

Virtual Private Computing

Martin Steckermeier
Franz Hauck

May 1999

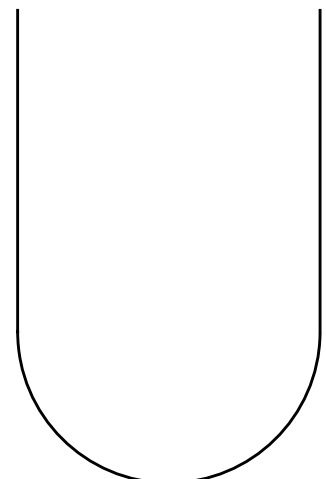
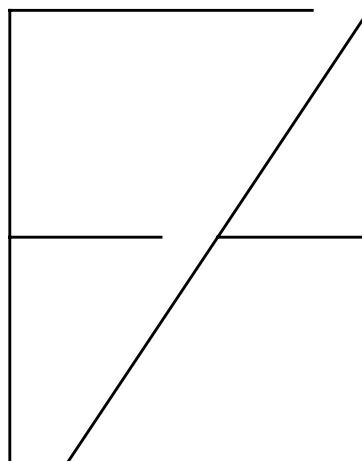
TR-14-99-06

Technical Report

Computer
Science Department

Operating Systems — IMMD IV

Friedrich-Alexander-University
Erlangen-Nürnberg, Germany



Virtual Private Computing

Martin Steckermeier, Franz Hauck

*Department of Computer Science IV, University of Erlangen-Nürnberg,
Martensstraße 1,
91058 Erlangen, Germany*

Martin.Steckermeier@informatik.uni-erlangen.de
Franz.Hauck@informatik.uni-erlangen.de

Abstract

There is still no widely accepted system to exploit the capabilities offered by the tremendously grown number of computers connected to the Internet or to large intranets. Missing abstractions as well as deficiencies of software development prevent the effective use of more than two computers by standard applications.

The architecture that is presented in this report overcomes these problems by creating a vision of a Virtual Private Computer, which combines all participating machines and presents them as a large, omnipresent computer. Beside several services that are essential to realize this system, a programming model is introduced that supports the development of applications that are composed of components, which can be automatically distributed over the virtual computer. Several examples will show how this architecture is capable of realizing all usual distribution paradigms, and how it supports fully distributed applications. Moreover the paper indicates how applications and special areas like mobile computing can benefit from such a system.

1. Introduction

Although the performance of single computers has risen immensely over the last years, users are still faced with applications that require more resources than their current computer is able to provide. This is especially true for mobile computers. Due to size, weight, and energy restrictions these devices are mostly one step behind the performance of current stationary computers. But in most cases the peak performance that is demanded by the user is only needed for short periods of time, followed by longer periods in which the computer is nearly idle. From a user's point of view, it would be ideal to extend the performance of his local machine at some moments by using other machines that are currently idle. This would be possible by delegating parts of the computation to other computers. On one hand, this would improve the peak performance the user can use at that moment, and on the other hand the other participating machines would be used much more effectively.

The Internet with its huge number of connected computers offers an interesting environment for such resource sharing. But although it is the largest existing resource pool, there is still no widely accepted system that makes use of these resources by distributing applications onto several hosts.

Especially mobile users are faced with a problems that was originally created by software development which was driven by the influence of personal computers or workstations. Applications are typically aimed to run on a single computer and exclusively use local code and data. Although there are mechanisms like NFS or AFS, they are mostly restricted to a local network. In the case of worldwide mobility, users have only very restricted access to code and data that resides on a dedicated host, e.g., at their home department, and therefore usually have to carry around everything on their own mobile device. The situation becomes even worse if the user has no mobile computer but instead depends on a foreign one. In this situation, the user is usually limited to applications that are installed on this computer, but has no access to own programs and data, which lie at a home location.

To overcome this lack of location transparency and to support global resource sharing, we propose an architecture that establishes the notion of a *Virtual Private Computer* (VPC). Virtual private computing is comparable to time sharing, which was introduced in operating systems years ago to improve processor utilization. In analogy to time-sharing systems, which share the system's processor between different processes to improve processor utilization, a VPC shares distributed resources, like processors, memory, secondary storage, and networking capacity between different applications and users. Like time sharing, which creates the illusion of a virtual processor for every process, a VPC creates the vision of a virtual computer for every user. This virtual computer is dedicated to the user and is dynamically expanded to cover all necessary resources, especially those that hold code and data.

