrete implementation. If an implementation is published via `shareFile()`, the service contacts two other components, named `CodeBase` and `CodeServer`, as displayed in Figure 6.2. A

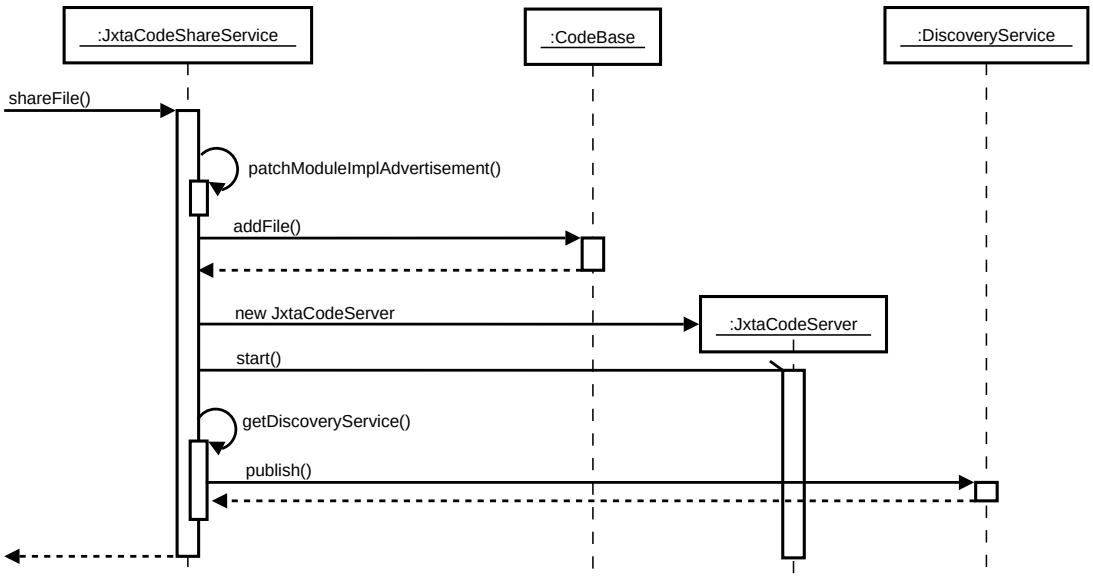



Fig. 6.2: Sequence chart code sharing

code archive together with the three advertisements is handed over to the class via the `shareFile()` method. If an archive should not be served any longer this can be achieved by calling `unshareFile()` passing only the module implementation advertisement as it uniquely identifies an archive.

First, the `CodeShareService` modifies the module specification advertisement to include its pipe advertisement for the code transfer, then it passes the archive to the `CodeBase`. The `CodeBase` administrates the offered code archives through persistently maintained information, which enables a consistent restart. Then, an instance of the autonomously working `JxtaCodeServer` is created. It provides a multi-threaded server, responsible for the concrete file transfer via a simple JXTA-based protocol. In the last step the advertisements are published via the JXTA discovery service.

Finally, the `CodeShareService` can be run in scope of a service environment like any other service but is implemented as traditional non-replicated service. To offer a fault-tolerant code provision service it should be enhanced to a replicated decentralised adaptive service that uses JXTA. However this can be implemented in a straight forward way. The `shareFile` operation is thereby propagated to all replicas that independently publish the code and the associated advertisements.

### 6.7.4 Dynamic Loader

The dynamic loader builds the core of the prototype implementation. Figure 6.3 illustrates the collaboration between its important components that are all local to the client. The class `JxtaCodeHandler` takes up a central role during the whole process of dynamic loading.

In order to find a unique mapping between a demanded interface and the requested network and platform requirements the dynamic loader only expects a module class id or a fully qualified interface name to determine the basic interface, and additionally to that the mentioned information of a module specification advertisement to determine the

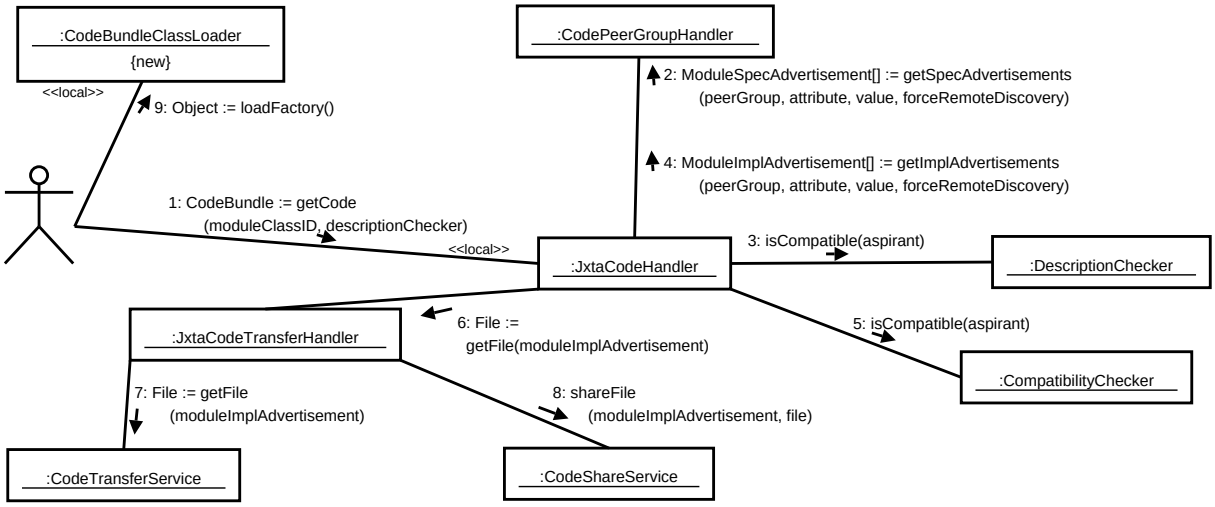



Fig. 6.3: Collaboration of central system components

required network functionality. Chapter 4 outlines that sometimes the module specification advertisement is already preselected as it is part of the reference, then some of the following steps can be omitted.

Through the method `getCode()` the `JxtaCodeHandler` offers a way to search for a module specification advertisement providing certain key identifiers like a module class id, name and version of a specification advertisement or a generic description within the *desc* element. The latter is achieved by passing an object that implements the `DescriptionChecker` interface in order to perform a validity test for the concrete use case (1). To select an implementation, the loader service uses the module class id to discover corresponding module specification advertisements within the code peer group (2). With the passed module specification advertisement and the passed generic `DescriptionChecker` provided by the requesting application the discovered specification advertisements can be filtered for a suitable one (3). Here, it might be necessary to perform multiple requests to the JXTA network. With the extracted module specification id, a search for corresponding implementation advertisements within the certain peer group can start (4). As next step, the dynamic loader analyses the received implementation advertisements with the help of requirements provided by the local execution environment (5). More precisely, a distinct advertisement is chosen by using an object that implements the `CompatibilityChecker` interface, which expects the handling over of a compatibility document in order to validate the suitability for the current execution environment. If a suited module implementation advertisement could be found, `JxtaCodeHandler` is able to initiate the code transfer, if an appropriate transfer handler is locally available (6). Otherwise, a suited transfer handler could be fetched recursively. This process is transparently executed for the `JxtaCodeHandler` by the `JxtaCodeTransferHandler` (7). If the code transfer fails for a certain module implementation advertisement, another code provider could be chosen. The class `JxtaCodeTransferHandler` encapsulates the whole transfer process by offering

a method `getFile()` that only takes a module implementation advertisement as parameter. Exemplarily, a code transfer service supporting file transfer using a JXTA network is realised. As the amount of transferred data could be arbitrary large, but a direct usage of a JXTA Pipe allows only messages of a maximal size of 64 KB, there is a need for segmentation. For transportation the `JxtaSocket` and `JxtaServerSocket` from the JXTA reference implementation are used which support message segmentation and a bidirectional and reliable communication between two peers. Thereby, a file transfer in binary mode is obligatory. On success, it can again be offered by the requesting peer to support the scaling of the whole peer-to-peer system (8). Additionally, the persistent caching of the code portion avoids remote transfers resulting from future requests. As last step, an object-specific factory is used to dynamically integrated the fetched code bundle into the the running system (9).

## 6.8 Integration of Dynamic Loading Support into Aspectix and EDAS

The dynamic loader has to be provided at every node inside every process that requires the dynamic loading of code. It can be accessed in two ways, either as an internal service during the binding process of a fragmented object, or explicitly as an initial CORBA service provided by the local ORB instance. In the first case the dynamic loader is accessed during the binding of a reference by the responsible profile manager. This profile manager takes the full-quallified interface name from the reference and forwards this information to the dynamic loader that is then responsible to load an application according to the process outlined in the previous section. In the second case an application demands for a locally not available functionality or a fragment implementation targets its exchange against a better-suited one like outlined in the following section. Here the ORB is queried for the dynamic loader via `resolve_initial_references()` the standard method to ask for well known services in context of CORBA.

Up to now the described support is fairly independent of the EDAS infrastructures and can be used to provide dynamic loading of code in context of Aspectix and for arbitrary distributed applications. For EDAS it is assumed that every home environment and more important every service environment has its own code repository. This is necessary as service implementations might be specific for a certain project. To provide such a support the peer group that is spanned by every service environment is also used as a code peer group for the associated code repository. As soon a service environment is bound the local dynamic loader can query the associated peer group for specific implementations.

## 6.9 Case Study: Extended Multi-User Chat Application

To clarify how to use the dynamic loading infrastructure, the JXTA based multi-user chat application described in Section 4.5 is extended by dynamic code loading in two ways: First, the initial fragment implementation is loaded at binding time by the responsible JXTA

profile manager. Second, a fragment implementation is made exchangeable against better-suited ones which have to be dynamically loaded as they are locally unavailable.

In addition to the specified development and deployment steps the described extended advertisements have to be generated. Advertisements and a code archive holding the basic service code and the initial fragment implementation are published via an instance of the Code Provider service. Finally, an object reference has to be generated, including the JXTA profile. This time the reference is not only used to connect the service but also to select and load the code. At middleware level, the code is dynamically loaded as described in Section 6.7.4 and a fragment is instantiated and initialised. After that point, the fragment can exchange messages with all other available fragments via a propagate pipe.

```
private void exchangeFragmentImpl(String desc) {
 CodeBundle codeBundle = null;
 try {
  codeBundle = foxtaHandler.getFoxtaCodeHandler().getCode(moduleClassID,
      new DescriptionCheckerImpl(desc));
 } catch (NoCodeAvailableException ncae) {
  System.out.println(ncae.getMessage());
 }
 CodeBundleClassLoader classLoader = null;
 try {
  classLoader = new CodeBundleClassLoader(codeBundle);
 }catch (MalformedURLException e) {
  System.out.println(e.getMessage());
 }
 Fragment fragmentImpl = classLoader.loadFragment(view, foxtaHandler,
     pipeAdvertisement);
 view.setFragImpl(fragmentImpl);
}
```

Fig. 6.4: Fragment implementation exchange (Java)

To outline further benefits of the support for dynamic loading and service integration based on the fragmented object model, not only a fragment implementation providing the basic multi-user chat application has been implemented but also two further fragment implementations. One provides a simple logging facility that gives a user the opportunity to record all user actions and messages. Another one provides a simple bot that can substitute a user as she is away or act as simple knowledge base of a project or interest group. The basic feature of the bot is to memorise answers to simple question (e.g., "Q:Where is Hal?, A: In town").

All three fragment implementations provide the same interface but possess different extended functional descriptions, i.e., different specification advertisements. By default, the plain fragment providing only the basic chat functionality is loaded and instantiated as its specification advertisement is included in the object reference. A user can trigger a frag-

ment replacement via the user interface by submitting special control messages processed by the local fragment. The same applies for the extended facilities of the logging and bot fragment implementation.

Figure 6.4 details the necessary steps to exchange a fragment implementation. Some of the operations might be executed by different components and have been combined in the fragment method for a better understanding. First, the fragment's local handler class is invoked requesting a new implementation handing offer a custom object to select an appropriate implementation. If a suitable implementation could be determined a `CodeBundle` object is returned. Its reference is passed to our custom class loader instance. This loader possesses not only a method for generic factory creation but also a specialised method to create and initialise a fragment implementation. Finally, the implementation is exchanged with the help of the fragment infrastructure provided by the Aspectix middleware.

## 6.10 Summary

This chapter presented a generic and decentralised approach to dynamically search, select, load and integrate platform-specific code. Dynamic loading of code is an essential prerequisite when binding fragmented objects and decentralised adaptive services as both require specific code, which is usually not deployed at the binding system. This is especially true in the context of EDAS. Concluding, the presented support for dynamic loading gives the answer of this thesis on how applications could be deployed in a novel global-scale infrastructure.

Security issues are beyond the scope of the proposed infrastructure. Dynamic loading of code always involves security considerations. However, standard security mechanisms as code signing and a public-key infrastructure can be used for securing the presented peer-to-peer based dynamic loading service.

Even though the prototype supports the precise selection of platform-specific code, currently it is assumed that a concrete implementation is more or less self-contained. This means that either necessary libraries are available at the target platform, or included in the dynamically loaded code archive. It would be worthwhile to provide support for interdependencies between different implementations. These should be resolved at loading time, resulting in the dynamic loading of dependent implementations. Along the lines, investigating the OSGi platform [All05] seems promising as it provides sophisticated code-sharing mechanisms at the Java virtual machine level, but misses support for remote selection and loading of code.

# 7 Resource Management

One of the four questions that have been aimed to answer by this thesis is: *How to manage the resources for long-running services?* So far, decentralised adaptive services are characterised as services that provide fault tolerance using decentralisation and are able to adapt their distribution according to service and environment demands. This general description was realised by two concrete service models: replicated and peer-to-peer based decentralised adaptive services. Both service models provide support for dynamic distribution, e.g., by the support of parallel state transfer in case of replicated decentralised adaptive services. Extending this basis, EDAS offers the infrastructure to enable dynamic distribution and management of services across multiple domains. This infrastructure support is invigorated and made practical usable by enabling dynamic loading of platform-specific code. Still missing at this point is the necessary information to adapt the distribution of decentralised adaptive services according to resources. However, sufficient resources are essentially needed to successfully execute services. On the other side the execution environment has vital interests (e.g., monetary reasons) to control and restrict resource usage to be able to dedicate resources to projects and users as needed. Thus, narrowing the question from the beginning, this chapter aims at answering: *How to provide support for resource management of decentralised adaptive services in the context of EDAS?*

Consequently, this thesis targets to provide a resource management approach and infrastructure support that uses the proposed architecture of Chapter 5 to efficiently distribute decentralised adaptive services while keeping certain resource constraints. This demands for five tasks: discovery, monitoring, accounting, assignment and finally usage-restriction of resources. Monitoring and accounting are addressed by integrating existing resource management support of the Sun JVM Version 1.5 into the EDAS infrastructure. The two main contributions of this chapter focus on the issues of discovery, assignment and the restriction of resource usage. First, a decentralised and fault-tolerant protocol to enforce global resource quotas is presented. It enables a scalable and efficient management of global quotas across a potentially large set of distributed nodes. For EDAS it provides the basis to handle and restrict the resource usage of consumable resources, such as transmission volume. Second, a decentralised placement algorithm is proposed that utilises the EDAS infrastructure and enables a resource-aware distribution of decentralised adaptive services. In the context of EDAS it enables the resource management and usage restriction of CPU, memory, disk space and network bandwidth.

The chapter starts with a brief introduction about resources and discusses at which level their usage should be managed. Section 7.2 details how to monitor resource usage in EDAS. Next, it is described how resource usage can be restricted at the level of service instances and service environments using a decentralised quota management protocol. Finally, Section 7.4 proposes an approach to dynamically distribute decentralised adaptive services.

## 7.1 Resources and Resource Limits

Generally, one can distinguish between *system* and *conceptual resources* [CGSM03]. Typical system resources are: memory, disk space, CPU, and network. Conceptual resources are for example software licences or the number of open files. Furthermore, system but also conceptual resources can be subdivided in *consumable* and *non-consumable*. For example, CPU rate is non-consumable, but CPU cycles are consumable. The same distinction can be made for network with network bandwidth and transfer volume. Karmon et al. [KLLS05] refer to these resource classes as *refundable resources* and *non-refundable resources*, as once consumed fractions of a resource depending on their type can be refunded to an application or not, after usage.

To manage and restrict resource usage, EDAS introduces resource management at three different stages: service, node and service-environment level.

At the node level the resource usage of all kinds of system and conceptual resources is monitored. The entity of monitoring is a service or a fragment for a decentralised adaptive service. Every service has *local limits* that restrict the resource usage of resources in the context of its hosting node. If a service runs the risk at exceeding a local limit, the home environment starts, depending on the resource type, a resource management process (see Section 7.3 for consumable resources) and if this does not help, it notifies the service. Once informed about an imminent limit overrun, a service can react in two ways, either it reduces its resource consumption or it migrates to another node that provides enough resources (see Section 7.4).

Considering non-consumable resources each node has *node limits* that restrict the overall usage of certain resources at the level of a node. A node limit must never exceed the physical resource (e.g., disk space) of a node and might be much smaller, e.g., for supporting local users with the remaining resources. The sum of all local limits of all services at a node must never exceed the node limit of a resource. Therefore observing and controlling the local limits will keep the node limits and prevents overload.

At the home environment level, the global resource usage of a service environment and all its associated services is restricted by *global limits*. These limits determine the maximum resource usage of a service environment in scope of a home environment. The sum of all local limits on all nodes of the institution for a service environment therefore never exceeds its global limit. Observing and controlling global limits can have monetary (e.g., a project has paid for a certain amount of arbitrarily distributed resources) or security reasons (e.g., by enforcing global limits a program cannot accidentally allocate too many resources).

## 7.2 Resource Monitoring and Accounting Infrastructure

To enable resource management and enforce local and global resource limits, a monitoring and accounting infrastructure on top of the resource monitoring features provided by the Sun JVM 1.5 was implemented.

### 7.2.1 Basic Support for Resource Monitoring

The Sun JVM 1.5 already offers monitoring capabilities for CPU and memory usage at the granularity of a JVM instance. Each resource monitor is realised as a Java bean that can be remotely accessed via the Java Management eXtensions (JMX) [Mic02]. In addition to this support, a monitor bean for network bandwidth and transmission volume, given by a special `Socket` class, was implemented. It basically supports the accounting of all send and receive operations of a socket instance. To enable the monitoring the extended `Socket` class has to be used by services instead of the original socket implementation provided by the Java class library. To cover all essential physical resources, a monitoring bean for disk space can be provided along the lines of the network-monitoring bean. JMX offers the option to implement additional beans that can monitor conceptual resources as well but this was not exploited.

However, resource monitoring is not the main goal of the project as other research projects have already developed extended JVMs [CvE98, SBB+01, GLS02] or modified bytecode [BHV01, HB04] to do resource monitoring of Java applications. Instead, it was aimed to provide a stable basis for implementing and evaluating the proposed concepts for decentralised adaptive resource management, as addressed by the following sections.

### 7.2.2 Node Infrastructure for Resource Monitoring

Figure 7.1 shows the architecture of the prototype that extends the basic EDAS architecture (see Section 5.5) by resource monitoring, accounting and management facilities. Each service is executed in its own JVM instance and monitored by an instance of the *resource manager* executed in a dedicated management JVM instance. This is achieved by observing resource usage of the service JVMs by JMX beans for CPU, memory and network. These beans are accessed via RMI by a *sensor/actor pair* of the resource manager component. A sensor periodically collects resource-usage information provided by the management beans and forwards them to the resource manager. An actor configures the management beans and notifies services if a limit is reached. The resource manager hosts information about all monitored services and provides this information to the local home environment fragment. The resource limits are described as simple policy rules that are managed by an internal component of the resource manager, the *policy manager*. A resource policy does not only define the current resource limits but also when to notify the service about an upcoming limit overstepping.

The presented infrastructure enables resource monitoring and restriction at the node and the service level. Furthermore, it offers a good basis for distributed resource management at the level of home environment and service environment. However, resource limits cannot be enforced for all types of resources, as there is no support for restricting CPU usage. Thus, if a service oversteps its CPU limit it either has to be terminated and migrated to another node or reduce its CPU consumption for example in a cooperative way (e.g., by calling `wait()`). A possible solution to this problem offers the Multi-User Virtual Machine (MVM) [CDT03] that has recently been released by the Barcelona project. The MVM is a special virtual machine that supports the execution of multiple applications in scope of one JVM instance

and provides advanced resource management facilities based on *isolates* [Cza00, CHS+04]. However, this option has so far not been exploited.

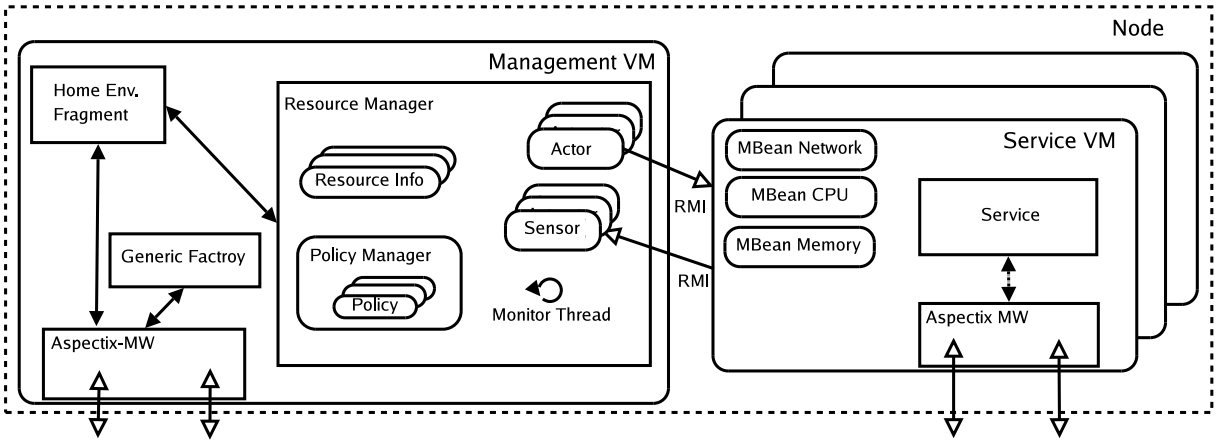



Fig. 7.1: Resource accounting and monitoring infrastructure

## 7.3 Restricting Resource Usage using Quotas

Resources of a distributed infrastructure are in general limited. Therefore, their use has to be managed and restricted. In most cases, this is achieved by resource quotas either at project level or at application level. If the resource demands of an application are known in advance, resource accounting and quota enforcement can be done by a resource allocator or grid scheduler like they are detailed in Section 7.4.1. However, an a priori estimation of resource consumption is already very complicated for applications executed in a utility data centre or on a grid. It is almost impossible for long-running services in a global-scale infrastructure such as the proposed EDAS infrastructure or Xenoserver, as their resource demands mainly depend on user access.

Dynamic resource demands can be handled and restricted by a separate quota management service. A centralised implementation hosts all quotas of a project or an application and allocates resources to applications via resource leases [GC89]. Leases assign resources to applications for a fixed amount of time. If a lease expires the application is no longer allowed to use the resources and has to renew the lease in case the resources are still needed. This ensures that resources can be safely reassigned even if the lease holder crashes. This is convenient for non-consumable resources, which can be reused after they are no longer needed by the current application, like it is the case for memory or disk space. For consumable resources such as CPU cycles and transfer volume, which can be consumed only once, coping with node faults is not so easy and leases cannot be applied.