After a survey of existing paradigms of distributed computation, we will introduce the idea of *Virtual Private Computing* in Chapter 3. Our system consists of several services that make up the environment for VPC applications which

have to be installed on a participating machine. Additionally a programming model is needed that bases on components as its distribution entities. Their high reusability is exploited to improve code distribution and scalability. In Chapter 4, we will show that this architecture can be used to simulate all existing paradigms, how it can solve their problems, and how it supports fully distributed applications.

2. Overview of Existing Paradigms

To point out common problems of current applications and execution environments, we suppose the following application: On a large data structure, e.g., gained by scientific simulation or measurement, statistics have to be done which require lots of computing resources. The resulting data is rendered to an image that is displayed and can be interactively changed in size and point of view.

Depending on the computing environment and the application, different configurations are possible. The following criteria will be used to compare these configurations:

- How can the workload be divided between different hosts and which communication costs arise from this distribution.
- Does the user have to be aware of code or data locality, and does he have to care for code and data distribution.
- Is the user urged to know about static or dynamic properties of the computing environment, e.g., hostnames or the workload of participating machines.
- How difficult is it to start the application by the user.

One approach for running the application is to do the whole computation locally with code and data residing on the client computer. This approach, which is typical for personal computers, has the advantage that there are no communication costs for downloading that data. On the other side, there are huge requirements regarding the storage capacity and especially the computing performance of the client computer. If the data size exceeds a certain limit, computation is either impossible or too time consuming.

Some storage capacity can be saved, if data and code can be stored on a central fileserver that exports them, e.g., via a network filesystem like NFS, as shown in Figure 1. For the client computer only the real computation costs remain. On the other hand expenses for storage have then been replaced by high communication costs for transferring large volumes of data.

Another approach would be to shift the computation to a separate machine that does the statistics and rendering. Only the graphical representation is done by the client computer, which requires that the application can display its results remotely, e.g., by using a mechanism like X-Windows. Figure 2 shows this situation with a dedicated com-

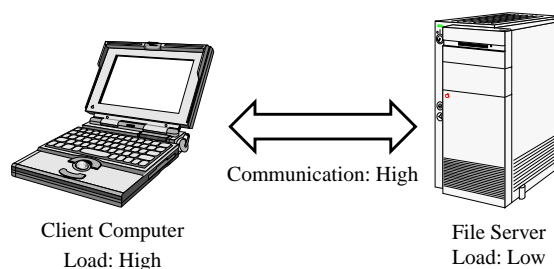


Fig. 1. Client-side computation

pute server and an additional fileserver that holds data and code.

Although the workload from the client computer has been reduced, this approach has several drawbacks, like e.g., the additional communication costs between the compute server and the client computer. Due to the nature of the problem with the need for an interactive manipulation of high resolution images, the communication between compute server and client computer requires a high bandwidth.

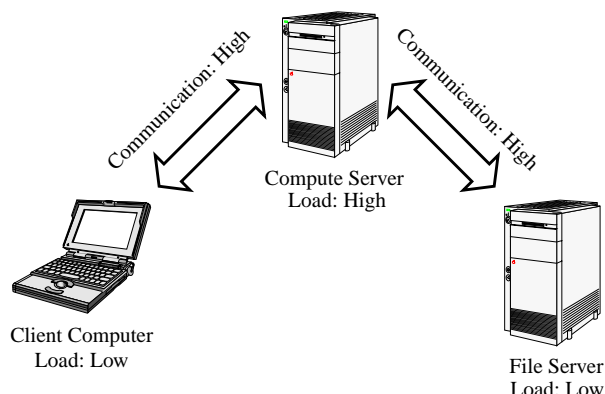


Fig. 2. Server side computation

But the more severe drawbacks lie in the lack of transparency for the user. The user has now the possibility but also the responsibility to find a computer that is appropriate for his computation. The decision has to be based on considerations about the possible computing performance of the machines that are available, their current workload, the location of the data and the communication bandwidth between the host holding the data, the compute server, and the client host. In most cases, the user will not be able to find an optimal decision that ensures best overall performance and at the same time minimizes costs. Moreover for the user it is awkward to start his application as it requires knowledge about the computing environment as well as additional permissions on the participating machines, which is typically not true for arbitrary machines on the

Internet. The user has to login to the compute server, which presumes a valid account on that machine and then has to start the computation in a way that it redirects the graphical output to his client computer.

An additional problem that may arise is a lack in location transparency for accessing the data. If the data for the computation resides on the client machine instead of a central, well known file server, the computing host might have no access to this data via mechanisms like NFS. This is equally true for the application code, that either has to be installed on the destination host or has to be accessed via NFS.

To reduce the high communication costs between compute server and client computer, the application has to be partitioned and distributed among both. In our example, the part doing the statistics remains on the compute server and sends its static result data to the rendering engine, which is now located at the client computer. However, this improvement is done at the expense of additional complexity for the user and the application developer. Now, the user has to care for the distribution of two applications, i.e., by installing code on two machines, starting the parts on that machines, and connect both correctly. The application developer must decide how to partition his application, based on considerations about the internal communication behavior. As both parts will run on different machines and therefore probably on different platforms, the developer is urged to provide different binaries and either he or the user is responsible for installing the right ones.

Several systems have been developed over the last years to support such client/server-like application structure. Typically these systems provide a transparent remote procedure call or remote method invocation together with an own object model. Examples for such systems are RPC [Ne81], RMI [Sun98], CORBA [OMG98a], and DCOM [EdEd98]. Nevertheless, none of these systems assists in distribution decision. Application developers still have to decide which objects belong to the server or to the client part of the application and users are still responsible for locating code, starting application parts and connecting them.

Partitioning and distribution can also be done automatically, e.g., by systems like Coign [Hunt98]. This system analyses the communication behavior of a binary DCOM application in several profiling runs and then divides it into client and server components. The application is then installed in its original binary form on both, the client and the server machine. The decision, whether a component should be started at the client or server side is done at runtime. Although this automatic and transparent partitioning algorithm relieves the application programmer from these decisions, in the current implementation of Coign, the cho-

sen distribution is static and can not react on changes in the computing environment and is limited to a distribution over two machines.

Another approach, especially for high performance computing is known as *Metacomputing*. Originally using only a few special high performance computers, metacomputing has been extended to use Networks of Workstations (NoWs) or even any willing computer connected to the Internet in form of so-called *Volunteer Computing*. According to the different types of machines, there is a broad spectrum of systems, from simple communication mechanisms like *PVM* or *MPI* to architectures like *Mentat* [Gri93] or *Legion* [Gri96]. Typically, these systems follow a fork-join approach, i.e., computing intensive work is split into parts which are distributed on all other machines. The originating machine then waits for the results before it continues its own computation.

Metacomputing experienced increasing popularity with the introduction of Java in 1995. Java allowed the distribution of code on potentially every computer connected to the Internet. Especially its support for heterogeneous architectures, the ability to download code dynamically and to run it safely on a host pushed the development of metacomputing frameworks like *Javelin* [ChCa+97], *Bayanihan* [Sar98], and *PopCorn* [Shm98]. In these systems arbitrary people can join and provide resources by simply downloading an applet that runs computations on behalf of other users. Work is usually sent to participating machines in form of so-called “computelets” that contain Java code and data. The code is executed on the destination machine and results are sent back. No or only very little communication should occur between these computelets for performance reasons. Moreover, due to the overhead of downloading Java bytecode every time and checking it locally via a bytecode verifier, only long lasting computations benefit from those frameworks. Therefore scalability in terms of the number of computelets as well as the number of participating hosts is limited.

As can be seen, to successfully exploit additional resources offered by other machines connected to the Internet, several properties have to be satisfied by applications and runtime environments:

- Applications must not be monolithic, but must consist of several independently distributable parts.
- Application code and data must be transparently accessible via a network filesystem, or it must be downloaded automatically in an efficient way.
- Starting distributed applications must be as easy and efficient as starting typical local applications, independent of where the user initiates the computation.