However, using a centralised service itself has several drawbacks even for refundable resources. First of all, additional mechanisms (e.g., active replication) are necessary to provide a fault-tolerant solution. Otherwise, the quota management service represents a single point of failure. Furthermore, the service can turn into a bottleneck and slow down the whole system, if the infrastructure scales up, or if an unanticipated high number of applications are started and stopped, frequently requesting and releasing quotas. Another

problem can arise due to the lease size. If a lease is too small, an application has to request additional resources. This might occur even frequently. If a lease is too large, the resource remains unused on the allocating node and might prevent or delay the execution of other jobs or applications.

For this reasons a novel decentralised, fault-tolerant, and scalable quota enforcement protocol is proposed that enables resource restriction for consumable but especially non-consumable resources. At its core, the idea of dynamic load balancing via diffusion algorithms is utilised. Originally independently proposed by Cybenko [Cyb89] and Bolliat [Boi90], these algorithms have been developed to equally balance load in a multiprocessor or cluster system. In the context of EDAS, diffusion is used to balance the information about free resource quotas of consumable resources. Balancing unused quotas ensures that all nodes, hosting jobs or applications of a certain project, have local knowledge of free quotas, enabling a fast resource allocation and deallocation. Whereas the original diffusion algorithms for load balancing addressed closed systems, the quota enforcement protocol targets large-scale infrastructures like EDAS. Therefore, special care needs to be taken to address node faults and fluctuation of nodes in general.

In the following, a brief overview about diffusive load balancing is given, which builds the foundation of the developed quota enforcement protocol. In Section 7.3.2, the basic protocol for non-consumable quotas is outlined and subsequently extended to support consumable quotas. Section 7.3.6 details how the proposed protocol is integrated and used in the context of the EDAS infrastructure. Section 7.3.5 describes results gained via simulations. Finally, Section 7.3.7 gives a short overview about related approaches.

### 7.3.1 Diffusive Load Balancing

Belonging to the nearest-neighbour algorithms, diffusion algorithms represent one of the most popular approaches for dynamic load balancing in parallel computing. First proposed by Cybenko [Cyb89] and Bolliat [Boi90], the basic idea behind these algorithms is to *diffuse* fractions of load to under-utilised neighbouring nodes, thereby balancing load between all nodes.

This is achieved by organising nodes in overlapping domains. In a minimal setup, domains comprise two nodes. The diffusion process starts if the load $w$ at one of the nodes—for example $i$—exceeds a certain threshold. Then Node $i$ exchanges load information with other nodes—in our case node $j$. If $w_i > w_j$, then load $\lceil \frac{w_i - w_j}{2} \rceil$ is migrated to $j$, thereby balancing the domain [1]. As discrete load is assumed, the minimal unit of load is 1. Consequently, there might still be an imbalance if $w_j < w_i \leq w_j + 1$, resulting in further unnecessary balancing operations. To avoid this problem, one can restrict balancing to differences above 1. This introduces a system-wide global load gradient that can be alleviated by probabilistic migration. This strategy migrates load with a certain probability even if the imbalance is not reduced.

[1] The description is taken form Douglas and Harwood [DH04].

### 7.3.2 Diffusion-based Quota Protocol

A large-scale infrastructure usually assigns virtual resource shares in form of quotas to projects or even single users. If a user or project submits a job or deploys an application, quotas are mapped to real resources. As resources are limited and their access potentially has to be paid, the infrastructure has to manage and control resource usage. This means that the infrastructure has to enforce that the physical resource usage never exceeds the quota of a project or user.

In the context of EDAS it is assumed that a domain manager assigns consumable resources in form of quotas to a project represented by a hosted service environment. As outlined in Section 7.1, there are two basic levels of control for resource usage: the node level and the infrastructure level. At the node level, *local limits* have to be enforced matching *local quotas* in scope of this section. At the level, *global limits* further on called *global quotas*, have to be enforced. This is achieved by ensuring that the sum of all local quotas, never exceed a global quota. How the protocol is integrated into the EDAS infrastructure is detailed in Section 7.3.6.

### 7.3.3 Basic Concept

To outline the basic concept, a simple example scenario is described. There is a project that runs only a single application, which is spread over three nodes. For briefness, the diffusive load balancing process of a single resource is examined. However, the concept can be applied to balance quotas for all resources of a distributed application.

At start-up time of the application, the global quota of the project (180 units in our example) is equally split over all supporting nodes (see Figure 7.2, step 1). Furthermore, every node is a member in at least one group comprising two nodes, as described in Section 7.3.1. These groups overlap, and thus a complete coverage is achieved. In our example, node 1 and 2 build a group, and node 2 and 3 build another one.

As the application starts executing, the resource demand rises to the amount of 10 units at node 1 and to 50 units at node 3. Step 2 of Figure 7.2 shows that the real resource usage is mapped to the locally available quota. This means that the local resource demand can be fulfilled immediately without interaction with other nodes. At node 3, the resource usage has almost reached the local quota. Now the diffusive quota balancing process starts, as the free quota has changed by more than a certain threshold at node 3 as well as at node 1. All nodes contact all members of their groups and balance the free quota as long as the changes to the local free quota exceed the threshold. If the resource demand of the application is stable, the system eventually reaches equilibrium as shown by step 3. If the application needs further resources on node 3, the infrastructure is well prepared, as the amount of free quota has risen to 40, the global free average quota.

There might be situations in which an application spontaneously needs more resources than allowed by the local quota. It is assumed that the application can either be slowed down or even suspended. If this is not possible, a local short-term credit can be given, so the application can proceed to execute. The node then waits for the balancing process to legitimate the resource usage of the application. If the necessary free quota cannot be

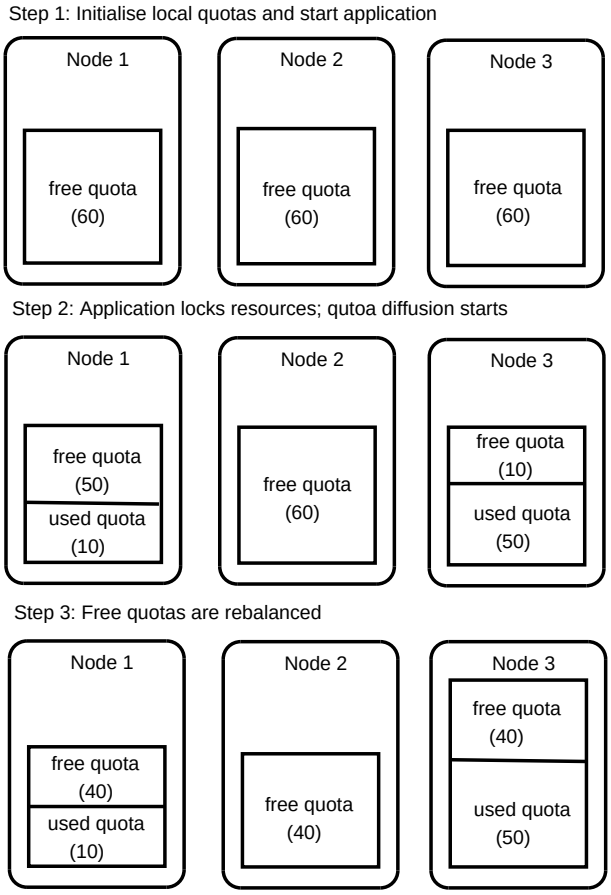



Fig. 7.2: Concept of diffusion-based quota balancing

resolved in a certain time span, the credit is simply withdrawn from the quota. If a new application should be started and the initial resource demand clearly exceeds the local free quota, a fictive demand can be generated that, if resolved, enables the safe start of an application.

At this stage it is assumed that the actually used quota does not exceed the resources provided by the underlying system. Thus, where to place an application and in case of exceeding the physical resources of a node how to redistribute it, is not addressed by the protocol. This as a separate task that can be handled by using a distributed resource allocator. In context of EDAS the quota management protocol is primarily used for the management and restriction of consumable quotas where this is not an issue. The management of non-consumable resources is handled by a different approach that is subject of Section 7.4.

### 7.3.4 Protocol Description

In the previous section, the core concept of diffusion-based quota management was outlined. This section addresses the details of the basic protocol, thereby paying special attention to dynamic changes of the node set, which are common in large-scale infrastructures such as EDAS. As a prerequisite a fail-stop behaviour of nodes is assumed and furthermore the reliable detection of node crashes.

### Data Structures

Each node possesses the following local variables for every supported project (For briefness only one resource is considered) :

**quota** - local resource quota

**level** - level of the node in the quota routing tree

**consumed** - used to account consumed quota fractions of non-refundable resources

**neighbours** - list of connected neighbours, where each entry is a triple of

- **connection** - connection to a neighbour hosting quota
- **counter** - determines the amount of quota transferred across the connection
- **level** - level of the neighbour in the quota tree

### Basic Protocol

If a new global quota for a project is granted, a set of nodes is randomly picked from all available nodes. This is done by a special node called *quota manager*, which is mainly used for initialisation and for providing a stable access point. All nodes of this initial set are connected to the quota manager and supplied with the addresses of all other initially selected nodes. Each of the nodes randomly connects some of the other nodes thereby forming a random graph (see Figure 7.3). During this procedure, every node determines its level in a tree that is formed as a by-product of the connection process. At start-up, only the quota manager representing the tree root is part of the tree and all other nodes are not yet connected. During the connection process, the neighbour with the smallest distance to the top, as determined by its level, is selected as uplink and the own level is set to one plus the level of the uplink. Thus, all nodes that possess a direct connection to the quota manager node have a level of one. Later on it is investigated how this information can be utilised to provide fault-tolerance for consumable quotas and to provide information about the amount of globally available consumable quota.

After connecting the initial node set, the global quota is injected via the quota manager by initialising its local quota with the global quota value. Then the diffusion process starts, as every node randomly balances its quota with all connected neighbours. Neighbours are basically all nodes a node is connected with. The listing in Figure 7.4 outlines the basic balancing process in a python-like programming language. For each neighbour, the amount of unused quota is determined, then this value is sent to the neighbour via `balance()`. This method adjusts the quota at the neighbour node and returns the amount by which to change the local quota. At this stage, only positive quota values are considered. Balancing is not only invoked periodically, but also by the underlying resource accounting system if the resource usage changes by more than a certain threshold or if the resource exchange with another node modifies the local quota. After this initial diffusion phase, the system eventually reaches its equilibrium, in which all connected nodes possess an almost equal part of the quota. Now one or more applications can be started at some or all of the

```
def initial_connect(self, nodes):
  for node in random.sample(nodes,max_con):
    node.connect(self):
    self.neighbors.append(node)
    if self.level == None or self.level > node.level:
      self.level= node.level+1
      self.uplink=node

def connect(self,node):
  if node not in self.neighbors:
    self.neighbors.append(node)
```

Fig. 7.3: Connecting nodes with each other

```
def do_balancing(self):
  for n in self.neighbors:
    free=self.quota−self.usage
    # ask other node how to change my quota
    self.quota +=n.balance(free)

def balance(self, remote_free):
  free = self.quota − self.usage
  diff = free + remote_free
  average = diff/2

  self.quota += average − free
  return −(average − free)
```

Fig. 7.4: Simplified basic balancing process

nodes. If nodes other than the initial set should run applications of the project, they simply connect one of the nodes belonging to network and ask for a random subset of supporting nodes. This requires some kind of discovery support, e.g., broadcast discovery or a set of well-know nodes that provide contact information such as used by Gnutella [KM]. For EDAS a similar approach can be applied (see Section 7.3.6). Once a node set is obtained, these nodes are connected via the described initial connection procedure.

#### Extension for Fault-Tolerance

As the protocol shall tolerate node crashes, additional mechanisms have to be provided. For this purpose, every node maintains a counter for each link to a connected neighbour. This link counter represents the net amount of quota transferred to the neighbour and is updated on each exchange of quotas via the corresponding link. If free quota is donated to a neighbour, the local link counter is incremented by the transferred quota value. The

```
def fix_crashedNode(self,neighbor):
    self.quota += neighbor.counter
    self.neighbors.remove(neighbor)
    # check if uplink is concerned
    self.replace_crashedNode()
```

Fig. 7.5: Recovering quota after neighbour crash

receiving node decrements its counter by the same amount. Both counters are adjusted vice versa if the neighbour provides free quota in the opposite direction. A negative value of the link counter indicates that a node just received quota from that link.

All connected neighbours of the node detect any node crash and the node is removed from the neighbour set of all of them. Furthermore, the counter value of the failed connection is added to the local value. This way, the quota originally held by the crashed node is reconstructed by all the neighbours. Figure 7.5 shows a listing of the actions a node performs when a neighbour node crashed. It has to be noted that the recovery may temporarily result in a negative local quota, but the algorithm will balance this with the remaining neighbour nodes. However, not only the quota information has to be reconstructed, but also the neighbour set has to be consolidated. If a node fails, a new connection should be set up and if the failed node has been the previous uplink, a new uplink has to be determined. This uplink can be one of the current nodes if there is one with a lower level than the node. If there is no such node a new node has to be connected that is nearer to the tree root.

Apart from the fact that this provides an easy and lightweight approach to compensate single node failures, it also offers fault-tolerance for multiple node crashes of interconnected nodes. Even network partitioning of nodes not belonging to the initial set is addressed, although a disconnected node set will run out of quota, as a negative quota is introduced due to the multiple link failures. But this would result only in suspending all applications and jobs at that node set, and after reconnection the balancing process distributes the free quota information, so all applications can be resumed.

#### Extensions for Non-Refundable Quotas

So far, the handling of faults considering only refundable quotas that have a constant global quota has been described. For non-refundable quotas, additional actions have to be taken, as their global quota decreases along the lines of resource use. If the same approach were applied to compensate node failures and all surrounding nodes reconstructed the quota via the link counter, a project would get back not only the free resources hold by the crashed node, but also the consumed amount of quota.

To address this fact, each node possesses a `consumed` counter that is incremented by the accounting system each time a fraction of a non-refundable quota is consumed. The information about the consumed quota has to be distributed to prevent the reassignment of consumed quota. This is achieved by passing the amount of consumed quota to the uplink node, which in turn passes this information to its own uplink node. This goes on until the tree root—the quota manager—is reached. Along the way to the tree root, the

link counters are adjusted to reflect the reduced global quota. The lower node at a link adds the consumed quota to its link counter; the upper node subtracts the value from its counter, similar as in quota distribution. Figures 7.7 and 7.6 show updated listings of the balancing process from Figure 7.4 that takes care of the link counters and passes the `consumed` counter up to the root of the tree.

This approach is not absolutely exact because, if a node fails before its locally consumed quota fraction has been passed to the uplink, it is simply reassigned by the neighbours. This should be negligible, as non-refundable resources usually can be consumed only at a certain rate (e.g., CPU speed in case of CPU cycles) and this restricts the amount of consumed quota as the node periodically exchanges quota information. This is not the only error that is introduced, but also the information hosted by the crashed node about consumed quota by neighbours and their children is lost. If the node is higher to the top, the amount of lost consumed quota information rises. Fortunately, not the project but the infrastructure is penalised by a node crash, as the consumed quota is reassigned by the compensation procedure. So, in case of a node failure the infrastructure will eventually grant more resources than originally remained resources.

#### Handling Failures of the Quota Manager

Up to this point is has been addressed how to compensate faults of nodes supporting the project by providing resources. For the quota manager node, additional steps have to be taken. Apart from managing the initial connection phase, this node represents the tree root and therefore is the origin of the global quota. Thus, this node has the duty to ensure the availability of the global quota despite the crash of nodes of the initial set. If a node of the initial set fails, the quota manager adds the link counter to the local quota value to compensate the failure. Additionally, it adds a new node to the initial set. In short, it handles node failures in the same way as any other node.

If the quota manager fails, the other nodes do not take any special actions to compensate the failure, as a link to the quota manager is marked to be an initial link. Otherwise, the whole global quota would vanish from the project as it has been injected via the quota manager node. This is more severe than a disconnected sub network that may run out of quota due to an missing uplink, as the whole system would stand still until the quota manager is recovered, whereas in the former case just to a new uplink connection has to be established. Thus, the whole system can proceed to execute although the quota manager has crashed. However, the quota manager has to be recovered, as nodes of the primary node set can fail, causing fractions of quota to disappear.

It is assumed that the state of the quota manager can be recovered e.g. via snapshots and infrastructures supporting passive replication. If the quota manager node is restarted, all nodes of the initial set are connected and in case one or more of them has crashed during the absence the failure handling is started.

This builds the minimal solution to handle failures of the quota manager. Section 7.3.6 addresses how to map the protocol and its requirements to the EDAS infrastructure. EDAS offers the opportunity to provide the quota manager as a replicated decentralised adaptive service providing better support for failure handling.

```
def do_balancing(self):
  for n in self.neighbors:
    free = self.quota−self.usage

    if n.level < self.level:
      # pass the consumed quota up to the root
      n.counter += self.consumed
      result = n.balance(self.id,free,self.consumed)[0]
      self.consumed = 0
    else:
      # receive consumed quota from lower nodes
      (consumed,result) = n.balance(self.id,free)
      n.counter −= consumed
      self.consumed += consumed

    n.counter −= result
    self.quota += result
```

Fig. 7.6: Initiate balancing process for refundable resources

The presented failure handling approach is based on the assumption that node failure can be reliably detected. If this is not the case in the sense that nodes of the initial set may not have crashed but are subject to a network partitioning a different approach is required. First of all not only nodes of the initial set treat node faults of the quota manager differently but also the quota manager delays the failure handling of initial set nodes until a local administrator has verified by out of band information that the node either failed or are subject to a network partitioning. In the first case the described failure handling is performed, whereas in the second case all further actions related to the disconnected nodes are suppressed until the network is healed. These considerations are only relevant to network partitioning across the initial set and no additional actions have to be taken for disconnected sub networks of non-initial set nodes.

### 7.3.5 Simulation

The protocol was evaluated by the implementation of a round-based simulator. In an initial experiment, a network of 1000 nodes supporting a project that possesses a global quota of 50,000 units of fictive resources was generated. After the initial diffusion process, every node holds a local quota of 50. To evaluate the adaptiveness to changing demands, a fictive load was introduced by starting each run a rising number of applications ranging from 0 to 500. Every application had the same resource demand of 100 units. Then the number of rounds from an unloaded network in equilibrium to reach a state where all application demands are satisfied (see Figure 7.8(a)) and the number of rounds to reach the equilibrium again (see Figure 7.8(b)) for a neighbour set of maximal 5 and 10 nodes were measured. During each round the nodes where picked in a random order to balance their quota with

```
def balance(self,id, remote_free,consumed=0):
  neighbor = neighbors[id]
  free = self.quota − self.usage
  diff = free + remote_free
  average = diff/2

  # handle the consumed quota
  if neighbor.level < self.level:
    consumed = self.consumed
    neighbor.counter += consumed
    self.consumed = 0
  else:
    neighbor.counter −= consumed
    self.consumed += consumed

  # balance the remaining quota
  if remote_free < 0 and free< 0:
    return (consumed,0) # nothing left on both sides
  elif remote_free < 0 or free < 0:
    # take care of negative quotas
    # [...]
  else: # there are free resources on both sides
      self.quota        += average − free
      neighbor.counter −= average − free
      return (consumed, −(average−free))
```

Fig. 7.7: Extended balancing process for non-refundable resources

all their neighbours. As Figure 7.8(a) shows, the demand is quickly satisfied for wide ranges of the simulation, usually within one or two rounds. Up to a resource utilisation of 98% the demand is satisfied in less than 10 rounds for 10 open connections. For 5 open connections this is still the case for 92% resource utilisation. On reaching the global limit, it takes longer and longer to satisfy the demand. This is expected, as small fractions of free quota may strand in less connected areas of the network and randomly move around until they reach a demanding node. The total number of rounds to reach the equilibrium is, in general, much higher and varies in cycles. This is the result of the strong equilibrium condition where any pair of nodes has a maximum quota difference of one unit. In a more relaxed and realistic setting where the equilibrium condition is met if any pair of neighbours has maximum quota difference of one unit, the equilibrium state is reached much faster and the peaks disappear. The last fact has been verified by a recently finished prototype [Dis07].

To delay the time-consuming phase when reaching the global limit it is useful to choose a small resource unit. Finally, it might be reasonable to consider small fractions of quota as waste. Thus, an application can be notified that a resource request cannot be satisfied instead of probably unsuccessfully waiting for numerous rounds.

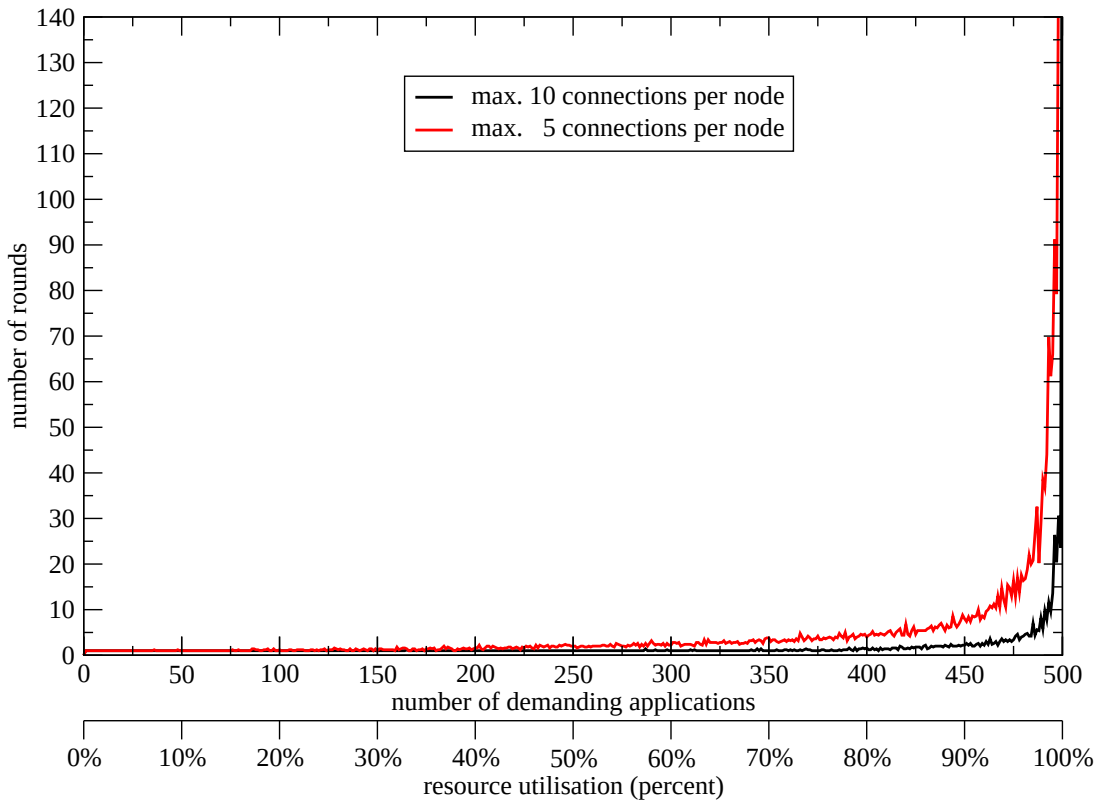
### 7.3.6 Integrating Quota Management into EDAS

This section targets the integration of the presented quota management protocol into the EDAS infrastructure. Non-consumable resources have to be provided by the underlying hardware. The quota management protocol is not sufficient to give guarantees for these kinds of resources. Consequently, Section 7.4 presents a different approach based on predefined fixed-size resource bundles. However, consumable resources can be easily managed with the presented protocol. For example, the protocol can be applied to manage the overall network transmission volume of the service environment, in the context of supporting home environment. Thus, a project mentioned by the protocol description matches a service environment that is supported by a home environment and as earlier stated a global quota matches a global limit (see Section 7.1 for global limit).