- Application parts must be distributed automatically with respect to performance and cost issues.
- The runtime environment must provide a location transparent communication mechanism between distributed parts.
- The whole system should support heterogeneous computing environments as is common in the Internet with its different participating hardware architectures and operating systems.

The next chapter will sketch an architecture that is capable of providing these properties in form of a so-called *Virtual Private Computer* (VPC).

3. The VPC Architecture

To overcome the deficiencies of existing systems our VPC architecture relies on components as distribution entities and a strict decoupling of component types and component implementations. We assume, that in the near future there will be a small number of standardization boards, which will define interfaces of commodity components, like for example spelling checkers, spreadsheets, and word processors. Furthermore, they will establish a system for uniquely naming these interfaces. On the other hand, there will be companies that will realize these components, probably in different languages and for different platforms. Applications will be built by exploiting this pool of already existing components.

Our VPC system uses these commodity components for software composition and provides the following additional properties:

- Components get bound to applications at runtime by mapping abstract component types to real implementations. Therefore no large, single binary has to be downloaded to run the application.
- Component implementations are chosen with respect to different criteria as for example efficiency. This allows an automatic adaption of the application to its environment, e.g., by using component code optimized for the actual hardware platform.
- Component code is downloaded dynamically and transparently for the user, if it is not already contained in a local component repository. Neither the user nor any system administrator has to care for code distribution.
- Implementations for commodity components can be installed locally to decrease the latency for most component instantiation requests. This property allows better scalability in the number of components that are used for an application and the number of hosts used for distribution.
- Component instances are distributed automatically to optimize the overall system performance and communi-

cate in a location transparent manner. User as well as administrators do not have to keep track of the workload of machines, or the bandwidth and latencies of networks. Additionally application developers need not care about distribution decisions either.

To realize this properties, the VPC architecture consists of a programming model, which describes components and applications built from components, and several services, which have to be installed on every participating machine.

3.1. VPC Services

Figure 3 gives an overview over the most important building blocks of the VPC system. These parts have to be installed on every participating machine. Their installation can either be done by dynamically downloading an applet, similar to the execution environment in volunteer computing systems like PopCorn [Shm98], or for more serious resource providers, by intalling special versions of these services manually.

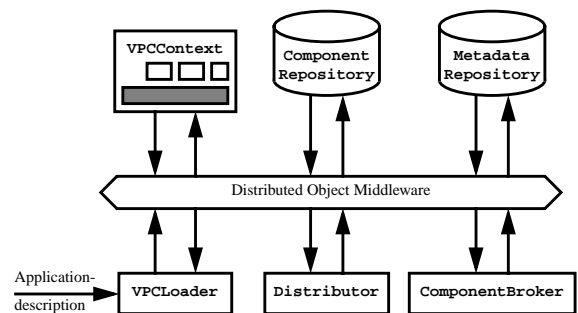


Fig. 3. An Architectural Overview

All communication in a VPC system is assumed to be done via remote method invocations. Therefore the central building block is a middleware, like CORBA, DCOM and Java RMI, providing a location transparent method invocation mechanism. These middleware products are usually accompanied by services like nameservers, e.g., CosNaming [OMG97] and JNDI [Sun99], interface repositories, and presumably extended services like transaction and persistence service, which are not explicitly shown in the figure.

The VPCLoader shown in the figure is only needed for starting VPC applications. From the application description that is fed into the VPCLoader, it can determine which execution environment is necessary to run the application. In case of Java components, the loader would start a VPCContext object in a Java Virtual Machine; in case of binary components it would start a binary application that provides a VPCContext object suitable for running the application components. The VPCContext objects, as shown in Figure 4, realize the execution environment consisting of a VPCInterpreter and the FactoryFinder.

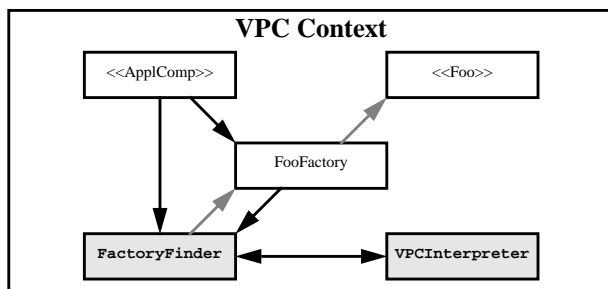


Fig. 4. A VPC Context hosting application components

Both are instantiated automatically at the time the VPCContext object is created. Finally the application description is handed over to the interpreter. It is the task of the interpreter to evaluate application and component descriptions and initiate the creation of initial components based on this description.