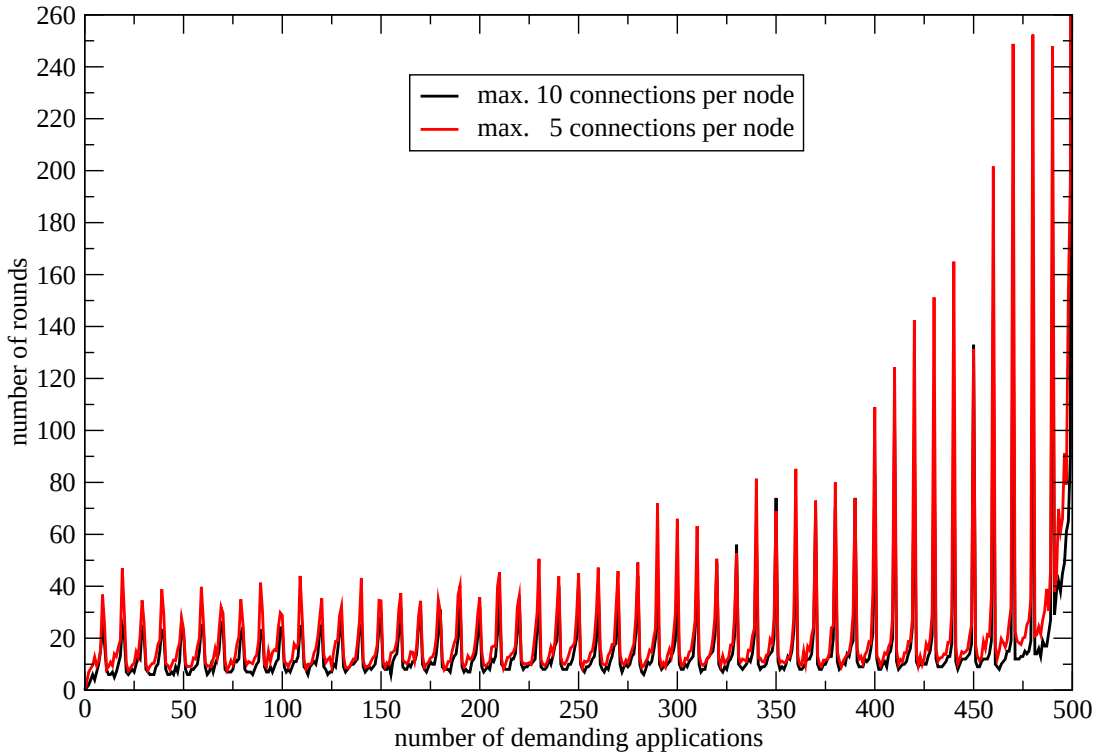
The protocol is realised as a peer-to-peer based decentralised adaptive service named *quota management service*. If a new service environment is bound by a home environment, quotas are assigned to a newly instantiated quota management services.

The quota management service is implemented by a JXTA service that according to the protocol connects all nodes of a home environment that host services of the associated service environment. This is achieved as resource managers (see Section 7.2) bind the responsible quota management service as soon as a service instance is created. This way a resource manager knows about the free quota and can handle quota requests via the quota management service.

The protocol can be implemented in a straightforward way with two extensions. The first extension concerns the failure handling of the quota manager. The SES of a service environment is responsible for the quota manager node; this includes a recovery of the manager. One replica of the SES binds the quota management service. This replica is determined by being the current coordinator of the group communication framework. To ensure fault tolerance, the quota manager service periodically hands over its state to the SES




(a) Rounds to satisfy resource demand




(b) Rounds to reach equilibrium

Fig. 7.8: Simulation results

that is transmitted via the group communication. If the coordinator replica fails the group communication framework automatically selects a new one and the quota management service is recovered by this replica. It binds the quota management service and injects the state snapshot. Thus, passive replication is used to provide a fault-tolerant quota manager.

The second extension affects the join procedure of nodes. According to the protocol description, discovery mechanisms are required to connect to the network. Fortunately, JXTA offers the option to search for JXTA services via their advertisements. Therefore, a node that hosts a new service just needs to bind the associated quota management service that spans its own peer group. (The reference of the quota management service can be obtained from the service environment.) Once joined the peer group the local quota management service fragment can use the JXTA discovery service to search for other JXTA service instances (quota management service fragments) that belong to the quota management service. Finally, it connects some of them according to the protocol.

### 7.3.7 Related Approaches

Resource management in grids, peer-to-peer networks, and other general infrastructures for distributed applications is a popular area of research. However, quota management and especially protocols and algorithms for decentralised quota management have rarely been addressed so far.

GWiQ-P [KLLS05] is a decentralised quota enforcement protocol based on the construction of a spanning forest for distributed resource location. Each node in which the resource demand exceeds the local quota forms a root of a tree; unused quota, represented by *resource coins*, is transferred to the root of these trees. In contrast, the presented protocol uses diffusion of unused quota, which balances the available resources even before nodes run out of local quota. GWiQ-P, similarly to the presented approach, uses counters on links for handling failures, but it does not address in detail the problems raised by consumable resources in failure situations.

Pollack et al. [PLG+06] recently proposed a microhash inspired approach that provides lower overhead and better scalability than current centralised quota tracking services. A quota server acts as a bank that issues resource vouchers to clients. Clients can spend vouchers to allocate resources on arbitrary nodes of a cluster system. Vouchers are only valid for a certain time in order to handle failed nodes and to ensure that clients do not overuse their quota. Although this approach offers more flexibility than other centralised quota management services, knowledge about the resource demand of client seems to be needed. Otherwise a client might request too much or too less vouchers. Whereas the first case wastes resources the second case leads to more voucher requests that might cause load at the quota server.

### 7.3.8 Summary

The presented protocol offers a novel decentralised and fault-tolerant approach for quota management in large distributed systems. It provides good support for non-consumable as well as for consumable resources. The presented simulations have shown the usability and

scalability of the protocol. Recently, a JXTA-based prototype was implemented [Dis07] that provides similar results. As outlined, the integration of the protocol into EDAS can be easily achieved and the quota management for consumable resources can be used in the context of EDAS.

## 7.4 Decentralised Application Placement

As earlier stated decentralised adaptive services should be able to adapt their distribution according to resource requirements. This is necessary due to changing user demands resulting in different resource requirements of a service and as a consequence of changing availability of resources provided by supporting entities. Accounting these facts the following section describes a protocol for a resource- and distance-aware online placement of distributed and possibly replicated services that fits especially the needs of decentralised adaptive services. This was achieved by investigating related approaches in closed and open system. Consequently, the proposed protocol takes properties of both categories into account. Approaches targeting closed systems usually require an accurate global view (e.g., utility computing) that is not available for the placement of decentralised adaptive services in a wide-area setting. On the other side, projects are explicitly supported by home environments that dedicate a fixed set of resources to projects. Thus providing an open system best-effort approach for resource management where every party can acquire resources (e.g., PlanetLab) misses essential mechanisms for admission control.

Guaranteeing certain amounts of non-consumable resources does not help much if the resources are arbitrarily distributed over a huge number of servers. As projects would like to place services that consume a certain amount of resources and services cannot be split into multiple parts, a node has to be found that provides the needed resources as a whole. Thus, just guaranteeing resources as a sum might require a global search covering all nodes and in the worst case fails as free resources are arbitrarily scattered over a large set of nodes. Even if a server providing sufficient resources is found, this might be no longer the case at the time of the service deployment as another project might already have allocated them. Consequently, only guaranteeing the availability of resources does not provide a solution. For this reason, the proposed approach guarantees that certain *resource bundles* that approximate the estimated resource demand of services are available in scope of a home environment, but provides best-effort support for additional requirements regarding distribution and network-related QoS requirements.

The remainder of this section is structured as follows: First the problem of application placement in open but also in closed systems is investigated. Then an approach is presented to categorise and partition resources of nodes to determine how many resources can be assigned to projects while preserving a certain degree of fault-tolerance and single out node fluctuation. Finally, a distributed placement protocol is outlined that utilises the proposed two-stage architecture of EDAS to search.

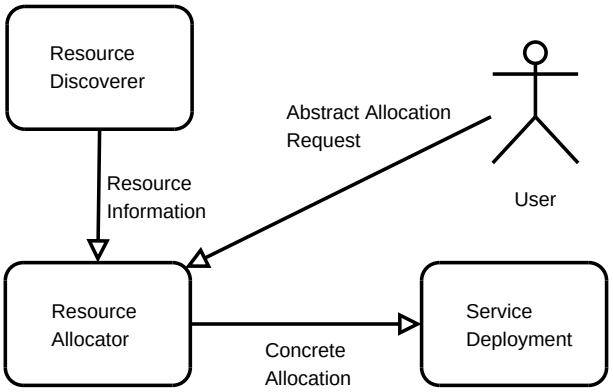



Fig. 7.9: Services for resource-aware distribution

### 7.4.1 State of the Art

Application placement has recently been investigated in several research areas like: grid computing, peer-to-peer computing, novel global-scale infrastructures, and especially utility computing. The first three areas target *open systems* with high dynamics, where multiple users concurrently access resources that are subject to changes. Utility computing in contrary aims at the reliable provision of large scale applications in the context of big computing centres. This is achieved via detailed capacity planning and requires not only mechanisms to exclusively dedicate resources to applications but also information about all available resources and application workloads [RCAM06]. This is further on considered as resource management in *closed systems*. Resource management in open as well as in closed distributed systems usually follows a similar basic workflow. The same applies for the parameters that are taken into account for the distribution of applications, more precisely the assignment of resources to applications. Despite these similarities, existing systems handle the actual application placement process differently, at least when comparing open and closed systems.

#### Basic Resource Management Workflow

Resource management in grids, but also in novel global-scale infrastructures [PACR03, HHaIP03] and testbeds for distributed applications [WLS+02], is usually split in several steps addressed by different components (see Figure 7.9 [ROLV06]). There is either a dedicated service for resource discovery, collecting information about all available resources or a resource registry that is directly informed by resource providers. To compute a task or distribute an application, a user has to provide information about the required resources. This information is used by a resource allocator or a grid scheduler to map the requirements on the available resources. In a final step, the job or application is deployed on the selected nodes.

A similar workflow is applied for deploying applications in utility farms. However, the available information-basis, used to decide how to distribute applications, is in general larger and there are tighter QoS constraints. Due to the QoS restrictions, an offline capacity

planning process and mechanisms for resource isolation are required. So far, these systems tend to over-provision to meet demand peaks but recent research projects try to improve resource utilisation by more sophisticated planning processes at various time-scales and the provision of specialised workload managers [RCAM06].

### Parameters Affecting Application Placement

As detailed, the basic approach for resource management is quite similar in all considered environments. The same applies for a basic set of parameters that are taken into account to map an application on resource-providing nodes, but varies for additional parameters like QoS requirements and time. This has two main reasons: application demands and feasibility in terms of complexity of the placement. In the following, an overview about the parameters that might be taken into account for application distribution and the way they influence application distribution.

**Resources** System resources are taken into account by nearly all infrastructures, as they are a key to successfully execute an application. In context of utility computing infrastructures, system resources can be dedicated and accounted at a very fine-grained level [SGS+05].

PlanetLab [PACR03] in contrary offers as previously mentioned a weak notion of norm resource-bundle named *sliver* that provides enough resources for one user to execute an application. However, so far there are no resource guarantees for all system resources. Thus, not only the bare resources are taken into account for the placement process in PlanetLab, but also quality metrics like load [OAPA05].

For grid computing, conceptual resources often play an important role as expensive software licences are usually attached to certain nodes. Thus applications might need a node possessing a licence for a certain program.

**Distance** Three kinds of distance can be considered for application placement:

- Network distance
- Geographical distance
- Logical distance

Long-range network distance is typically defined by message round-trip delay. Such a metric is useful to place several parts of an application within some network proximity for performance reasons.

Geographical distance is useful if a service should be placed near its users, for fault-tolerance or juridical reasons. For example, an instant messaging service should be placed on a node in the North of America as most of its users live there. This shortens the request-response times in the average case.

Finally, some systems consider logical distance as *collocation* and *dislocation*. An application might be composed of several sub-components: For example, a three-tier web-application consisting of a web-server, an application server, and a database or as another

example an actively replicated service provided by a number of identical replicas. In case of the three-tier web-application, it might be valuable to place two or even all three components on the same node, if this is possible. Collocating components is useful if they are tightly coupled and heavily communicate with each other. Thus, the first example has a collocation requirement. For the replicated service all replicas have to be placed on different nodes for fault-tolerance reasons. This can be expressed by a dislocation requirement. In most cases, collocation is optional whereas dislocation is mandatory at least if fault tolerance is the reason for the collocation requirement.

Network distance is usually essential for application placement on the PlanetLab and similar infrastructures but is also addressed in the context of grid applications, e.g., the placement of large data replicas to enable a fast access. Geographical distance can be combined with network distance as done by SWORD [OAPA05] and is mainly important for wide-area application placement. Both kinds of distances are secondary to utility computing as applications are placed on closely connected compute farms. However, utility computing requires local network capacity planning to prevent bottlenecks and optimise network utilisation. Logical distance is usually implicitly treated by infrastructures performing wide-area placement of distributed application because application components with collocation preferences are simply treated as one unit and only one unit can be placed on a node [OAPA05, SCHH05]. For utility computing, and also cluster systems, these requirements are explicitly handled as they might strongly influence application performance.

**Time** Time is a crucial parameter for application placement and influences placement decisions in multiple ways. First, the lifetime of an application might be bound as this is usually the case for grid computing applications or applications in a testbed infrastructure like Emulab [WLS+02]. Second, the resource usage of an application is often time-dependent. An application workload might have different requirements depending on month, weekday, daytime or other periodical metrics. If there are such fluctuations one can change resource assignment, and consequently the placement of an application, for utilisation reasons [RCAM06]. In both cases, time increases the complexity of the planning and placement process and requires a scheduler component. For utility computing most of the planning process can be handled offline as application workload and available resources are known a priori. Sometimes, the workload prediction of an application might be wrong or resources fail (e.g., due to a node crash). In these cases a partial online replanning process is required.

### Problem Description

After investigating the parameters that influence application placement in distributed infrastructures, this section addresses the possible variants of the application placement problem and discusses its current solutions. Thereby one can distinguish between open and closed systems as they have different basic conditions.

**Application Placement in Closed Systems** Starting with the most basic variant of a placement problem, named *basic application placement problem (BAPP)*: There is a set of nodes, each node providing different amounts of resources. Then, there is a set of applications,

each application requiring different amounts of resources. The applications should be placed on the nodes so that the amount of required resources of the applications placed on each one of the nodes does not exceed its provided resources. The optimisation goal is to utilise as much resources as possible. This problem description matches the *multidimensional knapsack (MKP)* problem, which is a strongly NP-hard combinatorial optimisation problem [KPP04].

This basic problem definition has been extended by Urgaonkar et al. [URS04]. They proposed the *application placement problem (APP)* that takes collocation and dislocation constraints of applications into account. However, they only considered a single resource scenario. They claim that even this problem is NP-hard and may not admit a polynomial-time approximation scheme (PTAS).

A more general variant, the *resource assignment problem (RAP)* has been investigated by Zhu et al. [ZSW+03]. They do not only try to place applications on a given set of servers but also optimise the network distance and utilisation in a data utility centre.

All three placement problems and other similar ones have been addressed in the context of closed system where a global view is available. There are several general approaches to solve these placement problems. First of all, heuristic algorithms like they are proposed for the MKP or in case of the APP [URS04]. Other approaches are to employ linear integer programming [Wol98] using a constraint solver [SGS+05] or by applying genetic algorithms. Due to the NP-hardness of these problems finding a suitable solution strongly depends on the problem size and is resource- and time-intensive. Thus, these approaches are only suitable for systems of limited size and a low rate of changes or in case of offline planning.

**Application Placement in Open Systems** For open systems usually the aim is not to find an optimal distribution considering a global optimisation goal, like the overall resource utilisation, but rather to find a sufficient distribution for an application having certain requirements. This relaxed form of a placement problem can be named the *open system placement problem (OSPP)*.

To solve this problem, information about the available resources has to be gathered. This information usually represents only a partial view that might even be outdated due to system dynamics. However, this view is queried to find a set of nodes matching a given application placement request. Sometimes, there might be multiple sufficient answers. Then, the application can refine the query, rate the answers, or choose randomly one answer. During the actual deployment it might turn out that the demanded resources are no longer available. In this case, a different answer has to be evaluated, or the initial query process is restarted.

#### Related Approaches: Closed Systems

For closed systems, two research projects will be presented, both representing protagonists for a group of related projects. Sharc [URS04] primarily targets resource management in shared commodity cluster systems for three-tier applications. The Quartermaster [SGS+05] project in contrary addresses large-scale IT applications in scope of utility data centres.

**Sharc** At its core, Sharc provides a resource management layer on top of standard operating systems that provide some notion of quality of service like reservations [JRR97] as offered by Solaris and other operation systems. Every node of a Sharc cluster hosts a *nucleus* component that consists of a resource manager component and a fault-tolerance component. The resource management component is responsible for the management of resources in scope of a node. The fault-tolerance component is provided to tolerate node faults, link faults and failure of the control plane.

The control plane is needed if an application is placed on a Sharc cluster. For resource allocation the application has to be described in a simple specification language. This language offers to specify an application that is composed by multiple components called *capsules* each having different resource requirements. An application can explicitly allow resource trading of reserved but currently not allocated resources. This way, the resource management component is able to dedicate these resources to capsules on the same node that currently have higher resource demands. The actual placement of applications is performed at runtime by a heuristic algorithm with linear runtime [URS04, US04]. Sharc considers only two resources CPU and bandwidth, for the placement process and only one at a time.

The Sharc project is only one of several systems targeting the resource-aware distribution in commodity clusters. The cluster reserves system [ADZ00] but also Chase et al. [CAT+01] provided resource management infrastructures in cluster environments. The main difference between these two approaches and Sharc is the actual management of only one resource namely CPU. However, according to the authors both systems can be extended to support further resources.

**Quartermaster** Quartermaster [SGS+05] has been designed and implemented by HP Labs as an integrated set of tools for the reliable provisions of large-scale applications in the context of enterprise IT environments. Its tool chain addresses policy-based resource composition to speed up the design of applications and resource providing environments and to reduce system design errors [SSMJ04]. At runtime, Quartermaster provides algorithms for scheduling and capacity management targeting complex resource usage pattern and time-dependent resource demand changes [RCAA05]. Finally, the tool set supports fine-grained resource assignment by a placement solver that uses CPLEX, a commercial solver to do offline but also online planning of resource allocation and placement [ZSW+03].

In contrary to the shared cluster systems, Quartermaster addresses resource planning and management more comprehensively as applications and resources are described and modelled before the deployment. This enables to buy and integrate more hardware for new applications before deployment time if the exiting resources turn out to be not sufficient. The placement solver follows also a more compressive approach as not only CPU and bandwidth are considered but also other resources and QoS requirements.

**Summary** In contrary to resource management in cluster systems, the resource management in utility data centres starts before an application is deployed. This offers the opportunity of an offline planning process that can take more time. However there is still a

need for an online placement process to address node faults and demand peaks. For the online problem, two general approaches have been proposed using heuristic algorithms or employing integer programming and a solver.

#### Related Approaches: Open Systems

Whereas closed systems are described more for completeness reasons, the following open systems are closely related to EDAS and the decentralised application placement approach presented in this section. As mentioned earlier, SWORD but also Xenosearch follow the basic resource management workflow outlined in Section 7.4.1. Both systems are in the following briefly summarised, together with their related projects. Finally, a small set of other open systems is presented that take a more decentralised approach to do application placement.

**SWORD and PlanetLab** As mentioned in Section 5.1.3, PlanetLab [PACR03] is a planetary-scale testbed for distributed algorithms, protocols and applications. It consists of a number of educational institutions and companies each of them providing a set of nodes running an enhanced Linux operating system that provides support for resource isolation [BBC+04]. Such an isolate is named a *sliver* and represents a resource bundle that should fit the average needs of an application. A researcher can get a *slice* of the PlanetLab denoted by a set of distributed slivers for a period of several months to perform experiments. So far, researchers have to select nodes on their own to setup slivers to build a slice for their experiments. This is cumbersome and time-consuming since the nodes should usually be located in certain geographical or network distance. If there are popular selection patterns this leads to overcrowded nodes and usually distorts measurements.

The SWORD project [OAPA05] aims at providing a distributed resource allocator simplifying the node selection process. This is achieved by providing a query language to specify resource and distance requirements for groups of nodes. Additionally, one can describe how these groups should be interconnected and where they should be located. The authors refer to CoDeen [PWP+04] a content distribution network as a good example for these requirements as it needs small clusters of tightly coupled nodes at certain geographical locations that should be well inter-connected.

To answer such queries, SWORD resembles a distributed query processor consisting of nodes organised by a distributed hashtable (DHT) algorithm [DBK+01, RD01]. DHT algorithms map arbitrary data to keys. These keys can be mapped to nodes that are responsible to store the key and the data. Consequently, one or more nodes are responsible for a certain part of the keyspace. The DHT algorithm thereby has to ensure that keys can be found despite changes of the node set. SWORD modifies the generation of the keys by encoding resource attributes and values in the keys. This offers a basis for doing attribute-basd range searches.

To query data in a DHT system it has to be inserted in the first place. In context of SWORD several monitoring services offered by the PlanetLab have been used to provide all kinds of resource information about the nodes. This information insertion process represents a key point of the system. If the resource information is updated too seldom it might be outdated.

On the other hand frequent updates lead to high traffic. Oppenheimer et al. [OCP+06] therefore did extended experiments to evaluate the update rate of several resources based on traces of the PlanetLab. The results indicated that some resource information is subject to quick changes whereas other information can be updated at a more coarse time scale. The latter applies for inter-node latency and bandwidth.

Complex queries, especially if they include range requests, might result in more than one answer. To select the best or a set of best answers the requesting application or user can provide penalty functions per attribute. These functions are evaluated by an optimiser component inside a SWORD node that received a query. The selection process is an NP-hard problem and its duration can be restricted by a timeout.

**Xenosearch and Xenoserver** As detailed by Section 5.1.3 the Xenoserver platform [HHaIP03] can be roughly compared to the PlanetLab in its structure.

The Xenoserver platform has a similar demand for a resource allocation engine like SWORD that allows an application or user to select a set of nodes that fulfil certain requirements. Xenosearch [SH03, SCHH05] and SWORD share the same basic concepts. Both provide a query language to describe resource requests consisting of interrelated node sets. In contrary to SWORD, a request is first pre-processed by a query-simplifier component before a distributed index is queried that is build up by the resource information provided by all resource-donating nodes. The results are preliminary and have to be evaluated by an optimiser component that selects the best answers that are passed to the requesting application. From an application perspective, the Xenosearch provides a distance notion by the keywords *near* and *far* to collocate or dislocate applications in a wide-area setting. This seems to be convenient for many application demands.

**Other Related Approaches** Iamnitchi and Foster [IF01] proposed the usage of decentralised peer-to-peer request forwarding strategies to handle resource request in a grid infrastructure. In contrary to SWORD or Xenosearch that are based on DHT algorithms, such an approach is not limited to well-known attributes but offers support for full-text searches and is in general more resilient to peer churn. The downside of decentralised peer-to-peer techniques is the higher network load due to request flooding.

At HP Labs, researchers [AGKT02, GAKT04] investigated several approaches to solve placement problems not only in a closed scenario like described for Quartermaster but also in a wide-area setting where multiple sites cooperate to distribute applications. First they outlined that the accuracy of a placement decision depends on the time the planning process takes. If resource efficient placement and strong QoS requirements are mandatory they propose the usage of integer programming like outlined in context of closed systems. However, this approach is usually not suitable for an open system where a accurately global view is not available. Therefore they proposed three different approaches:

- Ant-based control algorithm
- BLE-based algorithm
- Agents in overlay networks

The ant-based control algorithm has an initial principal that is called a service manager. This manager has to place an application consisting of multiple services on a set of nodes. It initially creates a set of agents, places them in scope of the infrastructure together with the list of services comprising the application. The ants move from one node to another and map a service on each node they visit. Each node has a local pheromone table that is updated by evaluating which services would fit to the resources of the current node. These tables are exchanged among the neighbourhood and build a background for further migration decisions. At the end the service manager evaluates based on an objective function which ant offers the best distribution.

The BLE-based algorithm resembles ideas from Broadcast of Local Eligibility used for the coordination of robots [WM01]. All nodes are organised in a cluster and communicate with each other via broadcast messages. Distribution and redistribution of applications is evaluated in so-called decision cycles. All nodes broadcast a list of services they host and new services that should be placed. Afterwards, each server evaluates if new services can be hosted using an objective function. In a next step all nodes publish their evaluation results. Finally, every node knows, based on the published results, which service should be hosted and which migrated to different nodes. As such a cluster approach using broadcast does not scale, the authors suggest using gossiping algorithms for efficient communication and the introduction of cluster hierarchies. A similar approach targeting the adaptive resource management for applications with multiple end-to-end QoS guarantees like predictable latency and jitter has been proposed by Repantis et al. [RDK05]. This work proposed a more detailed inter-cluster protocol and targets the transmission and transcoding of multimedia streams.

The agent-based algorithm can be compared to the ant-based one as entities travel through the network to evaluate a set of suitable locations. However, in contrary to the ant approach, the agent moves from one node to another and evaluates each time how to place all services in one step on the current node and its neighbourhood. To guide an agent to locations where enough resources are available the authors suggest to use an overlay network supporting range queries to search for node sets with appropriate resources.

Malek et al. [MMRM05] proposed a decentralised redeployment algorithm based on auctions. A component that should be deployed or redeployed is auctioned by the local system that acts as both auctioneer and bidder. Furthermore, all nodes of the local domain can bid for the component. As a node can only participate in one auction a component might be auctioned multiple times to find the currently best solution. This is the case when further auctions determine the same winner.

**Summary** All presented approaches target a decentralised solution requiring only a partial system overview to distribute or redistribute applications. SWORD and Xenosearch provide more or less distributed query engines on top of a set of nodes building an enhanced peer-to-peer network. These index-carrying nodes have to be filled with information by all nodes providing resources. Both systems assume a set of more or less stable nodes. Furthermore, it is anticipated that the resource availability changes moderately and according to Oppenheimer et al. [OCP+06] this is the case at least according to experiments on the PlanetLab.

Except for the agent-based approach proposed in [GAKT04], all other systems require less infrastructure and target a more decentralised approach that aims at an incremental improvement of distribution. It is complicated to compare the remaining three approaches among each other. However, it seems clear that the ant-based algorithm is in general applicable to place applications with arbitrary resource demands. In some cases, rare resources may be hard to find. The BLE-based algorithm might be only useful for applications with standard resource demands as the search scope is more or less limited to the local cluster and therefore seldom conceptual resources (e.g., a special software licence) are hard to find. However, it might be possible to enhance the algorithm by propagating resource class information among clusters or hierarchy levels. Finally, the described auction algorithm [MMRM05] does not scale for large domains where many nodes bid for the same component and many auctions take place at the same time.

Finally, with the exception of SWORD and Xenosearch, geographical location as is not explicitly treated but rather implicitly as a QoS requirement.

### 7.4.2 Resource Partitioning of Nodes

The prerequisite for the presented placement protocol is an approach to pre-partition the resources offered by a node. This happens before a node joins its associated home environment. For every system resource like memory, disk, network and CPU four coarse-grained *resource size classes* were defined: *small, medium, large* and *extra large*. The exact size of a resource size class depends on the implementation platform, the services and the available nodes. In context of EDAS targeting decentralised adaptive services a classification as outlined by Table 7.1 was identified. Thereby, the resource demands of typical Java services, and the resource settings of commercial hosting services such as Amazon Elastic Compute Cloud (Amazon EC2) [Ama] as well as the settings of PlanetLab, were taken into account. In general, the first two resources, memory and disk are mandatory to successfully execute a service whereas the second two resources, network and CPU, have a strong influence on service quality. After the prioritisation: memory, disk, network, CPU the different resources are combined by size classes descending from extra large to small. An average machine in a laboratory might offer the following resources 4 GB memory, 160 GB disk, 4 Gflops, 1 GBit/s. These resources can be partitioned according to the resource size class table into four resource sets. Such a resource set will be further on called a *resource bundle*.

| Resource | small | medium | large | extra large |
|---|---|---|---|---|
| memory | 125 MB | 250 MB | 500 MB | 1 GB |
| disk | 4 GB | 8 GB | 16 GB | 32 GB |
| network | 10 Mbit/s | 50 Mbit/s | 100 MBit/s | 250 Mbit/s |
| CPU | 125 Mflops | 250 Mflops | 500 Mflops | 1 Gflops |

Tab. 7.1: Resource size class table

The pre-partitioning of resources provides the benefit to enable the assignment of resources as coarse-grained sets. Thus a project can instantly decide if a service can be

mapped to a resource bundle or not. This offers already a weak form of local resource guarantees that is further on improved for a distributed setting to alleviate the process of application placement.

The drawback of a pre-partitioning of resources is the introduction of fragmentation. In a worst-case scenario the amount of offered resources varies strongly across the different kinds of resources. For example, a node offers only a small amount of memory but large quantities of disk, network, and CPU. According to the partitioning process, maybe only one bundle can be mapped and all other resources not covered by this bundle are not further taken into account and are therefore wasted. However, this is not an issue, as most current nodes possess a balanced resource set to execute average applications and services. On the other side, the introduced resource size classes and the bundles formed by their aggregation should satisfy the resource demands of average services. Taking both facts together, a carefully chosen resource size class table should produce good results despite resources are wasted.

### 7.4.3 Gaining Global Knowledge about Available Resources

To reliably assign resource bundles to projects and to guarantee their existence, the amount of globally available resources has to be determined in advance. Otherwise, it might be possible to accidentally overbook resources and grant resource bundles to projects that simply cannot be provided. Despite the fact that resources offered by a single node are fairly static, determining the amount of globally available resources is not that easy as the node set of a domain is subject to changes. Nodes may join or leave a home environment at any time ( e.g., due to host crashes or maintenance reboots).

To establish global knowledge about the available nodes and the resources, the home environment is extended. Every node that joins its associated home environment provides the information about the offered resource bundles to the locally bound home environment fragment. The fragment wraps this information into a custom resource advertisement and publishes it in scope of the peer group of the home environment. This way the resource information about all nodes should be accessible via the peer group and can be collected using the JXTA discovery mechanisms. This is done by the HEMS (see Section 5.2) that periodically searches them. As every advertisement has a lifetime, advertisements have to be republished on a regular basis and advertisements of crashed nodes will be eventually removed from the result set as they expire. Still, there might be non-expired advertisements even though the node has already dropped support. One way to reduce the impact of such false positives is to reduce the lifetime of advertisements, which would increase the network load because they have to be republished more frequently. Additionally, nodes that voluntarily shutdown can explicitly notify the HEMS; and also other nodes that detect a crashed node can inform the HEMS. Once discovered by the HEMS, every resource advertisement is inserted into a local database and removed if the node is no longer available. This way the HEMS has a global knowledge about all available resources not including very recently joined or crashed nodes, so the absolute amount of all available resources is not exactly known.

Such short-term inaccuracies are only a minor problem; the more important question is

how much resources can be reliably assigned to projects without accidentally overbooking. This might happen if all resources of nodes included in the database are assigned to projects and some nodes crash or simply leave as they shutdown for maintenance reasons. To prevent such a problem resource fluctuation has to be singled out from the assignment process. This is achieved by estimating the maximal number of nodes $n$ that simultaneously leave the home environment while no other node joins or rejoins the network. In a next step one can withdraw that number of nodes from the database and gains the amount of resources that can be safely assigned by summing up the bundles of the remaining nodes. The question is which nodes should be removed from the set of available nodes as nodes can provide different amounts of resources. A conservative approach would be to remove the $n$ most providing nodes. This brings up the question how to determine the maximal number of simultaneously leaving nodes.

In general, this number is domain-dependent as it strongly relates to what kind of nodes provide the resources, e.g., desktop computers in an office or machines in a local computer pool environment or dedicated machines in a server room, and how they are managed. According to Oppenheimer et al. [OCP+06] only two-third of all machines in the PlanetLab are functioning at any one time. This might build a lower bound and in general higher values can be expected in better administrated environments (e.g., companies). Instead of specifying an exact number, a percentage of nodes that can fail or leave at a time is more useful and helps to adapt to long-term changes of the domain size. For instance this is the case if a new computer pool is added to the home environment.

More sophisticated mechanisms than declaring a static fraction of nodes that simultaneous can fail are conceivable as there might be nodes that will fail more frequently than other ones. An option might be to keep uptime statistics that can lead to different estimations of simultaneously leaving nodes [ADS06].

### 7.4.4 Assigning Resources to Projects

In Section 5 it was already outlined that a project is represented by a service environment. At the home environment level, a project is represented by a replicated decentralised adaptive service, the SES (Section 5.3). If a home environment offers resources, it starts such a service and notifies it about the resources that have been assigned to the project by passing a resource bundle list and quotas for non-refundable resources like transmission volume. The bundles are withdrawn from the total number of unassigned resources managed by the HEMS. This way it is known how many resources can be used by a project and how they are arranged. However, it is not known which nodes will provide the resources and therefore has to be determined at runtime. The SES connects itself to the JXTA peer group of the service environment and further on responds to resource queries like described in Section 7.4.6.

### 7.4.5 Providing Distance Information

To enable a network and geographical distance-aware service placement every SES needs appropriate distance information. Therefore a SES has to know its GPS coordinates. This

information should be provided by the local home environment and has to be initially set by the domain manager.

At start-up of an SES, every other SES of the associate service environment is connected and the network delay is determined by an all-pair-wise ping as proposed by Stribling referenced by [OAPA05]. This information is updated from time to time but according to Oppenheimer et al. [OCP+06] it should be fairly stable. All nodes of a home environment approximately should have the same geographical location and share the same network. Thus, the ping values can be used to determine the network distance between two home environments. This is useful for service placement as outlined in the following section.

### 7.4.6 Placement of Services

Recapitulating, every SES instance has information about its geographical location, network distance to other supporting home environments and, most important, its granted resources. If a service should be initially distributed or redistributed, a query is sent to all SESs of a project. This query can include geographical location, network distance, dislocation/collocation and resource requirements. At the implementation level this can be achieved by using a propagate pipe provided by the JXTA middleware. The query is answered in a two-stage process. First, it is evaluated, if a home environment is a potential candidate to host the services that have to be placed. Second, one or more of the candidates are selected from the response set and the actual service placement is performed. The first step representing a preselection process that should reduce unnecessary network traffic and workload at the home environment level. The second step verifies if the selected home environment offers suitable places for execution if there are extended requirements like collocated or dislocation.

#### Searching for Candidates of Service Placement

In the most basic case, a service consists of a single component that should be placed on a node providing sufficient resources. An application wishing to distribute such a single instance service connects the local SES that submits the query to all connected SES.

On reception of such a placement request, a SES instance consults its bundle list and if a sufficient bundle is available that provides at least the requested resources, a response is returned. This response includes information about the total amount of currently free resources in scope of the answering domain. This offers the necessary information to rate all resource providers. A heavily loaded domain is rated lower than one that still has plenty of resources available. The reason is that the offered resources may be taken by another resource requester. This is less likely if there are a lot of free resources available. Requesting a set of bundles to place several services at a time that should be located at one domain can be handled the same way as requesting a single bundle.

However, a query can be more specific, e.g., include a geographical location and a distance range or a certain internet address and a network distance. The first case can be handled in a straightforward way as every SES posses its geographical location and is therefore able to determine if its home environment matches a distance requirement or not. In the sec-

ond case, the network distance to the provided internet address has to be determined at run-time by measuring the message round-trip delay. This might cause a denial of service attack like behaviour if a service environment is supported by a vast number of home environments. To alleviate this issue each supporting SES should randomly delay its distance measurement. Another solution offers the usage of network coordinates like proposed by Ng and Zhang [NZ02]. Furthermore, there can be queries that require several interconnected bundle sets that should be in a certain network distance to each other. Then a SES checks if one or more of the resource requirements of the required resource bundle sets can be met by the local resources. If this is the case, the distance requirements are evaluated by comparing the distance information to other service SESs. Should one or more distance requirements match other SESs, an appropriate response is returned. It includes for each of the matching resource sets a vector of domains that are reachable within the required distance. At this point it is not known, if the other domains provide sufficient resources. Consequently a client has to wait for the answers of the supporting domain. If a response is received a matching process has to be performed. This post-processing step at client-side requires combining the different answers and rate the best ones. This might be complex and should be performed by using an optimiser similar to the approach taken by SWORD and Xenosearch.

Finally, there might be collocation and dislocation requirements for single bundles or whole sets. It is anticipated that these kinds of requirements are only useful if several services or service components should be placed within one domain. Thus it is assumed that a bundle set has to be provided by one domain. To decide if a domain offers sufficient resources it is not enough to consult the bundle list but it has to be verified if the resource bundles are spread over different nodes or collocated as required by the placement request. This is addressed by the second step of the placement protocol, detailed by the following section. Thus in the first place a response is returned if there are sufficient resources available regardless if these resources are collocated or dislocated.

#### Performing Service Placement

Up to now the SES reports the possibility that a certain placement request can be fulfilled by a domain or in case of complex multi-set queries that a domain can at least partly fulfil a placement request.

After the first phase, an application has a list of answers and has to select appropriate candidates for the second phase of the placement process. A candidate domain or candidate domains are connected via their SES and the domain specific part of the original query is reissued. The SES has the duty to search for appropriate places for service placement and reduces the amount of free resources according to the requested amount of bundles. The search is performed again using a JXTA propagate pipe that connects all nodes of the home environment. Nodes that provide sufficiently free bundles answer by returning their generic peer factory reference and their free resources. The application can then select the appropriate nodes and issue the service creation by invoking the responsible generic factory. Thereby, first the resources are acquired by the resource manager from the SES and if all resources can be granted as requested, the service instantiated and started.

### 7.4.7 Summary

In contrary to previous work targeting open systems the proposed approach offers guarantees for resource availability by introducing resource size classes and bundles. Furthermore, this work assumes that projects represented by service environments are explicitly supported by domains assigning resources. Consequently, the resource management process and especially the placement protocol address these circumstances by providing an own search scope and infrastructure for every project for finding appropriate domains. At the level of a domain, all projects share the same infrastructure to search for nodes providing appropriate resources. This divides the query load according to the proposed two-level hierarchy and their responsibilities.

Despite this fact, all searches at service environment but also at home environment level rely on broadcast searches using mechanisms provided by the JXTA middleware. This might introduce scalability problems for very large domains or service environments. As JXTA has been designed to integrate all kinds of protocols the integration of scalable broadcast mechanisms is possible for example using probabilistic broadcasts and gossip based communication protocols. Another opportunity would be to integrate indexing mechanisms to implement range-based queries like proposed by SWORD and Xenosearch. Finally, the proposed protocol and its underlying approach address the issues of a resource-aware distribution of decentralised adaptive services but a concrete evaluation that is part of ongoing work is missing at this point.

## 7.5 Summary

This chapter targeted to enhance EDAS by resource management support to enable a resource aware distribution. This is achieved by offering a monitoring and accounting infrastructure based on the mechanisms provided by the Sun JVM. Utilising this infrastructure it was described how to enforce global resource limits in scope of a home environment by the proposed decentralised quota management protocol. Especially for the management of consumable quotas, this protocol is a promising approach.

Finally, the management of non-consumable resources was outlined. The proposed solution makes use of the two-level architectural hierarchy composed by service and home environment. The protocol picks up relevant parameters for an efficient and location-aware service placement. Furthermore, mechanisms to reliably assign resources to projects but also to restrict their usage were proposed. Both issues are not addressed by current open system placement approaches.

# 8 Conclusion

This thesis aimed at answering the four questions posed in the introduction chapter regarding aspects of novel global-scale infrastructures for long-running services:

- How do new service models look like?
- What is a suitable architecture for these systems?
- How to manage the resources for long-running services?
- How to dynamically deploy and redeploy services?

The last question, how new service models for long-running services should look like was expatiated at first by proposing decentralised adaptive services as a flexible and decentralised alternative to traditional object-oriented client/server services. The initial starting point is represented by the Aspectix middleware that at its core provides an extended CORBA ORB, supporting the fragmented object model. Based on this support two service models were proposed, named replicated decentralised adaptive services and peer-to-peer based decentralised adaptive services. The usefulness and applicability of both service models was verified by the implementation of two basic services for distributed development namely a decentralised version control system and a multiuser chat application. Furthermore, the EDAS architecture builds a use case for decentralised adaptive services.

EDAS provides a distributed execution and management environment for CORBA based services and especially for decentralised adaptive services. This is supported by two main components: the home environment managing the resources provided by nodes of a domain and the service environment providing an distributed execution scope for services spanning multiple domains. Summarising, EDAS builds the answer, found in scope of this thesis, regarding the question what is a suitable architecture for a distributed infrastructure for long-running services.

The basic EDAS architecture was extended by the support for decentralised dynamic loading of platform-specific code and the support for resource management. The latter can be subdivided in three different parts: resource monitoring and accounting; usage restriction via quota management for consumable quotas and finally support for resource assignment of non-consumable resources including resource-aware service distribution. The first extension is the answer to the question how services in the context of dynamic global-scale environments can be deployed. The second extension gives an answer to the question how the issue of resource management can be solved in such systems.

## 8.1 Main Contributions

Active replication of object-oriented services has previously been addressed by various research projects. Unlike these approaches, replicated decentralised adaptive services utilise the fragmented object model for the implementation. This has two main advantages. First of all, it allows transparent integration of replication at client side and, based on the profile manager support provided by the Aspectix middleware, it requires no changes to the ORB implementation. Second, and even more important, by using the fragmented object model and semantic annotation at the interface level, custom code generation is enabled. This offers a very flexible way for client-side programming and enables the implementation of complex fault-tolerant services like demonstrated by the implementation of a decentralised version control system.

As replicated decentralised adaptive services should be able to migrate at runtime and thereby completely change their set of hosting nodes, special emphasis was put on the provision of an extended support for migration. Time-bound object references offer a concept to reliably bind services despite migration and provide support to keep track of distribution changes without requiring a central location service or concepts like request forwarding proxies. As migration of replicas usually requires a state transfer that delays message delivery and therefore extends request response time especially if the service state is rather large, extended support for state transfer is needed. In scope of this thesis, a stream-based state transfer interface and parallel download algorithms were developed. This enables the transfer arbitrarily large application states from all members of a replication group to a joining node.

Peer-to-peer based decentralised adaptive services offer in contrary to other related approaches a very lightweight and lean integration of peer-to-peer techniques into the CORBA middleware and CORBA-based application. The contribution is here the mapping of a JXTA service to a CORBA interface and the transparent integration into the middleware by using the profile manager concept offered by the Aspectix middleware.

EDAS resembles the architecture of a grid infrastructure in the context of a object-oriented middleware. Both expose the concepts of domains that provide resources and projects that utilise resources of multiple domains. Furthermore, the EDAS infrastructure is based on decentralised adaptive services and therefore offers flexibility and fault-tolerance as there are no central, non-replicated components.

On top of this basic infrastructure the support for decentralised dynamic loading of platform-specific code is provided. In contrary to other related approaches almost all steps for the provision and the integration of code were decentralised. This was mainly achieved by the extension of JXTA-provided mechanisms for service description.

Finally, EDAS was extended by resource management facilities to manage and restrict resource usage in the scope of home and service environment. The presented work provides two main contributions: first, the decentralised quota management protocol that, based on diffusion and extended mechanisms, is scalable and fault-tolerant; second, an application placement protocol utilising the architecture of EDAS and the introduction resource bundles. The latter reduces the problem complexity and search effort to find a suitable location for a service or a groups of service components.

## 8.2 Limitations and Future Work

All questions posed at the beginning of this thesis have been processed. However, the answers are given from a middleware-centric perspective and have to be considered as one way to answer them.

Whereas the decentralised adaptive service models and large parts of the EDAS infrastructure are fully implemented and validated by applications, this is not the case for the placement protocol. Here, further research is necessary (e.g., to evaluate the scalability properties of the protocol).

Parts of this work like the decentralised dynamic loading infrastructure or the diffusive quota management protocol are fairly independent of decentralised adaptive services and EDAS. Therefore it would be worthwhile to evaluate these concepts and techniques in the context of different environments, like the Globus Toolkit for the quota management protocol or the dynamic loading of OSGi bundles in case of the dynamic loading infrastructure.

The proposed decentralised adaptive service model and EDAS offer only weak security mechanisms. The only exception builds the binding process for time-bound references. In general it is assumed that the utilisation of standard mechanism like access control using certificates or code signing will solve most of the security issues. Despite these facts an exhaustive analysis and the integration of these mechanism has to validate this.

# Entwicklung und Betrieb dezentraler adaptiver Dienste

Der Technischen Fakultät der
Universität Erlangen-Nürnberg

zur Erlangung des Grades

DOKTOR-INGENIEUR

vorgelegt von

**Rüdiger Kapitza** - Erlangen 2007

## Kurzfassung

In den letzen Jahren wurden zahlreiche Forschungsinitiativen gestartet um mit dem exponentiellen Wachstum an Rechenkapazität, Anzahl der verfügbaren Systeme und vor allem Information zu Recht zu kommen. Hauptziele dieser Unternehmungen sind die verbesserte Verwaltbarkeit und Nutzung der zur Verfügung stehenden Ressourcen.

Ziel dieser Arbeit ist die Nutzung verteilter Ressourcen zur skalierbaren und dauerhaften Bereitstellung von langlaufenden Diensten. Um dies zu erreichen stellt die Arbeit Entwicklungs- und Infrastrukturunterstützung für replizierte dezentrale adaptive Dienste und Peer-to-Peer basierte dezentrale adaptive Dienste bereit. Beide Dienstmodelle vermeiden zentrale Komponenten und unterstützen die dynamische Verteilung von objektorientierten Diensten gemäß dienstinterner und dienstexterner Anforderungen.

Replizierte dezentrale adaptive Dienste werden durch ein Framework unterstützt, welches die flexible Entwicklung von fehlertoleranten Diensten unter Verwendung von aktiver Replikation ermöglicht. Im Gegensatz zu existierenden Ansätzen wurde das fragmentierte Objektmodell verwendet. Dieses ermöglicht eine einfachere Unterstützung von fehlertoleraten Diensten im Kontext von Standardmiddleware, wie am Beispiel von CORBA gezeigt wird. Weiterhin erhält das entwickelte Framework weitgehend die Flexibilität fragmentierter Objekte durch die Einführung von semantischen Annotationen auf Ebene der Dienstschnittstelle für eine maßgeschneiderte Codegenerierung. Dies ermöglicht die Entwicklung von komplexen fehlertoleranten Diensten wie am Beispiel eines verteilten Versionskontrollsystems validiert wurde. Um die dynamische Verteilbarkeit von replizierten dezentralen adaptiven Diensten zu unterstützen wurde ein spezielles Objektreferenzkonzept entwickelt, welches einen sicheren Bindeprozess trotz Ortswechsel des Dienstes gewährleistet und Techniken bereitgestellt, die einen schnellen Transfer von großen Dienstzuständen ermöglichen.

Peer-to-Peer basierte dezentrale adaptive Dienste ermöglichen die Nutzung verteilter Ressourcen unter Verwendung von Peer-to-Peer Mechanismen. Dies wurde durch die Integration der Peer-to-Peer Middleware JXTA in CORBA unter Verwendung des fragmentierten Objektmodels erreicht. CORBA Anwendungen haben nun die Möglichkeit Objekte zu binden, die intern Peer-to-Peer basiert interagieren und damit das Potential haben besser zu skalieren und fehlertoleranter zu sein als traditionelle Klient/Server Objekte.