Both, the interpreter and all other application components have to use factory objects for the creation of new components. References to these factory objects can be obtained from the FactoryFinder. It gets the information about the component type and returns a reference to an already existing or a newly created factory object, which can either be located in the current context, in another context on the same machine, or even in a context on an arbitrary other machine. The reference, which is returned is then used to instantiate a new component of the required type in the same context as the factory object, i.e., factory objects determine the location of instantiation. Therefore the placement of factories is crucial for the overall system performance and for load balancing. The decision for the placement of factories is done by the so-called Distributor. The local Distributor instance communicates with other known distributors and tries to find an optimal distribution, based on a cost model and on information about the workload of participating machines, network latency and bandwidth, as well as code availability. The request to create a new factory is finally forwarded to the chosen host, which offered the lowest costs for instantiating a component of the required type.

As already mentioned, instantiation requests do not address specific implementations, but instead describe components that should be created. These descriptions at least contain a component type identifier. With this identifier, additional information can be retrieved from the MetadataRepository. This repository is an enhanced version of the interface repository, which is common in systems like CORBA. It does not only contain information about the interfaces that components support, but can also specify further properties like relationships to other components. An example

are connectors that specify associations between components. This information can be utilized by the Interpreter and the Distributor to collocate or dislocate components.

The actual mapping from an abstract component description to a real implementation is done by a local ComponentBroker. Typically, commodity components will already be contained in its local component repository, i.e., the broker can directly return information about this implementation. If no implementation can be found locally, the broker will try to download an appropriate component from another broker or an implementation server that can be specified in the application description. To reduce the latency for further use of these implementation, it can be inserted into the local component repository.

Beside these central building blocks, some more services will be necessary, but are not in the scope of this paper. To be actually useful, accounting services for code and resource usage have to exist as well as authentication mechanisms to be able to identify users and map user names to accounts. Monitors have to exist to measure workload of machines, and network bandwidth and latency to assist in distribution decisions. And last but not least, security services have to be used to protect user data when it is stored or transmitted.

3.2. The Application Description Language

In traditional component oriented systems, the software composition done by the application developer is evaluated at compile time and results in a monolithic binary. This is no longer true for VPC applications, as those are dynamically bound at runtime. In this system, the information about the internal structure of the application has still to exist. It is used by the previously introduced architectural components to control instantiation and distribution.

For expressing the necessary information, we developed an own language. Main responsibilities of this “composition code” are the description of component types and properties, and additionally the employment of such components as well as their interconnections.

No component code or other application functionality is expressed in the description language, as components should be realized in the language that is best suited for the problem they solve. Therefore no full programming language is necessary. Instead we encode the descriptions in XML [XML98], which is already used in form of XMI (XML Metadata Interchange, [OMG98b]) to describe CORBA datatypes and interfaces. One important advantage of XML is that it allows to freely annotate components, interfaces, etc. with additional information. These annotations can for example describe communication behavior in

terms of number and/or size of messages that are exchanged by components over a single connection to assist in distribution decisions.

Another important advantage of XML is the already existing, well established transmission system in form of the World-Wide Web. Application descriptions can be downloaded over the Internet by simple HTTP requests using WWW browsers that are installed on nearly every host.

3.2.1 Component Description

The component model used for VPC applications is similar to the OMG component model [OMG99] and concentrates on the concepts of interfaces, components and connectors.

Interfaces

Similar to CORBA IDL, interface descriptions comprise datatypes, exception types, attributes and signatures of methods. Multiple inheritance of interfaces is supported, with the same restrictions as in CORBA regarding overriding and overloading [OMG98a].

Events

Events as a simple message passing mechanism are also supported by the language. Specifications for events comprise three parts: the type of an event, event sources and event sinks. Whereas event sources and sinks are part of the component description, event types can be declared beforehand, similar to data types.

Components and Connectors

Components are the units of distribution in a VPC system and are therefore the central entity of the specification language.