Die beiden vorgestellten Dienstmodelle bilden die Basis für die entwickelte Infrastruktur *Environment for Decentralised Adaptive Services (EDAS)*, welche es ermöglicht Dienste zu betreiben und zu verwalten, die über mehrere administrative Domänen verteilt sein können. Die Verteilung von Diensten und das Binden von dezentralen adaptiven Diensten im Allgemeinen erfordert die Instanziierung von einer objektspezifischen Implementierung deren Code nicht am Zielort vorliegen muss. Als Konsequenz bietet EDAS Systemunterstützung zum dynamischen Laden von plattformspezifischem Code an. Im Gegensatz zu verwandten Arbeiten werden heterogene Systeme betrachtet und es sind alle Schritte des dynamischen Ladens, wie Suche, Auswahl und Transfer des Codes dezentral gelöst. Weiterhin bietet EDAS Konzepte zur verteilen Ressourcenverwaltung. Es wurde ein Protokoll zur dezentralen und fehlertoleranten Einhaltung globaler Ressourceschranken und ein Protokoll zur ressourcegewahren Verteilung von dezentralen adaptiven Diensten entwickelt.

# Inhaltsverzeichnis


**1 Einleitung** **1**
1.1 Ziele der Arbeit . . . . . 2
1.2 Wissenschaftlicher Beitrag . . . . . 3
1.2.1 Entwicklungsunterstützung für dezentrale adaptive Dienste . . . . . 4
1.2.2 Infrastrukturunterstützung für dezentrale adaptive Dienste . . . . . 6
1.3 Gliederung der Arbeit . . . . . 7

**2 Die Aspectix Middleware** **9**
2.1 CORBA Architektur . . . . . 10
2.2 Status Quo der erweiterbaren Middlewareansätze . . . . . 11
2.2.1 Adaptionspunkt von CORBA . . . . . 11
2.2.2 Adaptierbare Middlewareinfrastrukturen . . . . . 12
2.3 Profile Manager . . . . . 12
2.4 Aspectix Profile Manager . . . . . 15
2.5 Zusammenfassung . . . . . 17

**3 Replizierte dezentrale adaptive Dienste** **19**
3.1 Objektorientierte Replikationsansätze . . . . . 21
3.2 Ansätze zur Replikationsunterstützung für verteilte objektorientierte Middleware Plattformen . . . . . 21
3.3 Fehlertolerante Replikation unter Verwendung des fragmentierten Objektmodels 23
3.3.1 Systemarchitektur der Fragmente . . . . . 24
3.3.2 Zusammenfassung . . . . . 26
3.4 Zeitgebundene Objektreferenzen . . . . . 26
3.4.1 Aktueller Stand der Forschung . . . . . 27
3.4.2 Basiskonzept . . . . . 30
3.4.3 Aktualisierung von zeitgebundenen Objektreferenzen . . . . . 30
3.4.4 Sicheres Binden von verteilten Objekten . . . . . 32
3.4.5 Integration eines Ortsdienstes . . . . . 36
3.4.6 Anwendungsfall: Referenzverwaltung eines dezentralen Versionsverwaltungssystems . . . . . 39
3.4.7 Verwandte Arbeiten . . . . . 40
3.4.8 Zusammenfassung . . . . . 41
3.5 Zustandstransfer . . . . . 41
3.5.1 Stand der Forschung . . . . . 42

3.5.2 Anwendungsschnittstelle zum Zustandstransfer . . . . . 46
3.5.3 Transferformat . . . . . 47
3.5.4 Ansätze zum parallelen Datentransfer . . . . . 50
3.5.5 Anpassung der Ansätze zum parallelen Datentranfer für den Zustandstransfer . . . . . 51
3.5.6 Verwendung von Synchronisationstechniken . . . . . 55
3.5.7 Zwischenspeichern von Teilzuständen . . . . . 57
3.5.8 Implementierungsübersicht . . . . . 58
3.5.9 Experimentelle Evaluierung . . . . . 59
3.5.10 Zusammenfassung . . . . . 64
3.6 Deterministisches Multithreading . . . . . 66
3.6.1 Deterministisches Threadscheduling . . . . . 66
3.6.2 Unterstützung zur Codetransformation . . . . . 67
3.6.3 Zusammenfassung . . . . . 67
3.7 Semantische Annotation als Mittel zur flexiblen Implementierung . . . . . 68
3.7.1 Stand der Forschung . . . . . 69
3.7.2 Semantische Annotationen . . . . . 70
3.7.3 Angepasste Codegenerierung und Anfragebearbeitung . . . . . 71
3.7.4 Zusammenfassung . . . . . 72
3.8 Basislaufzeitinfrastuktur . . . . . 72
3.9 Performanceevaluierung . . . . . 74
3.9.1 Evaluierung von performancerelevanten Annotationen . . . . . 74
3.9.2 Evaluierung der deterministischen Mutltithreadingunterstützung . . . . 75
3.10 Demonstrator: Ein repliziertes dezentrales Versionskontrollsystem . . . . . 77
3.10.1 Grundkonzepte von Git . . . . . 79
3.10.2 Operationen auf dem lokalen Repository . . . . . 81
3.10.3 Operationen auf dem verteilten Repository . . . . . 81
3.10.4 Zusammenfassung . . . . . 83
3.11 Zusammenfassung . . . . . 84

**4 Peer-to-Peer basierte dezentrale adaptive Dienste 87**
4.1 Verwandte Ansätze . . . . . 89
4.2 JXTA Übersicht . . . . . 89
4.2.1 Peers und Peer Gruppen . . . . . 89
4.2.2 Ressourcenbeschreibung in JXTA . . . . . 90
4.2.3 Nachrichten und Kommunikationsmechanismen . . . . . 90
4.2.4 Ressourcensuche in JXTA . . . . . 90
4.2.5 Architektur der JXTA Referenzplattform . . . . . 91
4.2.6 Zusammenfassung . . . . . 92
4.3 Integration von JXTA Diensten unter Verwendung des fragmentierten Objektmodells . . . . . 92
4.4 JXTA Profile Manager . . . . . 94
4.5 Demonstrator: Mehrbenutzer-Chat-Anwendung . . . . . 94
4.6 Zusammenfassung . . . . . 95

**5 Ausführungsumgebung für dezentrale adaptive Dienste 97**
5.1 Stand der Forschung . . . . . 99
5.1.1 Verteilte Middleware . . . . . 99
5.1.2 Middleware mit Unterstützung von mobilem Programmcode . . . . . 101
5.1.3 Allgemeine Verteilungsplattformen . . . . . 102
5.1.4 Zusammenfassung . . . . . 103
5.2 Infrastruktur auf Domänenebene . . . . . 104
5.3 Infrastruktur zur domänenübegreifenden Verteilung . . . . . 106
5.4 Integration dezentraler adaptiver Dienste . . . . . 107
5.5 Systemübersicht . . . . . 108
5.6 Zusammenfassung . . . . . 109

**6 Dezentrales, dynamisches Laden von plattformspezifischem Progammcode 111**
6.1 Stand der Forschung . . . . . 112
6.2 Anforderungen und Eigenschaften für die Code Auswahl . . . . . 114
6.3 Basisinfrastruktur . . . . . 115
6.4 Datenstrukturen zur Beschreibung und Verwaltung von Code . . . . . 115
6.5 Workflow zur Publikation, Auswahl und dem Transfer von Code . . . . . 117
6.6 Dynamische Suche und Integration von JXTA-Diensten . . . . . 117
6.6.1 Suche und Integration von Diensten unter Verwendung von Ressourcenbeschreibungen . . . . . 118
6.6.2 Mängel des dynamischen Ladeprozesses von JXTA . . . . . 119
6.7 Dezentrales dynamisches Laden von Code unter Verwendung von JXTA . . . . 119
6.7.1 Erweiterung der Ressourcenbeschreibung . . . . . 119
6.7.2 Verteiltes Implementierungsrepository . . . . . 120
6.7.3 Codeanbieter . . . . . 121
6.7.4 Ladedienst . . . . . 122
6.8 Integration der Infrastruktur zum dynamischen Laden in Aspectix und EDAS . 123
6.9 Demonstrator: Erweiterung der Mehrbenutzer-Chat-Anwendung . . . . . 124
6.10 Zusammenfassung . . . . . 125

**7 Ressourcenverwaltung 127**
7.1 Ressourcen und Ressourcenbeschränkung . . . . . 128
7.2 Infrastruktur zur Beobachtung und Protokollierung von Ressourcennutzung . 128
7.2.1 Basisunterstützung zur Ressourcenbeobachtung . . . . . 129
7.2.2 Infrastruktur zur Ressourcenkontrolle . . . . . 129
7.3 Verwendung von Quotas zur Begrenzung der Ressourcennutzung . . . . . 130
7.3.1 Diffusionsbasierte Lastverteilung . . . . . 131
7.3.2 Diffusionsbasiertes Quotamanagement . . . . . 132
7.3.3 Basiskonzept . . . . . 132
7.3.4 Protokoll . . . . . 133
7.3.5 Simulation . . . . . 140
7.3.6 Quotamanagement im Kontext von EDAS . . . . . 142
7.3.7 Verwandte Arbeiten . . . . . 143

7.3.8 Zusammenfassung . . . . . 143
7.4 Dezentrale Anwendungsverteilung . . . . . 143
7.4.1 Stand der Forschung . . . . . 144
7.4.3 Dynamische Ermittlung der global verfügbaren Ressourcen . . . . . 154
7.4.4 Zuweisung von Ressourcen . . . . . 155
7.4.5 Bereitstellung von Ortsinformationen . . . . . 155
7.4.6 Verteilung von Diensten . . . . . 155
7.4.7 Zusammenfassung . . . . . 157
7.5 Zusammenfassung . . . . . 158

**8 Zusammenfassung 159**
8.1 Kernpunkte des wissenschaftlichen Beitrags . . . . . 160
8.2 Limitationen der Arbeit und Ausblick . . . . . 161

# Einleitung

In den vergangenen Jahren wurden zahlreiche Forschungsinitiativen gestartet, die sich mit dem exponentiellen Wachstum der Anzahl von Rechensystemen, Rechenleistung und vor allem auch Information beschäftigten. Verwaltbarkeit und die bessere Ausnutzung von Ressourcen bilden die Hauptziele dieser Unternehmungen.

Verwaltbarkeit hat verschiedene Ausprägungen, zielt aber in der Regel darauf ab von der Komplexität der verwalteten Systeme zu abstrahieren, so dass Nutzer sich darauf konzentrieren können, was sie erreichen wollen und nicht wie dies geschehen soll. Aktuelle Forschungsprojekte von großen Hard- und Software-Herstellern wie Sun Microsystems, HP und Microsoft beschäftigen sich deshalb mit der Frage wie einzelne Systeme, aber auch ganze Serverlandschaften sich autonom administrieren können. Dieses Ziel beinhaltet sehr komplexe Problemstellungen wie die Erkennung von Fehlern und ihre automatische Behebung, sowie eine automatische Ressourcenverwaltung und Optimierung von Systemen zur Laufzeit.

Neben der Verwaltbarkeit spielt die Verbesserung der Ressourcennutzung eine sehr wichtige Rolle, da mit existierenden Mitteln unter Umständen komplexere Aufgaben gelöst werden können ohne dass zusätzliche Maßnahmen notwendig sind. Ein sehr gutes Beispiel hierfür bildet die Wiederentdeckung des *Peer-to-Peer Computing*. Ausgangspunkt dieser Entwicklung bildete 1999 Napster, ein Programm zum verteilten Austausch von Musikdateien. Bis heute bildet Peer-to-Peer Computing eines der aktivsten Themen im Forschungsbereich verteilte Systeme. Neben dem Tausch von geschützter Musik, welches oft als zentrale Triebfeder gesehen wird, ermöglicht es Benutzern ungenutzte Ressourcen einem Projekt oder einer Gemeinschaft zur Verfügung zu stellen. Ein prominentes Beispiel bildet hier SETI [SWB+97], das die Suche nach außerirdischem Leben zum Ziel hat, aber auch andere Projekte wie zum Beispiel ePost, einem Peer-to-Peer basierten E-Mail System [MPHD06], belegen die Anwendbarkeit von Peer-to-Peer Mechanismen. Die Grundidee von Peer-to-Peer Computing im Gegensatz zum herkömmlichen Klient/Server Ansatz ist, dass nun alle Teilnehmer gleichermaßen Information anbieten und abrufen können. Die Nachteile eines solchen Vorgehens sind die Konsequenzen einer üblicherweise hohen Fluktuation von Teilnehmern, da durch fortlaufende Änderung der teilnehmenden Rechner sich auch das Ressourcenangebot verändert. Aktuelle Systeme adressieren diese Problematik durch massive Replikation von Daten und die konsequente Dezentralisierung aller Aufgaben. Die Dynamik der Systemteilnehmer ist aber nur ein Teilproblem. Viel zentraler ist die Frage wie Peer-to-Peer basierte Systeme gesichert werden können, da in der Regel jeder Teilnehmer potentiell die Möglichkeit hat das System anzugreifen [Dou02, SCRD04]. Aktuell wird erforscht wie man diese und verwandte Probleme beheben kann, dennoch ist festzustellen, dass schon jetzt Peer-to-Peer Applikationen, vor allem für die Verteilung von Dateien, weit verbreitet sind.

Ein weiterer Trend in verteilten Systemen bildet *Grid Computing*, welches darauf abzielt aus einer beliebigen Anzahl von Rechnern und Speichersystemen einen großen Verbund zu bilden. Zielsetzung sind hier die verbesserte Ressourcennutzung durch einen einheitlichen Zugriff auf alle Systeme, sowie die automatische Abwicklung von rechen- und datenintensiven Applikationen. Erweiterte Unterstützung für Ressourceverwaltung ist hierbei essentiell, da es sonst nicht möglich ist effizient und automatisch komplexe Applikation mit zahlreichen Teilaufgaben auf ein Grid zu verteilten. Grid Infrastrukturen werden häufig innerhalb eines Unternehmens genutzt, es gibt aber auch Beispiele von weltweiten Grids die zahlreiche Firmen und Institutionen verbinden. Im Gegensatz zu Peer-to-Peer Computing, welches bisher in der Regel die langfristige Bereitstellung einer Anwendung zum Ziel hat sind Infrastrukturen für Grid-Computing wie das Globus Toolkit [Fos05] oder Condor [TTL02] darauf ausgerichtet Unterstützung für die dynamische Verteilung und Ausführung von großen Anwendungen, die sich aus vielen kleineren und mittleren, zeitbefristeten Teilaufgaben zusammensetzen, zu bieten. Um dies zu erreichen ist es jedoch nötig eine Vielzahl von verschiedenen Diensten aufzusetzen und zu warten. Ein Weg um diesen Aufwand zu reduzieren würde die Einführung von Peer-to-Peer Mechanismen bieten, da hier der Fokus auf der Dezentralisierung von Aufgaben über alle teilnehmenden Systeme liegt, was den administrativen Aufwand reduziert und das Gesamtsystem fehlertoleranter macht.

Analog zu den Entwicklungen im Grid-Computing, die auf die Ausführung von daten- und rechenintensiven Aufgaben ausgerichtet sind gibt es einen Trend grid-ähnliche Infrastrukturen zu entwickeln, welche den Betrieb beliebiger Applikationen ermöglichen. Eine der grundsätzlichen Unterschiede zwischen den jetzigen Grid Computing Infrastrukturen und diesen aktuell entwickelten Plattformen ist die Laufzeit von Applikationen. Traditionelle Grid-Applikationen besitzen auf Grund ihrer Zielsetzung eine begrenzte Laufzeit. Dies ist in der Regel nicht gegeben für beliebige Applikation. Hier kann die Laufzeit wesentlich länger, wenn nicht sogar unbeschränkt sein. Dies erfordert jedoch eine andere Art von Ressourcenverwaltung als bisher von Grid Computing Infrastrukturen bereitgestellt wird.

Als einer der ersten Vertreter einer solchen allgemeineren Infrastruktur kann PlanetLab [PACR03] gesehen werden, welches die Zielsetzung hat eine global verteilte Testumgebung zur Entwicklung von verteilten Applikationen bereit zu stellen. Eine weitere solche Plattform bildet Xenoserver [HHaIP03]. Beide Systeme zielen auf die Verteilung von beliebigen Applikationen über eine beliebige Anzahl von weltweit verteilten Rechnern ab. Sowohl Xenoserver als auch PlanetLab unterstützen dies, indem sie Applikationen ganze Betriebssysteminstanzen bereitstellen, wie dies zum Beispiel durch Xen [BDF+03] ermöglicht wird. Dies bildet die Grundlage für die transparente Verteilung und Ausführung von beliebigen existierenden Applikationen. Solche Plattformen sind jedoch ähnlich wie Peer-to-Peer Systeme mit einer gewissen Fluktuation der Teilnehmersysteme konfrontiert (z.B. durch Rechnerausfälle) und zusätzlich ist der Ressourcenbedarf der betrieben Anwendungen auf Grund von Benutzerzugriffen schwankend. Dies erfordert die Umverteilung von langlaufenden Applikationen je nach Ressourcenangebot sowie Intensität der Benutzerzugriffe. Aufgrund dieser Umstände wird es in der Regel erforderlich sein, dass Applikationen ihrer Verteilung und etwaigen Ortswechseln gewahr sind. Die direkte Konsequenz ist, dass existierende Anwendungen zumindest für Verteilung angepasst werden müssen. Es kann al-

so davon ausgegangen werden, dass es nicht ausreicht neue Infrastrukturen zu entwickeln, sondern auch Dienste bzw. sogar Dienstmodelle, welche die Mechanismen ausnutzen und von ihnen profitieren können

Obwohl es bereits erste Infrastrukturen für den Betrieb von beliebigen verteilten Applikationen gibt, die zum Teil auch wie im Fall von PlanetLab schon im Betrieb sind, hat die Entwicklung solcher Systeme erst begonnen. Existierende Systeme beantworten die folgenden Fragen bestenfalls teilweise:

- Wie sieht eine geeignete Architektur für solche Systeme aus?
- Wie sind Ressourcen für langlaufende Dienste bereitzustellen und zu verwalten?
- Wie kann man Anwendungen ausbringen und unter welchen Umständen erfolgt eine Umverteilung?
- Werden neue Dienstmodelle und Applikationsansätze benötigt? Wie sehen sie aus?

## Ziele der Arbeit

Ziel dieser Arbeit ist es alle im vorangegangenen Abschnitt aufgeworfenen Fragen zu bearbeiten und zumindest teilweise zu beantworten. Im Gegensatz zu PlanetLab und Xenoserver soll im Rahmen dieser Arbeit keine Plattform für beliebige existierende Anwendungen geschaffen werden, da der Autor die Meinung vertritt, dass es zwingend erforderlich ist, dass Anwendungen ihrer Verteilung und etwaigen Umverteilungen gewahr sind. Folglich sollten Ausführungsplattform und Anwendungen aufeinander abgestimmt sein.

Um dies zu erreichen und die Entwicklung entsprechender verteilungsgewahrer Dienste zu erleichtern müssen Dienstmodelle und Entwicklungsunterstützung angeboten werden. Jedoch ist es nicht nur nötig, dass Dienste über ihre Verteilung Bescheid wissen, sondern es muss ihnen auch möglich sein diese aktiv zu beeinflussen. Im Rahmen der Arbeit wird die dynamische Verteilung von Diensten als Form der Adaption gesehen um veränderte Anforderungen des Dienstes und der Ausführungsumgebung zu adressieren. Anforderungsänderungen des Dienstes sind in der Regel verursacht durch wechselnde Dienstnutzung, können aber auch andere Ursachen haben, wie zum Beispiel die Rekonfiguration des Dienstes. Anforderungsänderungen der Ausführungsumgebung begründen sich normalerweise in wechselnder Verfügbarkeit von Ressourcen. So kann es zum Beispiel sein, dass Wartungsarbeiten an einem System vorgenommen werden müssen und es deshalb für eine gewisse Zeit nicht zur Verfügung steht. Es kann davon ausgegangen werden, dass sowohl Dienstanforderungen als auch Anforderungen der Ausführungsumgebung sich fortlaufende ändern, unter Umständen sogar periodisch. Zusammenfassend werden Dienste im Rahmen dieser Arbeit als *adaptive Dienste* bezeichnet, wenn sie ihrer Verteilung gewahr sind und diese entsprechend interner und externer Anforderungen dynamisch anpassen können.

Ausfälle von einzelnen Rechnern sind in großen verteilten Systemen an der Tagesordnung und müssen von langlaufenden Diensten toleriert werden können. In der Regel wird dies erreicht durch Replikation. Je nach verwendetem Ansatz können Rechnerausfälle dennoch den Verlust von Daten oder den kurzzeitigen Ausfall des Dienstes bedeuten. In der

Langzeitperspektive bleibt der Dienst jedoch verfügbar. Im Rahmen der Arbeit wird davon ausgegangen, dass alle Dienste eine gewisse Anzahl von Knotenausfällen tolerieren können. Diese Anforderung wird durch den Begriff des *dezentralen Dienstes* erfasst. Ein solcher Dienst ist über mehrere Rechner verteilt und verwendet Replikation um einzelne Rechnerausfälle tolerieren zu können.

Nimmt man die beiden vorgestellten Anforderungen zusammen ist es das Ziel dieser Arbeit Dienstmodelle und Konzepte zur Entwicklung und zum Betrieb von *dezentralen adaptiven Diensten* bereitzustellen. Diese Dienste sind in der Lage eine Reihe von Rechnerausfällen zu tolerieren und können ihre Verteilung gemäß eigener als auch externer Erfordernisse anpassen. Dieses Ziel erfordert Techniken zur Replikationsunterstützung sowie Mechanismen zur dynamischen Verteilung.

Gleiches gilt für eine Infrastruktur, die den Betrieb von dezentralen adaptiven Diensten unterstützt. Sie sollte ebenfalls dezentral und adaptiv sein. In der Basis kann dies erreicht werden indem man die Infrastruktur auf dezentralen adaptiven Diensten aufbaut. Jedoch ist die Bereitstellung zusätzlicher Mechanismen nötig. Eine essentielle Anforderung für die verteilte Ausführung von Diensten bildet die dynamische Bereitstellung von Programmcode am Ausführungsort, da man nicht davon ausgehen kann, dass jeglicher Code an allen Orten des verteilten Systems vorgefunden werden kann. Mindestens ebenso wichtig ist das Ressourcenmanagement in einem solchen System. Es erfordert die dynamische Bestimmung der Ressourcennutzung von Diensten, die Bereitstellung von Mechanismen zur Ressourcenzuteilung, sowie die Begrenzung ihrer Nutzung. Zusammengenommen bildet das Ressourcenmanagement die Basis für eine ressourcengewahre Verteilung von Diensten.

## Wissenschaftlicher Beitrag

Als Implementierungsbasis für dezentrale adaptive Dienste wurde das objekt-orientierte Entwicklungsparadigma gewählt, da es die Strukturierung und Entwicklung von beliebigen Applikationen ermöglicht und im speziellen für die Entwicklung von verteilten Anwendungen geeignet ist. Standard Middleware, welche das objekt-orientierte Entwicklungsparadigma unterstützen wie Common Object Request Broker Architecture (CORBA) [Obj04], .NET [MWN02], Java Remote Method Invocation (Java RMI) [Wal98] und die ICE Plattform [Hen04] werden häufig für die Entwicklung komplexer verteilter Systeme genutzt. Alle diese Plattformen sind auf die Entwicklung von Klient/Server Applikationen zugeschnitten, verfügen jedoch über verschiedene Mechanismen zur Modifikation der Interaktion zwischen Klient und Server. Weitergehende Unterstützung, wie Mechanismen zur Selbst-Optimierung oder Mechanismen für Skalierbarkeit, Fehlertoleranz oder Dienstqualität werden nicht angeboten. Um diese Anforderungen, die speziell in dynamischen Systemen auftreten zu adressieren, unterstützt das Aspectix Projekt die Entwicklung und den Betrieb von *fragmentierten Objekten* [Sha86, MGNS94]. Dies wurde für die CORBA Plattform durch einen flexible konfigurierbaren Referenzmanager erreicht, der von dem Aspectix Object Request Broker (ORB) [RHKS03, HKRS05] zur Verfügung gestellt wird. Ähnliche Unterstützung ist für die Java RMI Middleware durch die Entwicklung der FORMI-Bibliothek [KKSH05, KDH+06] verfügbar. Fragmentierte Objekte bieten weitreichende Unterstützung für echt verteilte Ob-

jekte. Dies geht deutlich über Anpassungen der Aufrufbehandlungen in einem Klient/Server System [BMV03, Obj01] hinaus und übersteigt auch die Fähigkeiten von Smart Proxies [KK00, SMS02] indem es einzelnen Objektanteilen möglich ist gleichberechtigt miteinander zu interagieren. Im Rahmen dieser Arbeit wurde der Aspectix ORB und die zugehörige Unterstützung für fragmentierte Objekte gewählt um dezentrale adaptive Dienste und eine Plattform für ihren Betrieb anzubieten. Jedoch können die erlangten Ergebnisse auch mit Hilfe von FORMI auf Java RMI übertragen werden und sind ebenfalls auf die .NET Infrastruktur übertragbar. Das fragmentierte Objektmodel, so wie es von dem Aspectix ORB unterstützt wird bietet die Möglichkeit beliebige verteilte Objekte zu implementieren, da es die interne Verteilungsstruktur eines Objektes nicht festlegt. Ziel dieser Arbeit ist es basierend auf dieser rudimentären Unterstützung Mechanismen für die Entwicklung von dezentralen adaptiven Diensten bereitzustellen. Dezentrale adaptive Dienste können so nahtlos in Standard Middleware Infrastrukturen einpassen werden. Dies ermöglicht es ein traditionell entwickeltes Klient/Server Objekt durch einen dezentralen adaptiven Dienst, einfach durch Zuweisung einer Referenz, auszutauschen.

### Entwicklungsunterstützung für dezentrale adaptive Dienste

Zwei Arten von dezentralen adaptiven Diensten wurden im Rahmen dieser Arbeit entwickelt: Replizierte dezentrale adaptive Dienste und Peer-to-Peer basierte dezentrale adaptive Dienste. Diese Unterscheidung ermöglicht es den verschiedenen Anforderungen von Diensten bezüglich Fehlertoleranz und Skalierbarkeit gerecht zu werden.

#### Replizierte dezentrale adaptive Dienste

Replizierte dezentrale adaptive Dienste basieren im Kern auf dem Konzept der *aktiven Replikation* [Mul93] und bieten somit einen hohen Grad an Fehlertoleranz. Aktive Replikation geht davon aus, dass Objekte oder Dienste deterministisch über eine Menge von Rechnern repliziert werden können unter Verwendung eines Gruppenkommunikationssystems, welche die total geordnete Zustellung von Nachrichten ermöglicht. Im Gegensatz zu bisherigen Arbeiten [Maf95, MMSN98, Fel98, GNSY00, MMVB00, FH02, RBC+03] bietet das entwickelte Framework für replizierte dezentrale adaptive Dienste erweiterte Mechanismen für Dienstmobilität an um den Anforderungen einer global verteilten Ausführungsumgebung gerecht zu werden. Zusätzlich wird ein flexibles Programmiermodel, basierend auf semantischen Annotationen der Dienstschnittstelle angeboten welches, erweiterte Möglichkeiten zur Klient/Server Interaktion bietet.

Zwei grundsätzliche Probleme der Mobilität von Diensten sind ihre Lokalisierung und die Gewährleistung eines zuverlässigen Zugriffs trotz wiederholtem Ortswechsel. In der Regel sind diese Probleme durch einen allgemein bekannten und immer verfügbaren Ortsdienst oder die Weiterleitung von Anfragen bzw. einer Kombination aus beiden Ansätzen gelöst. Im Rahmen der vorliegenden Arbeit wurde mit der Entwicklung von zeitgebundenen Objektreferenzen ein neuer Ansatz entwickelt. Hierbei erhalten Referenzen eine Lebenszeit innerhalb derer garantiert ist, dass der referenzierte Dienst gebunden werden kann trotz der Migration einzelner Servicebestandteile. Weiterhin wurde das Konzept durch ein siche-

res Bindeverfahren ergänzt, so dass Dienste sicher gebunden werden, auch wenn frühere Ausführungsorte des Dienstes versuchen dies aktiv zu verhindern.

Existierende Replikationsinfrastrukturen behandeln im Wesentlichen nur die Kompensation von Rechnerausfällen durch die Instantiierung neuer Replikate. Grundsätzlich erfordert dies den Transfer des Zustandes von einem der existierenden Replikate zu dem neu beitretenden Knoten. Bisher wurde dies nur nötig wenn ein oder mehrere Replikate ausgefallen waren und sie deshalb ersetzt werden müssen. Für replizierte dezentrale adaptive Dienste ist dies jedoch nicht mehr ausschließlich der Fall, da es ihnen möglich ist ihre Verteilung dynamisch zu verändern. Es wird erwartet, dass dies weit häufiger der Fall ist als zur Kompensation von Ausfällen. Folglich behandelt diese Arbeit erweitere Mechanismen zur Erleichterung und Beschleunigung des Zustandstransfers. Die meisten existierenden Systeme [CM00] unterstützen einen blockierenden Ansatz bei dem alle Anfragen an einen replizierten Dienst so lange verzögert werden bis der Zustand komplett zu einem neuen Knoten übertragen ist. Weiterhin transferieren alle dem Autor bekannten Systeme den Zustand von genau einem Replikat zu dem neu beitretenden Knoten. Beide Vorgehensweisen liegen in der Annahme begründet, dass erstens die Integration eines neuen Knotens in einen Replikationsverbund ein seltener Fall ist und zweitens der auszutauschende Zustand relativ klein ist. Beide Annahmen sind nicht haltbar für replizierte dezentrale adaptive Dienste. Der Zustand eines Dienstes kann sehr groß sein wie das im Rahmen dieser Arbeit an Hand eines dezentralen verteilten Versionskontrollsystems gezeigt wird. Zusätzlich wird dynamische Verteilung als Maßnahme zur Adaption betrachtet, was den Zustandstransfer zu einer häufigen Aufgabe macht. Als Konsequenz wurden Techniken zum nicht-blockierenden und parallelen Zustandstransfer entwickelt und evaluiert. Der erste Ansatz verbessert die Verfügbarkeit eines Dienstes während der zweite Ansatz den Zustandstransfer beschleunigt und damit die Integration von neuen Replikaten. Die Konsequenz ist eine verbesserte Fehlertoleranz, da Ausfälle schneller kompensiert werden können.

Wie bereits zu Beginn des Abschnitts angemerkt ermöglicht die Integration des fragmentierten Objektmodels in eine Standard-Middleware die Entwicklung von Objekten, die eine beliebige interne Verteilung besitzen können. Wenn man nun nur die Unterstützung des fragmentierten Objektmodels genutzt hätte um replizierte dezentrale adaptive Dienste bereitzustellen wäre dies ein sehr inflexibler Ansatz gewesen, der die Vorteile von fragmentierten Objekten negiert. Im Kern entspricht dies dem Ansatz der meisten Replikationsinfrastukturen, welche lediglich die Erzeugung von erweiterten Stubs and Skeletons zur replizierten Ausführung von Anfragen unterstützen. Dies ist jedoch unzureichend um den Anforderungen von komplexen verteilten Diensten gerecht zu werden wie sich bei der Entwicklung eines dezentralen, verteilten Versionskontrollsystems bestätigt hat. Kern der Problematik bildet die Anforderung, dass trotz der verwendeten aktiven Replikation ein Bedarf nach einer flexiblen internen Kommunikation besteht. Als Konsequenz ist die Möglichkeit der semantischen Annotation der Dienstschnittstelle und ein entsprechender Codegenerationsprozess entwickelt worden. Dies ermöglicht Entwicklern, unter anderem, eigenen Code in den klientseitigen Stub des Dienstes einzubringen.

Zusammenfassend kann festgestellt werden, dass replizierte dezentrale adaptive Dienste, basierend auf aktiver Replikation, der erweiterten Unterstützung für Mobilität und dem be-

reitgestellten flexiblen Programmiermodell die Möglichkeit bieten komplexe fehlertolerante Dienste für dynamische Umgebungen zu entwickeln.

### Peer-to-Peer basierte dezentrale Dienste

Während aktive Replikation die konkurrierende Modifikation von Daten durch mehrere Benutzer eines fehlertolerant Dienstes gut unterstützt ist dieser Ansatz nicht optimal geeignet um große Datenmengen, für sehr viel lesende Zugriffe, bereitzustellen. Anwendungen mit diesem Zugriffsprofil wie zum Beispiel Dateitausch oder jegliche Art von Namens- und Ortsdiensten werden mittlerweile häufig mit Hilfe von Peer-to-Peer Protokolle implementiert, weil sie existierende Ressourcen nutzen und in der Regel skalierbar sind. Als Konsequenz dieses Bedarfs wurde das Konzept der Peer-to-Peer basierten dezentralen Dienste entwickelt. Hierbei handelt es sich um die Integration der Peer-to-Peer Middleware JXTA unter Verwendung des fragmenierten Objetmodells. Hier ist der Beitrag nicht mit der Entwicklung einer neuen Infrastruktur gegeben sondern einerseits in der Ergänzung der CORBA Plattform durch Peer-to-Peer Mechanismen andererseits aber auch die Ergänzung der JXTA Middleware durch Konzepte einer Middleware die entfernte Methodenaufrufe unterstützt und damit ein deutlich höheres Abstraktionsniveau anbietet. Die von JXTA zur Verfügung gestellten Mechanismen ermöglichen die replizierte Bereitstellung von Diensten die unter einer gemeinsamen logischen Adresse verfügbar sind und damit auf dieser Ebene Dezentralisierung und Fehlertoleranz bieten. Auch Unterstützung für Adaption wird durch JXTA geboten da es Diensten die Möglichkeit bietet den Ort zu wechseln und Anfragen entsprechend an den neuen Aufenthaltsort weiterleitet. Die vorgestellte Integration von JXTA mittels des fragmentierten Objektmodels bildet einen leichtgewichtigen Ansatz, der nicht auf CORBA beschränkt ist sondern auf alle anderen Middleware Systeme, die fragmentierte Objekte unterstützen, übertragen werden kann.

### Infrastrukturunterstützung für dezentrale adaptive Dienste

Nimmt man die beiden vorgestellten Dienstmodelle zusammen ist es nun möglich Dienste zu entwickeln, die einer Vielzahl Anforderungen gerecht werden und in dynamischen Umgebungen erfolgreich betrieben werden können. Dies zeigt auch die entwickelte Infrastruktur *Environment for Decentralised and Adaptive Services (EDAS)*, welche die Ausführung und Verteilung von dezentralen adaptiven Diensten über mehrere administrative Domänen hinweg unterstützt. Kernkomponenten der Ausführungsumgebung bildet das *Home Environment*, welches die Ressourcen aller Rechner einer Domäne verwaltet und das *Service Environment*, welches als verteilte Ausführungsumgebung für die Dienste eines Projektes oder Gruppierung dient und sich über mehrere Domänen erstrecken kann.

Wenn man dezentrale adaptive Dienste und fragmentierte Objekte nutzt erfordert der Bindevorgang in der Regel das dynamische Laden von objektspezifischem Code. Dies ist nötig, da man einerseits nicht jeglichen existierenden Code auf allen Rechnern innerhalb eines großen verteilen Systems installieren kann, andererseits aber auch die Ausführungsorte einer Applikation nicht im Vorfeld kennt. Dieses Problem verstärkt sich im Kontext von EDAS, da hier einzelne Rechner oder sogar Gruppen von Rechnern dem Gesamtsystem dynamisch

beitreten können. Um dennoch die dynamische Instanziierung und Verteilung von dezentralen adaptiven Diensten zu gewährleisten bietet EDAS eine auf JXTA basierende Infrastruktur zum dynamischen Laden von Code, wobei wesentliche Schritte wie die Suche und das Laden von Code verteilt realisiert sind. Dies ist, neben der Verbesserung des Selektionsprozesses, einer der geeigneten Implementierung der wichtigsten Unterschiede zu bisherigen Arbeiten des Autors [KH03]. Im Gegensatz zu verwandten Arbeiten [PKF05, SM05, PC03] ist das entwickelte System nicht auf das Laden von Code einer bestimmten Implementierungssprache beschränkt und verfolgt einen dezentralen Ansatz.

Wie zu Beginn der Arbeit motiviert ist die Verwaltung von Ressourcen ein wichtiger Bestandteil von Infrastrukturen zur Ausführung von verteilten Diensten. Dies ist im speziellen der Fall für EDAS und den Betrieb dezentraler adaptiver Dienste. Es wurde betrachtet wie Ressourceninformationen innerhalb einer virtuellen Maschine für Java erfasst und beschränkt werden können. Ausserdem wurde eine Infrastruktur entwickelt, die dies unter Verwendung von existierenden Mechanismen der Sun Java Virtual Machine (Sun JVM) 1.5 für alle laufenden virtuellen Maschinen eines Rechners unterstützt.

Basierend auf der vorgestellten Infrastruktur zur Beobachtung und Protokollierung der Nutzung von Ressourcen wurde ein verteiltes Quotamanagement Protokoll entwickelt, welche die globale Begrenzung der Nutzung einer Ressource ermöglicht. Im Gegensatz zu bisherigen Ansätzen (z.B. durch *Leases*) ist das vorgestellte Protokoll skalierbar, fehlertolerant und reagiert adaptiv auf Anforderungsänderungen.

Neben der Ermittlung von Ressourcennutzung und ihrer Beschränkung ist die Unterstützung zur dynamischen Verteilung von Diensten wichtig. Aus diesem Grunde wurden aktuelle Ansätze aus den Gebieten Grid Computing, geschlossene Infrastrukturen wie Serverfarmen, sowie offene Systeme wie zum Beispiele Ansätze im Kontext von PlanetLab und XenoServer betrachtet. Basierend auf den Erkenntnissen und den Anforderungen von EDAS wurde ein eigenes Protokoll zur Platzierung von Diensten, basierend auf zugewiesenen Kontingenten fester Größe entwickelt. Im Gegensatz zu verwandten Arbeiten im Bereich von offenen Systemen [SH03, OAPA05] ermöglicht diese eine schwache Form von Ressourcengarantien und integriert eine Beschränkung der Ressourcennutzung.

## Gliederung

Kapitel 2 gibt einen kompakten Überblick der für die Arbeiten wichtigen Mechanismen der Aspectix Middleware. Hier sind vor allem die flexible Behandlung von Referenzen und die Unterstützung des fragmentierten Objektmodels zu nennen.

Kapitel 3 behandelt replizierte dezentrale adaptive Dienste eines der beiden in dieser Arbeit vorgestellten Dienstmodelle. Zu Beginn widmet sich das Kapitel der Betrachtung verwandter Ansätze und stellt dann das entwickelte Framework zur fehlertoleranten Replikation von Diensten basierend auf dem fragmentierten Objektmodell vor. Letzteres bildet die Basis für die Vorstellung weiterer entwickelter Mechanismen zur Unterstützung der Migration von Diensten wie zum Beispiel zeit-gebundene Referenzen und Techniken zum schnellen Zustandstransfer. Für die effiziente Implementierung von replizierten dezentralen adaptiven Diensten unterstützt das Framework deterministisches Multithreading und bietet basie-

rend auf Annotation an der Schnittstelle des Dienstes ein flexibles Programmiermodell an. Um die Erzeugung und den Betrieb von replizierten dezentralen Diensten zu ermöglichen wurde des Weiteren eine Basisinfrastruktur entwickelt, welche im Folgenden erörtert wird. Das Kapitel schließt mit einer Performanceevaluierung und der Vorstellung eines Demonstrators, einem dezentralen Versionsverwaltungssystems ab.

Kapitel 4 beginnt mit einer kurzen Einführung in die JXTA Middleware. Im Anschluss wird beschrieben wie JXTA-Dienste unter Verwendung des fragmentierten Objektmodells in die Aspectix Middleware integriert werden können. Dies ermöglicht die Bereitstellung des zweiten vorgeschlagenen Dienstmodells: Peer-to-Peer basierte dezentrale adaptive Dienste. Den Abschluss des Kapitels bildet wiederum die Vorstellung eines Demonstrators, eines Mehrbenutzer-Chat Dienstes.

Kapitel 5 behandelt die Architektur von EDAS und den genauen Aufbau durch die Verwendung der beiden vorgeschlagenen Dienstmodelle. EDAS ermöglicht die Verteilung von Diensten über mehrere administrative Domänen hinweg.

Kapitel 6 beschäftigt sich mit dem dynamischen Laden von Code für verteilte Applikationen in einem heterogenen Umfeld im Allgemeinen und für dezentrale adaptive Dienste im Speziellen. Ausgangspunkt bildet die Fragestellung wie Programmcode beschrieben werden muss damit man ihn schnell, effizient und skalierbar in einem verteilten System suchen, auswählen und landen kann. Der anfänglich generelle Vorschlag für eine Infrastruktur mündet dann in eine konkrete Implementierung basierend auf JXTA.

Kapitel 7 beschreibt wie Ressourcen von EDAS verwaltet werden damit dezentrale adaptive Dienste ressourcengewahr verteilt werden können. Zu Beginn des Kapitels wird beschrieben welche Ressourcen betrachtetet werden können und auf welchen Ebenen sie verwaltet werden müssen. Im Folgenden wird dann eine Infrastruktur zur Erfassung der Ressourcennutzung auf einem Rechner vorgestellt. Dies bildet die Basis für verteiltes Ressourcenmanagement. Zuerst wird ein Protokoll zur Einhaltung globaler Ressourcenlimits vorgestellt. Danach ein Ansatz beschrieben wie Dienste ressourcegewahr verteilt werden können.

Kapitel 8 resümiert die vorgestellten Kernbeiträge dieser Arbeit und befasst sich mit den Limitierungen der präsentierten Ansätze sowohl auf Konzeptioneller- als auch auf Implementierungsebene. Abschließend werden mögliche Erweiterungen diskutiert.

# Zusammenfassung

Das Ziel dieser Dissertation war es die in der Einleitung aufgeworfenen Fragen zu beantworten:

- Werden neue Dienstmodelle und Applikationsansätze benötigt? Wie sehen sie aus?
- Wie sieht eine geeignete Architektur für solche Systeme aus?
- Wie sind Ressourcen für langlaufende Dienste bereitzustellen und zu verwalten?
- Wie kann man Anwendungen ausbringen und unter welchen Umständen erfolgt eine Umverteilung?

Die letzte der vier Fragen, ob neue Dienstmodelle und Applikationsansätze benötigt werden und wie sie aussehen könnten, wurde zuerst behandelt. Das Ergebnis sind dezentrale adaptive Dienste als eine flexible und verteilte Alternative zu traditionellen objektorientierten Klient/Server Diensten. Den Ausgangspunkt der Arbeit bildeten der Aspecitx ORB und seine Unterstützung für das fragmentierte Objektmodel. Basierend auf dieser Grundlage wurden die beiden Dienstmodelle replizierte dezentrale adaptive Dienste und Peer-to-Peer basierte dezentrale adaptive Dienste entwickelt. Die Verwendbarkeit der beiden Ansätze wurde am Beispiel zweier Dienste eines verteilten Versionskontrollsystems und einer Mehrbenutzer-Chat Anwendung gezeigt.

Die EDAS Infrastruktur unterstützt die verteilte Ausführung und das Management von CORBA Diensten im Allgemeinen und im Speziellen von dezentralen adaptiven Diensten. Dies wird erreicht durch die beiden vorgeschlagenen Komponenten Home Environment und Service Environment. Die Aufgabe des Home Environments ist es alle Ressourcen einer administrativen Domäne zusammenzufassen und zu verwalten. Ziel des Service Environment ist es von verschiedenen Home Environments angebotene Ressourcen zu vereinigen um eine verteilte Ausführung von Diensten über mehrere administrative Domänen hinweg zu ermöglichen. EDAS bildet somit die Antwort auf die Frage wie die Architektur für ein System zur Ausführung von langlaufenden, verteilten Diensten aussehen kann.

Die sich daraus ergebende Basisarchitektur wurde im weiteren Verlauf der Arbeit um das dynamische Laden von plattformspezifischem Code und Mechanismen zur verteilten Ressourcenverwaltung ergänzt. Letzteres kann in drei wichtige Teilaufgaben gegliedert werden: Beobachtung von Ressourcenverbrauch und dessen Begrenzung auf Rechnerebene; verteilte Begrenzung von Ressourcenverbrauch und die Unterstützung für verteiltes Ressourcenmanagement einschließlich einer ressourcengewahren dynamischen Verteilung von Diensten. Die erste Erweiterung beantwortet die Frage wie man Anwendungscode in einem solchen Umfeld bereitstellen kann. Die zweite Erweiterung behandelt die Frage nach dem Ressourcenmanagement für langlaufende Dienste.

## Kernpunkte des wissenschaftlichen Beitrags

Aktive Replikation und ihre Verwendung zur fehlertoleranten Implementierung von objektorientierten Diensten ist bereits in verschiedenen Projekten Gegenstand der Forschung gewesen. Im Gegensatz zu diesen Arbeiten basiert das entwickelte Framework für replizierte dezentrale adaptive Dienste auf dem Konzept der fragmentierten Objekte im Kontext der Standardmiddleware CORBA. Dies ermöglicht eine für die Anwendung transparente Bereitstellung von fehlertoleranten Diensten und ist zudem portabler als existierende transparente Ansätze. Weiterhin bietet das Framework im Vergleich zu existierenden Systemen die Möglichkeit mittels semantischer Annotationen der Objektschnittstelle, eine optimierte Behandlung von Dienstanfragen, sowie die Integration von dienstspezifischem Code auf Aufruferseite. Dies ermöglicht die Entwicklung von komplexen fehlertoleranten Diensten wie am Beispiel eines verteilten Versionsverwaltungssystems gezeigt.

Um es replizierten dezentralen adaptiven Diensten zu ermöglichen ihre Verteilung dynamisch zu verändern wurden neue Techniken zur Migrationsunterstützung entwickelt. Das vorgestellte Konzept der zeit-gebundenen Objektreferenzen gewährleistet das sicherer Binden von replizierten dezentralen adaptiven Diensten unabhängig von einem global verfügbaren Ortsdienst oder dem Weiterleiten von Aufrufen durch entsprechende Stellvertreter. Vorraussetzung für die Migration eines existierenden Replikats oder auch die Bereitstellung eines neuen Replikats ist ein Transfer des Dientszustandes. Diese Operation kann wie im Rahmen der Evaluierungen gezeigt wurde bei großen Zuständen unnötig lange dauern und reduziert damit die Verfügbarkeit des Dienstes sowie in gewissem Maße seine Fehlertoleranz. Es wurde deshalb eine Schnittstelle zum datenstrombasierten Zustandstransfer vorgeschlagen und Algorithmen zum parallelen Transfer entwickelt. Beide Maßnahmen beschleunigen den Zustandstransfer speziell bei großen Applikationszuständen.

Peer-to-Peer basierte dezentrale adaptive Dienste bieten im Gegensatz zu verwandten Arbeiten eine einfache und aus Sicht der Anwendung transparenten Integration peer-to-peer Mechanismen in CORBA und CORBA-basierte Anwendungen. Der Beitrag liegt hier in der Abbildung von JXTA Diensten auf eine CORBA Schnittstelle unter Verwendung des fragmentierten Objektmodels und der Bereitstellung eines eigens entwickelten Profilemanagers.