Instead of a dedicated component interface, components support one or more of the previously defined interfaces. A component must be able to hand out references to objects that implement the supported interfaces at runtime. Interfaces specify which services the component offers to other components and how these services can be accessed.

Similar to interfaces, a component can contain several event sources. Effectively, an event source specified by a name and an event type is nothing more than a special type of interface, that allows to register one or more event sinks. Events are pushed to these event sinks by method invocations.

The counterpart of interfaces and event sources are connectors. Connectors specify relationships between components and therefore possible communication paths. Beyond that, mandatory connectors determine which other components and connections are essential for a component to function

correctly. Three different types of connectors are distinguished:

- **Interface Connectors:**
This connector type declares, that the specified component will communicate with another component that offers a service with this specific interface, i.e., interface connectors are directed, typed communication paths.
- **Component Connectors:**
These connectors express a “uses” or “needs” relationship to another component. They do not state whether or how the related components will communicate. One of their main tasks is to bundle interface connections, i.e., express that a set of connectors is connected to interfaces of the one, specified destination component.
- **Event Connectors:**
Event connectors are the means to specify event sinks. Binding an event connector to an event source results in registering the sink object there.

Like interfaces, components can also be grouped into a hierarchical inheritance relationship. Basic specifications for components can be “subclassed” by successively adding new interfaces, event sources or sinks, and connectors, to create specialized versions. Every time a component of the basic type is requested by an application, a component implementation conforming to this type or some subtype can be returned by the component broker.

3.2.2 Application Description

VPC applications are composed of components, where components can be categorized into two groups. On one hand there are commodity components. Different implementations for these components can exist, provided by different vendors. On the other side there are glue components, that make up the remaining functionality which can not be realized by standard components and have to be programmed by the application developer. Unlike other component frameworks like DCOM, components are not bound to a single binary, but are dynamically bound at runtime. The application description contains no application code but mere structural and meta information, specifying component types to be used or connections between these components and is interpreted at runtime.

As the description does not address implementations but only component types, the runtime environment is free to choose an implementation that conforms to this type and additionally fulfills special requirements, e.g., regarding efficiency or security. On the other hand, due to this late binding, the application description can not predict and specify the complete resulting structure of the application, as the structure depends on the chosen components and their subcomponents. Therefore, it can only specify which initial components should be started, how they should be

connected and further application properties, like e.g., locations from where component code can be download.

4. Usage Scenarios

Based on this architecture and the description language, all of the mentioned traditional distribution paradigms can be realized and even enhanced.

4.1. Client-Side Computation

In this scenario, the application is formed of one single component. The application description mentions this component only. Starting the application by giving the description to the `VPCLoader`, results in an instantiation of this component at the local machine.

This procedure is nearly equivalent to starting a local binary application, as there is no distribution and therefore no usage of remote resources. The most important difference is that the user is no longer responsible for installing code locally or locating it on a NFS server. Therefore, this approach is more like starting an applet, where the application code is dynamically downloaded from a HTTP server. In case of a VPC application, the component code is either found in the local component repository, which results in a smaller latency to start the application, or is also downloaded from a known component server.

4.2. Remote Computation

In this case, the application consists of two main components. The actual computation is done by the first component, whereas the other one is responsible for creating a graphical user interface. Additionally, a glue component might be necessary to connect the GUI and the computational component.

The application description addresses at least the GUI and the glue component, whereas the computational component might be requested by the glue component at runtime and an appropriate implementation gets chosen by the component broker. Depending on the load of the participating machines, the VPC system now decides where to place the computational component and initiates the instantiation at the chosen host.

Compared to starting a remote application manually, this approach has several advantages:

- Users do not need to care for code distribution in a possibly heterogeneous environments.
- Users are relieved from distribution decision, as this is done transparently by the system, which automatically reacts on changes in the computing environment, workload, network latency, and bandwidth. Users do not even

need to know about the structure of the underlying resource pool.

- Components are started by the VPC system. The user does not need to login to another machine and start parts manually. This also eliminates the need for an additional user account on the chosen compute server.