Die EDAS Architektur übernimmt konzeptionelle Ideen einer Grid Infrastruktur, bietet sie jedoch im Rahmen einer objektorientierten Middleware an und ist auf die Ausführung von langlaufenden Diensten ausgerichtet. EDAS basiert auf dezentralen adaptiven Diensten, was auf Grund ihrer Eigenschaften dem System eine gewisse Flexibilität und Fehlertoleranz verleiht.

Basierend auf der EDAS Basisinfrastruktur wurden ergänzend Dienste zum dezentralen und dynamischen Laden von plattformspezifischem Code bereitgestellt. Im Vergleich zu verwandten Systemen sind alle wichtigen Teilschritte des Ladevorgangs dezentralisiert worden und es werden explizit heterogene Systeme betrachtet und berücksichtigt. Die erstellte Implementierung ergänzt und verbessert existierende Mechanismen der JXTA Plattform.

Abschließend wurde die EDAS Infrastruktur um Mechanismen zum Ressourcenmanagement erweitert. Hier sind zwei wichtige Beiträge zu nennen: Es wurde ein dezentrales Protokoll zur Einhaltung globaler Ressourcenlimits entwickelt. Dieses nutzt Ideen aus dem Bereich der dezentralen Lastverteilung und ist skalierbar und fehlertolerant. Weiterhin wurde

ein Protokoll zur ressourcegewahren Verteilung von Diensten entworfen. Es führt Ressourcegebinde verschiedener Grösse ein, die den unterschiedlichen Ressourcenanforderungen von typischen Diensten gerecht werden und nutzt die vorgeschlagenen Konzepte Home und Service Envrionement um eine effiziente Ortssuche zu ermöglichen. Beide Maßnahmen zusammengenommen bildet das vorgeschlagene Protokoll eine Alternative zu existierenden Ansätzen, da es Ressourcenkontrolle und Platzierung effizient miteinander kombiniert.

## Limitationen der Arbeit und Ausblick

Obwohl alle der am Anfang der Arbeit aufgeworfenen Fragen bearbeitet und beantwortet wurden bleibt anzumerken, dass die Fragen aus einem objekt-orientierten Middleware Kontext beantwortet wurden und das es zu jeder dieser Fragen viele verschiedene Antworten gibt.

Die beiden Dienstmodelle für dezentrale adaptive Dienste wurden implementiert und evaluiert. Auch die EDAS Infrastruktur ist in weiten Teilen verfügbar, dies gilt auch für die Unterstützung der Ressourcenkontrolle und dem dynamischen Laden von Code. Was an dieser Stelle noch aussteht ist eine Implementierung und Evaluierung des Protokolls zur dynamischen Verteilung von Diensten. Hier ist weiterer Forschungsbedarf gegeben.

Verschiedene Beiträge dieser Arbeit, wie zum Beispiel die Unterstützung zum dynamischen Laden von Code oder das Protokoll zum Management von globalen Quotas, sind unabhängig von dezentralen adaptiven Diensten und der EDAS Plattform. Folglich wäre ihre Evaluierung auch in anderen Systemkontexten sinnvoll, wie zum Beispiel der Einsatz des Quotamanagements im Rahmen des Globus Grid Computing Toolkits oder das dynamische Laden von Code als Dienst von OSGi.

EDAS und die entwickelten dezentralen adaptiven Dienstmodelle bieten bisher nur rudimentäre Sicherheitsmechanismen an. Einzige Ausnahme bildet die Unterstützung zum sicheren Binden von replizierten dezentralen Diensten. Aus Sicht des Autors können jedoch viele der Sicherheitsproblematiken durch existierende Techniken wie zum Beispiel Verschlüsselung, die Nutzung von Zertifikaten oder das Signieren von Code gelöst werden. Dennoch müssen entsprechende Bedrohungsszenarien erarbeitet werden und eine Validierung erfolgen, um dies zu verifizieren.

# Bibliography


[ADS06] Artur Andrzejak, Patricio Domingues, and Luis Silva. Predicting machine availabilities in desktop pools predicting machine availabilities in desktop pools. In *IEEE/IFIP Network Operations and Management Symposium (NOMS 2006)*, 2006. 152

[ADZ00] Mohit Aron, Peter Druschel, and Willy Zwaenepoel. Cluster reserves: a mechanism for resource management in cluster-based network servers. In *SIGMETRICS '00: Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 90–101, New York, NY, USA, 2000. ACM Press. 146

[AGKT02] Artur Andrzejak, Sven Graupner, Vadim Kotov, and Holger Trinks. Algorithms for self-organization and adaptive service placement in dynamic distributed systems. HPL Technical Report HPL-2002-259, Hewlett-Packard Laboratories, Palo Alto, USA, September 2002. 148

[All05] The OSGi Alliance. OSGi service platform: Core specification, release 4. Technical report, The OSGi Alliance, 2005. 87, 111, 123

[Ama] Amazon.com. Amazon EC2 - Getting started guide (http://docs.amazonwebservices.com/amazonec2/gsg/2006-06-26). 150

[aTMS03] Christian Kurmann amd Thomas M. Stricker. Zero-copy for CORBA - efficient communication for distributed object middleware. In *12th IEEE International Symposium on High Performance Distributed Computing*, pages 4–13. IEEE Computer Society, 2003. 81

[AV01] Marco Avvenuti and Alessio Vecchio. Embedding remote object mobility in Java RMI. In *FTDCS '01: Proceedings of the 8th IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 98–103, Washington, DC, USA, 2001. IEEE Computer Society. 98

[Bai97] Edward C. Bailey. *Maximum RPM: Taking the Red Hat package manager to the limit*. Red Hat Software, 1997. 110

[Ban98a] Bela Ban. Design and implementation of a reliable group communication toolkit for Java. Technical report, Department of Computer Science, Cornell University, 1998. 24, 58, 74

[Ban98b] Bela Ban. A flexible API for state transfer in the JavaGroups toolkit. Unpublished manuscript, 1998. 44

[Bau] Petr Baudis. Git – fast version control system (http://git.or.cz). 78

[BBC+04] Andy Bavier, Mic Bowman, Brent Chun, David Culler, Scott Karlin, Steve Muir, Larry Peterson, Timothy Roscoe, Tammo Spalink, and Mike Wawrzoniak. Operating system support for planetary-scale network services. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI'04)*, pages 253–266, San Francisco CA, March 2004. 101, 147

[BBH+98] Henri E. Bal, Raoul Bhoedjang, Rutger Hofman, Ceriel Jacobs, Koen Langendoen, Tim Rühl, and M. Frans Kaashoek. Performance evaluation of the Orca shared-object system. *ACM Transactions on Computer Systems*, 16(1):1–40, 1998. 99

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177, New York, NY, USA, 2003. ACM Press. 2, 101, 170

[BF03] Moshe Bar and Karl Fogel. *Open Source Development with CVS*. Paraglyph, 3 edition, 2003. 77

[BHV01] Walter Binder, Jane G. Hulaas, and Alex Alex Villazón. Portable resource control in Java. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 139–155, New York, NY, USA, 2001. ACM Press. 127

[Bir97] Kenneth P. Birman. *Building secure and reliable network applications*. Manning Publications Co., Greenwich, CT, USA, May 1997. 39, 44

[Bit] BitKeeper. http://www.bitkeeper.com. 78

[BKI03] Claudio Basile, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. A preemptive deterministic scheduling algorithm for multithreaded replicas. In *Proceedings International Conference on Dependable Systems and Networks (DSN 2003)*, pages 149–158, 2003. 67

[BMPV02] Roberto Baldoni, Carlo Marchetti, Raffaella Panella, and Luigi Verde. Handling FT-CORBA compliant interoperable object group references. In *WORDS '02: Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, pages 37–44, Washington, DC, USA, 2002. IEEE Computer Society. 16, 29

[BMV03] Roberto Baldoni, Carlo Marchetti, and Luigi Verde. CORBA request portable interceptors: analysis and applications. *Concurrency and Computation: Practice and Experience*, 15(6):551–579, 2003. 4, 11, 173

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984. 97

[Boi90] Jacques E. Boillat. Load balancing and poisson equation in a graph. *Concurrency: Practice and Experience*, 2(4):289–313, December 1990. 129

[BR93] Kenneth P. Birman and Robbert Van Renesse. *Reliable distributed computing with the ISIS Toolkit*. IEEE Computer Society, Los Alamitos, CA, USA, 1993. 22

[Bro97] Gerald Brose. JacORB: Implementation and design of a Java-ORB. In *Distributed Applications and Interoperable Systems, Proceedings IFIP TC 6.1 Working Conference DAIS'97*, pages 143–154. Chapman and Hall, 1997. 10, 74

[BRST02] J. Baumann, Hohl K. Rothermel, M. Strasser, and W. Theilmann. Mole: a mobile agent system. *Software: Practice and Experience*, 32(6):575–603, 2002. 43, 100

[BT88] Henri E. Bal and Andrew S. Tannenbaum. Distributed programming with shared data. In *Proceedings of ICCL — International Conference on Computer Languages*, pages 82–91, Miami, Florida, 1988. IEEE Computer Society Press. 99

[BV03] Bernd Brügge and Christoph Vilsmeier. Reducing CORBA call latency by caching and prefetching. In *IEEE Distributed Systems Online*, volume 04/06, June 2003. 72

[BWKI02] Claudio Basile, Keith Whisnant, Zbigniew Kalbarczyk, and Ravi Iyer. Loose synchronization of multithreaded replicas. In *SRDS '02: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, pages 250–255, Washington, DC, USA, 2002. IEEE Computer Society. 67

[CAC+02] Walter Cazzola, Massimo Ancona, Fabio Canepa, Massimo Mancini, and Vanja Siccardi. Enhancing Java to support object groups. In *Proceedings of the 3rd Conference on Recent Object-Oriented Trends (ROOTS'02)*, Bergen Norway, April 2002. 23

[Cas01] Miguel Castro. *Practical Byzantine Fault tolerance*. Ph.D., MIT, January 2001. Also as Technical Report MIT-LCS-TR-817. 45

[CAT+01] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centers. In *SOSP '01: Proceedings of the eighteenth ACM Symposium on Operating Systems Principles*, pages 103–116, New York, NY, USA, 2001. ACM Press. 146

[CBCP02] Geoff Coulson, Gordon S. Blair, Michael Clarke, and Nikos Parlavantzas. The design of a configurable and reconfigurable middleware platform. *Distributed Computing*, 15(2):109–126, 2002. 12

[CDT03] Grzegorz Czajkowski, Laurent Daynès Laurent Daynès, and Ben Titzer. A multi-user virtual machine. In *ATEC'03: Proceedings of the USENIX Annual Technical Conference 2003 on USENIX Annual Technical Conference*, pages 85–98, Berkeley, CA, USA, 2003. USENIX Association. 127

[CFH+05] Christopher Clark, Keir Fraser, Steven Hand, Jacob G. Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 273–286, Boston, MA, May 2005. 43

[CGSM03] Luc Courtrai, Frédéric Guidec, Nicolas Le Sommer, and Yves Mahéo. Resource management for parallel adaptive components. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, pages 134–140, Washington, DC, USA, 2003. IEEE Computer Society. 126

[CHS+04] Grzegorz Czajkowski, Stephen Hahn, Glenn Skinner, Pete Soper, and Ciarán Bryce. A resource management interface for the javatm platform. *Software: Practice and Experience*, 32(2):123–157, 2004. 128

[CM00] Luis M. Peña Cabañas and Juan Pavón Mestras. Conditions for the state transfer on virtual synchronous systems. In *Proceedings of The 10th International Conference on Computing and Information ICCI' 2000*, 2000. 5, 44, 174

[CSWH00] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, 2000. 32

[CvE98] Grzegorz Czajkowski and Thorsten von Eicken. JRes: a resource accounting interface for Java. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 21–35, New York, NY, USA, 1998. ACM Press. 127

[Cyb89] Gerorg Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7(2):279–301, Oct 1989. 129

[Cza00] Grzegorz Czajkowski. Application isolation in the java virtual machine. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 354–366, New York, NY, USA, 2000. ACM Press. 128

[Dar] Darcs. http://abridgegame.org/darcs. 77

[DBK+01] Frank Dabek, Emma Brunskill, M. Frans Kaashoek, David Karger, Robert Morris, Ion Stoica, and Hari Balakrishnan. Building peer-to-peer systems with Chord, a distributed lookup service. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 81–86. IEEE Computer Society, May 2001. 85, 87, 147

[DEJ01] 3rd D. Eastlake and P. Jones. US Secure Hash Algorithm 1 (SHA1). Technical Report RFC3174, RFC Editor, 2001. 34

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976. 32

[DH04] Scott Douglas and Aaron Harwood. Diffusive load balancing of loosely-synchronous parallel programs over peer-to-peer networks. *The Computing Research Repository (CoRR)*, cs.DC/0410009, 2004. 129

[DHRK06] Jörg Domaschka, Franz J. Hauck, Hans P. Reiser, and Rüdiger Kapitza. Deterministic multithreading for java-based replicated objects. In *Proceedings of the 18th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'06, Dallas, Texas)*, pages 516–521, 2006. 66, 75

[Dis07] Tobias Distler. Konzeption und implementierung eines dezentralen ressourcen-management systems. Master's thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2007. 138, 141

[DMS92] Peter Dickman, Mesaac Makpangou, and Marc Shapiro. Contrasting fragmented objects with uniform transparent object references for distributed programming. In *EW 5: Proceedings of the 5th Workshop on ACM SIGOPS European Workshop*, pages 1–5, New York, NY, USA, 1992. ACM Press. 9

[Dou02] John R. Douceur. The sybil attack. In *Proceedings of the IPTPS02 Workshop*, volume 2429 of *Lecture Notes in Computer Science*, pages 251–260. Springer, 2002. 1, 169

[Fel98] Pascal Felber. *The CORBA object group service: A service approach to object groups in CORBA*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998. Number 1867. 4, 22, 173

[FGS98] Pascal Felber, Rachid Guerraoui, and Andre Schiper. The implementation of a CORBA object group service. *Theory and Practice of Object Systems*, 4(2):93–105, 1998. 66

[FH02] Ray Friedman and Erez Hadad. FTS: A high-performance CORBA fault-tolerance service. In *Proceedings of the Seventh International Workshop on Object-Oriented Real-Time Dependable Systems*, 2002. 4, 22, 173

[FHMO04] Alois Ferscha, Manfred Hechinger, Rene Mayrhofer, and Roy Oberhauser. A light-weight component model for peer-to-peer applications. In *ICDCSW '04: Proceedings of the 24th International Conference on Distributed Computing Systems Workshops - W7: EC (ICDCSW'04)*, pages 520–527, Los Alamitos, CA, USA, 2004. IEEE Computer Society. 87

[FJ02] Thomas Fahringer and Alexandru Jugravu. JavaSymphony: New directives to control and synchronize locality, parallelism, and load balancing for cluster and GRID-computing. In *JGI '02: Proceedings of the 2002 joint ACM-ISCOPE Conference on Java Grande*, pages 8–17, New York, NY, USA, 2002. ACM Press. 99

[FJRS01] Pascal Felber, Ben Jai, Rajeev Rastogi, and Mark Smith. Using semantic knowledge of distributed objects to increase reliability and availability. In *WORDS*

*'01: Proceedings of the Sixth International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'01)*, page 153, Washington, DC, USA, 2001. IEEE Computer Society. 72

[FKT01] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International Journal on Supercomputer Applications*, 15(3), 2001. 1, 101

[FMLF97] J. Fraga, C. Maziero, L. C. Lung, and O. G. Loques Filho. Implementing replicated services in open systems using a reflective approach. In *ISADS '97: Proceedings of the 3rd International Symposium on Autonomous Decentralized Systems*, page 273, Washington, DC, USA, 1997. IEEE Computer Society. 66

[FN04] Pascal Felber and Priya Narasimhan. Experiences, strategies, and challenges in building fault-tolerant CORBA systems. *IEEE Transactions on Computers*, 53(5):497–511, 2004. 22

[Fos05] Ian Foster. *Globus toolkit version 4: Software for service-oriented systems*, volume 3779, pages 2–13. Springer, 2005. 2, 101, 170

[Fou04] Foundation for Intelligent Physical Agents (FIPA). FIPA agent management specification. Technical Report SC00023K, FIPA, 2004. 41

[GAKT04] Sven Graupner, Artur Andrzejak, Vadim E. Kotov, and Holger Trinks. Adaptive service placement algorithms for autonomous service networks. In *Engineering Self-Organising Systems*, volume 3464 of *Lecture Notes in Computer Science*, pages 280–297. Springer, 2004. 148, 150

[GAZ03] Christos Gkantsidis, Mostafa Ammar, and Ellen Zegura. On the effect of large-scale deployment of parallel downloading. In *WIAPP '03: Proceedings of the The Third IEEE Workshop on Internet Applications*, pages 79– 89, Washington, DC, USA, 2003. IEEE Computer Society. 51, 52, 54

[GC89] Cary G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP '89: Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 202–210, New York, NY, USA, 1989. ACM Press. 7, 40, 128

[GCBP05] Paul Grace, Geoff Coulson, Gordon Blair, and Barry Porter. Deep middleware for the divergent grid. In *Proceedings of the 6th IFIP/ACM/USENIX International Middleware Conference 2005, Grenoble, France*, volume 3790 of *Lecture Notes in Computer Science*, pages 334–353. Springer, 2005. 87

[Gei01] Martin Geier. *Fragmentierte Objekte für die Implementierung mobiler Agenten*. PhD thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg, April 2001. 43

[GGKS02] Karl Gottschalk, Stephen Graham, Heather Kreger, and James Snell. Introduction to Web services architecture. *IBM Systems Journal*, 41(2):170–177, 2002. 98

[GLS02] Frédéric Guidec and Nicolas Le Sommer. Towards resource consumption accounting and control in Java: A practical experience. Workshop on Resource Management for Safe Language, 2002. 127

[GNSY00] Andy Gokhale, Bala Natarajan, Douglas C. Schmidt, and Shalini Yajnik. DOORS: Towards high-performance fault-tolerant CORBA. In *Proceedings of the 2nd International Symposium on Distributed Objects and Applications (DOA '00)*, 2000. 4, 22, 173

[GNU] GNU arch. http://www.gnu.org/software/gnu-arch. 77

[Gon01] Li Gong. JXTA: A network programming environment. *IEEE Internet Computing*, 5(3):88–95, 2001. 6, 86

[Gro88] Network Working Group. RPC: Remote Procedure Call protocol specification. Technical report, Sun Microsystems, United States, 1988. 97

[HB04] Jarle Hulaas and Walter Binder. Program transformations for portable CPU accounting and control in Java. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 169–177, New York, NY, USA, 2004. ACM Press. 127

[Hen98] Michi Henning. Binding, migration, and scalability in CORBA. *Communications of the ACM special issue on CORBA*, 41(10):67–71, Oct 1998. 28, 98

[Hen04] Michi Henning. A new approach to object-oriented middleware. *IEEE Internet Computing*, 8(1):66–75, 2004. 3, 98, 172

[HHaIP03] Steven Hand, Tim Harris, and Evangelos Kotsovinos amd Ian Pratt. Controlling the XenoServer open platform. In *Proceedings of the Sixth IEEE Conference on Open Architectures and Network Programming (OPENARCH 2003)*, pages 3–11. IEEE Computer Society, April 2003. 2, 101, 142, 148, 170

[HKRS05] Franz J. Hauck, Rüdiger Kapitza, Hans P. Reiser, and Andreas I. Schmied. A flexible and extensible object middleware: CORBA and beyond. In *SEM '05: Proceedings of the 5th International Workshop on Software Engineering and Middleware*, pages 69–75, New York, NY, USA, 2005. ACM Press. 4, 10, 74, 172

[HL06] Beda C. Hammerschmidt and Volker Linnemann. Migratable Web services: Increasing performance and privacy in service oriented architectures. *IADIS International Journal on Computer Science and Information Systems*, 1(1):42–56, 2006. 98

[HM76] J. W. Hunt and M. Douglas McIlroy. An algorithm for differential file comparison. Technical Report CSTR 41, Bell Telephone Laboratories, 1976. 55

[HSS] Graydon Hoare, Nathaniel Smith, and Derek Scherger. *Monotone - A distributed version control system*. 77

[HvDvS+95] Philip Homburg, Leendert van Doorn, Maarten van Steen, Andrew S. Tanenbaum, and Wiebren de Jonge. An object model for flexible distributed systems. In *Proceedings of the 1st Annual ASCI Conference*, pages 69–78, Heijen, Netherlands, May 1995. 99

[IF01] Adriana Iamnitchi and Ian T. Foster. On fully decentralized resource discovery in grid environments. In *GRID '01: Proceedings of the Second International Workshop on Grid Computing*, pages 51–62, London, UK, 2001. Springer. 148

[JK98] H.-Arno Jacobsen and Bernd J. Krämer. A design pattern based approach to generating synchronization adaptors from annotated IDL. In *ASE '98: Proceedings of the 13th IEEE international Conference on Automated Software Engineering*, pages 63–75, Washington, DC, USA, 1998. IEEE Computer Society. 69, 72

[JK00] H.Arno Jacobsen and Bernd J. Krämer. Modeling interface definition language extensions. In *Proceedings of the 37th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS)*, pages 242–252, 2000. 69, 72

[JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, 1988. 40, 99

[JPPMA02] Ricardo Jiménez-Peris, Marta Patiño-Martínez, and Gustavo Alonso. Non-intrusive, parallel recovery of replicated data. In *SRDS '02: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, pages 150–159, Washington, DC, USA, 2002. IEEE Computer Society. 46

[JRR97] Michael B. Jones, Daniela Roşu, and Marcel-Cătălin Roşu. Cpu reservations and time constraints: efficient, predictable scheduling of independent activities. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 198–211, New York, NY, USA, 1997. ACM Press. 146

[KBB01] Bettina Kemme, Alberto Bartoli, and Özalp Babaoglu. Online reconfiguration in replicated databases based on group communication. In *DSN '01: Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*, pages 117–126, Los Alamitos, CA, USA, 2001. IEEE Computer Society. 46

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003. 1

[KDH+06] Rüdiger Kapitza, Jörg Domaschka, Franz J. Hauck, Hans P. Reiser, and Holger Schmidt. FORMI: Integrating adaptive fragmented objects into Java RMI. *IEEE Distributed Systems Online*, 7(10), 2006. 4, 172

[KH03] Rüdiger Kapitza and Franz J. Hauck. DLS: A CORBA service for dynamic loading of code. In *Proceedings of the OTM Confederated International Conferences*

*(DOA, Sicily, Italy)*, volume 2888 of *Lecture Notes in Computer Science*. Springer, 2003. 6, 16, 17, 109, 111, 113, 118, 176

[KHK96] Thomas Koch, Gregor Henze, and Bernd Kraemer. Annotations for synchronization constraints in CORBA IDL. In *SDNE '96: Proceedings of the 3rd Workshop on Services in Distributed and Networked Environments (SDNE '96)*, page 122, Washington, DC, USA, 1996. IEEE Computer Society. 69

[KK00] Rainer Koster and Thorsten Kramp. Structuring QoS-supporting services with smart proxies. In *Proceedings of the IFIP/ACM Middleware Conference (Middleware)*, volume 1795 of *Lecture Notes in Computer Science*, pages 273–288, Berlin, Heidelberg, New York, Tokyo, 2000. Springer. 4, 9, 24, 173

[KKSH05] Rüdiger Kapitza, Michael Kirstein, Holger Schmidt, and Franz J. Hauck. FORMI: An RMI extension for adaptive applications. In *ARM '05: Proceedings of the 4th Workshop on Reflective and Adaptive Middleware Systems*, New York, NY, USA, 2005. ACM Press. 4, 172

[KLLS05] Kfir Karmon, L. Liran Liss, and Assaf Schuster. GWiQ-P: An efficient decentralized grid-wide quota enforcement protocol. In *Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing*, pages 222–232. IEEE Computer Society, 2005. 126, 140

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997. 70

[KM] Tor Klingberg and Raphaël Manfredi. Gnutella 0.6 (http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html). 85, 133

[KMP+04] Evangelos Kotsovinos, Tim Moreton, Ian Pratt, Russ Ross, Keir Fraser, Steven Hand, and Tim Harris. Global-scale service deployment in the XenoServer platform. In *Proceedings of the First Workshop on Real, Large Distributed Systems (WORLDS '04)*, San Francisco, CA, December 2004. 101

[KOS+00] Fred Kuhns, Carlos O'Ryan, Douglas C. Schmidt, Ossama Othman, and Jeff Parsons. The design and performance of a pluggable protocols framework for CORBA middleware. In *PfHSN '99: Proceedings of the IFIP TC6 WG6.1 & WG6.4 / IEEE ComSoc TC on on Gigabit Networking Sixth International Workshop on Protocols for High Speed Networks VI*, pages 81–98, Deventer, Netherlands, 2000. Kluwer, B.V. 81

[Kot05] Evangelos Kotsovinos. Global public computing. Technical Report UCAM-CL-TR-615, Computer Laboratory, University of Cambridge, January 2005. 101

[KPP04] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack Problems*. Springer, 2004. 145

[KSH05] Rüdiger Kapitza, Holger Schmidt, and Franz J. Hauck. Platform-independent object migration in CORBA. In *Proceedings of the OTM'05 Conferences (DOA, Agia Napa, Cyprus, Oct 31-Nov. 04, 2005)*, volume 3760 of *LNCS*, pages 900–917. Springer, Oct 2005. 49, 94, 98

[KSSH06] Rüdiger Kapitza, Holger Schmidt, Guido-Arndt Söldner, and Franz J. Hauck. A framework for adaptive mobile objects in heterogeneous environments. In *On the Move to Meaningful Internet Systems 2006 - CoopIS, DOA, and ODBASE*, volume 4276 of *Lecture Notes in Computer Science*, pages 1739–1756. Springer, 2006. 49, 94

[LO98] Danny B. Lange and Mitsuru Oshima. *Programming and Deploying Java Mobile Agents Aglets*. Addison-Wesley, 1998. 43, 100

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982. 25

[Maf95] Silvano Maffeis. Adding group communication and fault-tolerance to CORBA. In *Proceedings of the Conference on Object-Oriented Technologies, (Monterey, CA), USENIX*, pages 135–146, 1995. 4, 22, 44, 173

[Mal96] Christoph Peter Malloth. *Conception and implementation of a toolkit for building fault-tolerant distributed applications in large scale networks*. PhD thesis, EPFL, 1996. 44

[MDP+00] Dejan S. Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Computer Survey*, 32(3):241–299, 2000. 43

[MGNS94] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro. Fragmented objects for distributed abstractions. In T. L. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*, pages 170–186. IEEE Computer Society, Juli 1994. 4, 9, 99, 172

[MHS98] Joshua P. MacDonald, Paul N. Hilfinger, and Luigi Semenzato. PRCS: The project revision control system. In *Proceedings System Configuration Management-SCM-8*, volume 1439 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 1998. 55

[Mic01] Sun Microsystems. Jini technology core platform specification. Technical report, Sun Microsystems, 2001. 10, 40

[Mic02] Sun Microsystems. Java™ management extensions instrumentation and agent specification. Technical Report 1.2, Sun Microsystems, Oct 2002. 127

[Mic05] Sun Microsystems. JXTA v2.3.x: Java programmer's guide. Technical report, Sun Microsystems, Sun Antonio Road, Paolo Alto, CA 94303-4900 USA, 2005. 87

[Mil92] David L. Mills. Network Time Protocol (version 3) specification, implementation and analysis. Technical Report RFC-1305, Network Working Group, 1992. 31

[MLC98] Dejan S. Milojičić, William LaForge, and Deepika Chauhan. Mobile objects and agents (MOA). In *4th USENIX Conference on Object-Oriented Technologies and Systems*, pages 179–194, Santa Fe, New Mexico, 1998. USENIX Association. 43, 100

[MMRM05] Sam Malek, Marija Mikic-Rakic, and Nenad Medvidovic. A decentralized redeployment algorithm for improving the availability of distributed systems. In Alan Dearle and Susan Eisenbach, editors, *Component Deployment*, volume 3798 of *Lecture Notes in Computer Science*, pages 99–114. Springer, 2005. 149, 150

[MMSN98] Luise E. Moser, P. M. Melliar-Smith, and Priya Narasimhan. Consistent object replication in the Eternal system. *Theory and Practice of Object Systems*, 4(2):81–92, 1998. 4, 22, 173

[MMVB00] Carlo Marchetti, Massimo Mecella, Antonino Virgillito, and Roberto Baldoni. An interoperable replication logic for CORBA systems. In *DOA '00: Proceedings of the International Symposium on Distributed Objects and Applications*, pages 7–16, Washington, DC, USA, 2000. IEEE Computer Society. 4, 22, 173

[Mon99] Alberto Montresor. The Jgroup reliable distributed object model. In *Proceedings of the 2nd IFIP International Working Conference on Distributed Applications and Systems (DAIS '99)*, page 389, Deventer, Netherlands, June 1999. Kluwer, B.V. 23

[MPHD06] Alan Mislove, Ansley Post, Andreas Haeberlen, and Peter Druschel. Experiences in building and operating ePOST, a reliable peer-to-peer application. In *EuroSys '06: Proceedings of the 2006 EuroSys Conference*, pages 147 – 159, New York, NY, USA, 2006. ACM Press. 1, 169

[MPS93] Shivakant Mishra, Larry L. Peterson, and Richard D. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. *Distributed Systems Engineering*, 1:87–103, 1993. 45

[Mul93] Sape Mullender, editor. *Distributed systems (2nd Ed.)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993. 4, 21, 173

[MWN02] Scott McLean, Kim Williams, and James Naftel. *Microsoft .Net Remoting*. Microsoft Press, Redmond, WA, USA, 2002. 3, 172

[NMMS99] Priya Narasimhan, Louise E. Moser, and P. M. Melliar-Smith. Enforcing determinism for the consistent replication of multithreaded CORBA applications. In *SRDS '99: Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, pages 263–273, Washington, DC, USA, 1999. IEEE Computer Society. 66

[NMMS00] Nitya Narasimhan, Louise E. Moser, and P. M. Melliar-Smith. Transparent consistent replication of Java RMI objects. In *DOA '00: Proceedings of the International Symposium on Distributed Objects and Applications*, pages 17–26, Washington, DC, USA, 2000. IEEE Computer Society. 23

[NMMS01] Priya Narasimhan, Louise E. Moser, and P. M. Melliar-Smith. State synchronization and recovery for strongly consistent replicated CORBA objects. In *DSN*, pages 261–270. IEEE Computer Society, 2001. 44

[NS06] Yoav Nebat and Moshe Sidi. Parallel downloads for streaming applications: A resequencing analysis. *Performance Evaluation*, 63(1):15–35, 2006. 52

[NZ02] T. S. Eugene Ng and Hui Zhang. Predicting internet network distance with coordinates-based approaches. *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, 1:170–179 vol.1, 2002. 154

[OAPA05] David Oppenheimer, Jeannie Albrecht, David A. Patterson, and Vahdat Amin. Design and implementation tradeoffs for wide-area resource discovery. In *Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing*, pages 113–124. IEEE Computer Society, 2005. 7, 143, 144, 147, 153, 176

[Obj01] Object Management Group (OMG). *Portable interceptors (final adopted specification) OMG technical committee document*. Object Management Group (OMG), formal/01-12-25 edition, December 2001. 4, 11, 22, 173

[Obj02] Object Management Group (OMG). *Life cycle service specification*. Object Management Group (OMG), 1.2 edition, September 2002. 28, 73, 97

[Obj04] Object Management Group (OMG). *Common Object Request Broker Architecture: Core Specification*. Object Management Group (OMG), 3.0.3 edition, March 2004. 3, 11, 21, 172

[OCP+06] David Oppenheimer, Brent Chun, David Patterson, Alex C. Snoeren, and Amin Vahdat. Service placement in shared wide-area platforms. In *Proceedings of the USENIX Annual Technical Conference*, pages 273–288, Boston,MA, June 2006. 148, 149, 152, 153

[PACR03] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A blueprint for introducing disruptive technology into the Internet. *SIGCOMM Computer Communication Review*, 33(1):59–64, 2003. 2, 101, 113, 142, 143, 147, 170

[PC03] Daryl Parker and David Cleary. A P2P approach to classloading in Java. In Gianluca Moro, Claudio Sartori, and Munindar P. Singh, editors, *Second Internatinal Workshop on Agents and Peer-to-Peer Computing (AP2PC 2003)*, volume 2872 of *Lecture Notes in Computer Science*, pages 144–149. Springer, 2003. 6, 111, 176

[PCSF04] C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. *Version control with Subversion*. O'Reilly Media, 1 edition, June 2004. 77

[PG00] Yvan Peter and Hervé Guyennet. Object mobility in large scale systems. *Cluster Computing*, 3(2):177–185, 2000. 98

[PHS96] Irfan Pyarali, Timothy H. Harrison, and Douglas C. Schmidt. Design and performance of an object-oriented framework for high-speed electronic medical imaging. *Computing Systems*, 9(4):331–375, 1996. 81

[PKF05] Stefan Paal, Reiner Kammüller, and Bernd Freisleben. Dynamic software deployment with distributed application repositories. In *14. ITG/GI-Fachtagug Kommunikation in Verteilten Systemen (KiVS 2005)*, Informatik Aktuell, pages 41–52. Springer, 2005. 6, 111, 176

[PKF06] Stefan Paal, Reiner Kammüller, and Bernd Freisleben. Self-managing application composition for cross-platform operating environments. In *International Conference on Autonomic and Autonomous Systems (ICAS 2006)*, pages 37–51. IEEE Computer Society, 2006. 111

[PLG+06] Kristal T. Pollack, Darrell D. E. Long, Richard A. Golding, Benjamin Reed, and Ralph A. Becker-Szendy. Quota enforcement for high-performance distributed systems. under submission, 2006. 140

[PR06] Larry Peterson and Timothy Roscoe. The design principles of PlanetLab. *Operating Systems Review*, 40(1):11–16, January 2006. 101

[PSWL95] Graham D. Parrington, Santosh K. Shrivastava, Stuart M. Wheater, and Mark C. Little. The design and implementation of Arjuna. *Computing Systems*, 8(2):255–308, 1995. 44

[PWP+04] Vivek S. Pai, Limin Wang, KyoungSoo Park, Ruoming Pang, and Larry Peterson. The dark side of the Web: an open proxy's view. *SIGCOMM Computer Communication Review*, 34(1):57–62, 2004. 147

[RB02] Pablo Rodriguez and Ernst W. Biersack. Dynamic parallel access to replicated content in the Internet. *IEEE/Transactions on Networking*, 10(4):455–465, 2002. 50, 51, 52, 54

[RBC+03] Yansong Jennifer Ren, David E. Bakken, Tod Courtney, Michel Cukier, David A. Karr, Paul Rubel, Chetan Sabnis, William H. Sanders, Richard E. Schantz, and Mouna Seri. AQuA: An adaptive architecture that provides dependable distributed objects. *IEEE Transactions on Computers*, 52(1):31–50, 2003. 4, 173

[RCAA05] Jerry Rolia, Ludmila Cherkasova, Martin Arlitt, and Artur Andrzejak. A capacity management service for resource pools. In *WOSP '05: Proceedings of the 5th International Workshop on Software and Performance*, pages 229–237, New York, NY, USA, 2005. ACM Press. 146

[RCAM06] Jerry Rolia, Ludmila Cherkasova, Martin Arlitt, and Vijay Machiraju. Supporting application quality of service in shared resource pools. *Communications of the ACM*, 49(3):55–60, 2006. 142, 143, 144

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001. 85, 147

[RDH05] Hans P. Reiser, Michael J. Danel, and Franz J. Hauck. A flexible replication framework for scalable and reliable .NET services. In *Proceedings of the IADIS International Conference on Applied Comuting 2005, Vol I, Algarve, P*, pages 161–169. IADIS, 2005. 23

[RDK05] Thomas Repantis, Yannis Drougas, and Vana Kalogeraki. Adaptive resource management in peer-to-peer middleware. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 2*, volume 03, page 132b, Washington, DC, USA, 2005. IEEE Computer Society. 149

[Rei06] Hans P. Reiser. *Flexible and reconfigurable support for fault-tolerant object replication*. PhD thesis, Fakultät für Ingenieurwissenschaften und Informatik, Universität Ulm, 2006. 25, 67, 70

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 161–172, New York, NY, USA, 2001. ACM Press. 85

[RHD+06] Hans P. Reiser, Franz J. Hauck, Jörg Domaschka, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat. Consistent replication of multithreaded distributed objects. In *SRDS '06: Proceedings of the 25st IEEE Symposium on Reliable Distributed Systems (SRDS'06)*, pages 257 – 266. IEEE Computer Society, 2006. 67

[RHKS03] Hans P. Reiser, Franz J. Hauck, Rüdiger Kapitza, and Andreas I. Schmied. Integrating fragmented objects into a CORBA environment. In *Proceedings of the Net.ObjectDays (Erfurt, Germany)*, pages 264–272. tranSIT GmbH, 2003. 4, 172

[Ric02] Jeffery Richter. *Applied Microsoft .NET Framework Programming*. Microsoft Press, 2002. 40

[RKB00] Pablo Rodriguez, Andreas Kirpal, and Ernst W. Biersack. Parallel-access for mirror sites in the Internet. In *Proceedings of the nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2000)*, volume 2, pages 864–873. IEEE Computer Society, 2000. 50, 51, 54

[ROLV06] Robert Ricci, David Oppenheimer, Jay Lepreau, and Amin Vahdat. Lessons from resource allocators for large-scale multiuser testbeds. *ACM SIGOPS Operating Systems Review*, 40(1):25–32, 2006. 142

[Rou96] Ellard T. Roush. Fast dynamic process migration. In *ICDCS '96: Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96)*, pages 637–645, Washington, DC, USA, 1996. IEEE Computer Society. 43

[RSH01] Hans P. Reiser, Martin Steckermeier, and Franz J. Hauck. IDLflex: a flexible and generic compiler for CORBA IDL. In *Proceedings of the Net.ObjectDays (Erfurt, Germany)*, pages 151–160. tranSIT GmbH, 2001. 23, 68

[SBB+01] Niranjan Suri, Jeffrey M. Bradshaw, Maggie R. Breedy, Paul T. Groth, Gregory A. Hill, and Raul Saavedra. State capture and resource control for Java: The design and implementation of the Aroma virtual machine. In *Java Virtual Machine Research and Technology Symposium*. USENIX Association, 2001. 127

[Sch95] Bruce Schneier. *Applied cryptography: Protocols, algorithms, and source code in C*. Wiley, 2 edition, October 1995. 32

[SCHH05] David Spence, Jon Crowcroft, Steven Hand, and Tim Harris. Location based placement of whole distributed systems. In *CoNEXT'05: Proceedings of the 2005 ACM Conference on Emerging Network Experiment and Zechnology*, pages 124–134, New York, NY, USA, 2005. ACM Press. 144, 148

[SCRD04] Atul Singh, Miguel Castro, Antony Rowstron, and Peter Druschel. Defending against eclipse attacks on overlay networks. In *Proceedings of the 11th ACM SIGOPS European Workshop*, New York, NY, USA, September 2004. ACM Press. 1, 169

[SDP92] Marc Shapiro, Peter Dickman, and David Plainfoss. Robust, distributed references and acyclic garbage collection. In *Proceedings of the eleventh Annual ACM Symposium on Principles of Distributed Computing*, pages 135–146. ACM Press, 1992. 40

[SGH+89] Marc Shapiro, Yvon Gourbant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Celine Valot. SOS: An object-oriented operating system - assessment and perspectives. *Computing Systems*, 2(4):287–337, 1989. 99

[SGS+05] Sharad Singhal, Sven Graupner, Akhil Sahai, Vijay Machiraju, Jim Pruyne, Xiaoyun Zhu, Jerry Rolia, Martin Arlitt, Cipriano Santos, Dirk Beyer, and Julie Ward. Quartermaster - a resource utility system. In *9th IFIP/IEEE International Symposium on Integrated Network Management*, pages 265–278, 2005. 143, 145, 146

[SH99] Michael Schröder and Franz J. Hauck. Juggle: Eine verteilte virtuelle maschine für Java. In *Java Informationstage (JIT'99)*, Informatik Aktuell, pages 278–289. Springer, 1999. 43, 99

[SH03] David Spence and Tim Harris. XenoSearch: Distributed resource discovery in the XenoServer open platform. In *HPDC '03: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, page 216, Washington, DC, USA, 2003. IEEE Computer Society. 7, 148, 176

[Sha86] Marc Shapiro. Structure and encapsulation in distributed systems: The proxy principle. In *Proceedings of the 6th International Conference on Distributed Systems (ICDCS)*, pages 198–204, Cambridge, MA, USA, 1986. IEEE Computer Society. 4, 9, 172

[SM05] Inc. Sun Microsystems. Java Web Start overview. White paper, Sun Microsystems Inc., 2005. 6, 111, 176

[SMS02] Nuno Santos, Paulo Marques, and Luis Silva. A framework for smart proxies and interceptors in RMI. In *ISCA 15th International Conference on Parallel and Distributed Computing Systems (PDCS 2002)*, September 2002. 4, 9, 24, 173

[Soc05] The Internet Society. JXTA v2.0 protocols specification. Technical Report http://spec.jxta.org/nonav/v1.0/docbook/JXTAProtocols.html, Sun Microsystems, Sun Antonio Road, Paolo Alto, CA 94303-4900 USA, 2001, last changed 2005. 86, 88

[SSMJ04] Akhil Sahai, Sharad Singhal, Vijay Machiraju, and Rajeev Joshi. Automated generation of resource configurations through policies. *POLICY*, 00:107–110, 2004. 146

[SWB+97] III Sullivan, Woodruff T., Dan Werthimer, Stuart Bowyer, Jeff Cobb, David Gedye, and David Anderson. A new major SETI project based on project serendip data and 100,000 personal computers. In *Proceedings of the Fifth International Conference on Bioastronomy*, number 161 in IAU Colloquium, pages 729–733, 1997. 1, 169

[Szy97] Clemens Szyperski. *Component software: Beyond object-oriented programming*. Addison-Wesley Professional, December 1997. 87

[TAA+03] Bernard Traversat, Ahkil Arora, Mohamed Abdelaziz, Mike Duigou, Carl Haywood, Jean-Christophe Hugly, Eric Pouyoul, and Bill Yeager. Project JXTA 2.0 super-peer virtual network. Technical Report Whitepaper, Sun Microsystems Inc., 2003. 88

[TM96] Andrew Tridgell and Paul Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, ANU Computer Science, 1996. 56, 78

[TTL02] Douglas Thain, Todd Tannenbaum, and Miron Livny. Condor and the Grid. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*, pages 299–236. John Wiley & Sons, Inc., December 2002. 2, 170

[TW96] David L. Tennenhouse and David J. Wetherall. Towards an active network architecture. *SIGCOMM Computer Communication Review*, 26(2):5–17, 1996. 100

[URS04] Bhuvan Urgaonkar, Arnold L. Rosenberg, and Prashant J. Shenoy. Application placement on a cluster of servers. In *ISCA 17th International Conference on Parallel and Distributed Computing Systems (PDCS 2004)*, pages 85–90, 2004. 145, 146

[US04] Bhuvan Urgaonkar and Prashant Shenoy. Sharc: Managing cpu and network bandwidth in shared clusters. *IEEE Trans. Parallel Distrib. Syst.*, 15(1):2–17, 2004. 146

[Vaz04] Sudharshan Vazhkudai. Distributed downloads of bulk, replicated grid data. *Journal of Grid Computing*, 2(1):31–42, 2004. 51

[VHPK04] Thomas Vergnaud, Jérôme Hugues, Laurent Pautet, and Fabrice Kordon. PolyORB: A schizophrenic middleware to build versatile reliable distributed applications. In *Proceedings of the 9th International Conference on Reliable Software Technologies Ada-Europe*, volume 3063 of *Lecture Notes in Computer Science*, pages 106–119. Springer, 2004. 12

[vNMW+05] Rob V. van Nieuwpoort, Jason Maassen, Gosia Wrzesinska, Rutger Hofman, Ceriel Jacobs, Thilo Kielmann, and Henri E. Bal. Ibis: a flexible and efficient Java based grid programming environment. *Concurrency and Computation: Practice and Experience*, 17(7-8):1079–1107, June 2005. 99

[vSHBT98] Maarten van Steen, Franz J. Hauck, Gerco Ballintijn, and Andrew S. Tanenbaum. Algorithmic design of the globe wide-area location service. *The Computer Journal*, 41(5):297–310, 1998. 99

[vSHT99] Maarten van Steen, Philip Homburg, and Andrew S. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, 7(1):70–78, 1999. 99

[Wal98] Jim Waldo. Remote procedure calls and java remote method invocation. *IEEE Concurrency*, 6(3):5–7, 1998. 3, 172

[WGT98] David J. Wetherall, John Guttag, and David L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *Open Architectures and Network Programming (IEEE OPENARCH'98)*, pages 117–129. IEEE Computer Society, 1998. 100

[WHN+01] Johnny Wong, Guy Helmer, Venkatraman Naganathan, Srinivas Polavarapu, Vasant Honavar, and Les Miller. SMART mobile agent facility. *Journal of Systems and Software*, 56:9–22, 2001. 100

[Wil02] Brendon J. Wilson. *JXTA*. New Riders, jun 2002. 90

[WJD03] Eric Wohlstadter, Stoney Jackson, and Premkumar Devanbu. DADO: Enhancing middleware to support crosscutting features in distributed, heterogeneous

systems. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 174–186, Washington, DC, USA, 2003. IEEE Computer Society. 70, 72

[WJD04] Eric Wohlstadter, Stoney Jackson, and Premkumar Devanbu. Design and implementation of distributed crosscutting features with DADO. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 706–707, Washington, DC, USA, 2004. IEEE Computer Society. 70, 72

[WLS+02] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI '02: Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 255–270, New York, NY, USA, 2002. ACM Press. 142, 144

[WM01] Barry Brian Werger and Maja J. Matarić. From insect to internet: Situated control for networked robot teams. *Annals of Mathematics and Artificial Intelligence*, 31(1-4):173–197, 2001. 149

[Wol98] Laurence A. Wolsey. *Integer Programming*. John Wiley & Sons, Inc., 1998. 145

[WPSO01] Nanbor Wang, Kirthika Parameswaran, Douglas Schmidt, and Ossama Othman. The design and performance of meta-programming mechanisms for object request broker middleware. In *Proceedings of the 6th USENIX Conference on Object-Oriented Techology and Systems (COOTS'01)*, 2001. 12

[YCM06] Alexander Yip, Benjie Chen, and Robert Morris. Pastwatch: a distributed version control system. In *Proceedings of the 3rd USENIX/ACM NSDI*, pages 381–394, 2006. 77

[ZMMS05] Wenbing Zhao, Louise E. Moser, and P. M. Melliar-Smith. Deterministic scheduling for multithreaded replicas. In *WORDS '05: Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 74–81, Washington, DC, USA, 2005. IEEE Computer Society. 66

[ZSW+03] Xiaoyun Zhu, Cipriano Santos, Julie Ward, Dirk Beyer, and Sharad Singhal. Resource assignment for large-scale computing utilities using mathematical programming. Technical Report HPL-2003-243R1, Hewlett Packard Laboratories, 2003. 145, 146