4.3. Client/Server Computation

The same benefits apply to client/server computation. In contrast to remote computation, in client/server applications not only a graphical user interface is located on client side, but parts of the application functionality can be shifted onto the client computer. In our example of Chapter 2, the client part could comprise the rendering machine that gets its data from the compute server that still does the statistics.

4.4. Fully Distributed Computation

All previously mentioned scenarios distribute parts of the application over at most two distinct computers. The mechanisms of a VPC allow more sophisticated distribution strategies as soon as applications consist of a larger number of components.

There may be several situations, where a distribution onto more than two computers is beneficial. One area are highly computing-extensive applications. Like in classical meta-computing, the actual computation can be broken down into pieces and distributed onto a network of workstations or another resource pool, but partitioning is now done in form of components that are able to communicate between one another. Based on information about the components, the VPC system tries to find an optimal distribution to maximize resource usage and minimize costs. Beside information about the current computing environment, these considerations take also into account whether there are any locality constraints or collocation relationships that might not be obvious to the user.

More interesting to the general user than metacomputing is the full location transparency that is gained by the system and can be exploited for general purpose applications. As long as the user can login to a participating host of the VPC or connect his own, possibly portable computer to the VPC, and is moreover able to access the application description, he can now start his application wherever he is. The system then cares for locating code and data, makes distribution decisions and starts components on the chosen hosts. Even in a complete foreign environment, starting an application that is mainly built from commodity components is more efficient than downloading a complete monolithic binary and starting it locally, as most of the component implementations can be picked up from a local component repository that can contain versions optimized for the local architecture.

Especially in case of mobile computing, the load distribution done by the VPC allows to start large applications on a mobile host. Depending on the performance of this mobile device, more or less components will be started on other participating machines. This allows to use smaller and less powerful mobile devices to actually solve large problems, which not only reduces size and price, but also the average power consumption, which is still a major problem in mobile computing.

Similar to mobile hosts, thin clients can benefit from VPCs, as more computing-intensive applications can automatically be distributed onto compute servers. In case of thin clients, beside the services shown in Chapter 3 only a basic operating system has to be installed locally. Component code is downloaded dynamically and stored in the local component repository, which effectively learns which components are typically used on this host. If components are realized in languages like Java, the local machine can decide whether the code should run on the Java Virtual Machine, or should be compiled to native machine code based on usage information. Typical commodity components, which will be used more frequently will be chosen to be compiled into native code and stored in binary form, to improve efficiency and reduce latency for starting this components. This approach is superior to using a just-in-time compiler that converts bytecode to native code every time an application is started.

Compared to existing thin client systems, the VPC offers the same benefit, that there is only a single central software pool that has to be maintained, but it also reduces the latency to start applications, as code can be chosen from the local repository. Furthermore, it improves performance by compiling code and it distributes applications if the thin client is not able to run the whole computation.

5. Conclusion

We showed in this paper that existing abstractions are not sufficient to effectively use more than two computers by a single application. In systems that want to exploit more resources, e.g., by using all computers connected to local network, new mechanisms and strategies have to be provided that exceed those that are currently provided by typical operating and middleware systems.

Our architecture allows to access distributed resources in form of a *Virtual Private Computer*. Applications are composed of components that are distributed based on information about available resources, workload of participating machines, and information about network bandwidth and latency. The composition is done in a separate language, independent of the languages which are used to realize the components themselves. The description language based on

XML allows to cleanly separate composition code from functional code and is interpreted at runtime. As components are only addressed by their types in these descriptions, they can be mapped to optimized implementations for the actual platform by a component broker. By utilizing a local component repository, implementations for commodity components can be picked up locally and do not have to be downloaded from a component provider, which drastically reduces the latencies for starting components and thus allows more fine-grained distribution. Code that is not found locally is transparently downloaded, relieving the user from code distribution issues.

Our next step will be a prototype implementation of the basic services and the interpreter to show the usefulness of the presented architecture. The prototype should provide a testbed to find out, which essential information about the execution environment has to be gathered and which application properties have to be known to realize useful distribution strategies.

Moreover, research will have to be done in how additional properties, like security or persistence can be specified for dedicated components, although the final application structure is not known at composition time.

References

- ChCa+97 Bernd O. Christiansen, Peter Cappello, et.al. Javelin: Internet-Based Parallel Computing Using Java. Department of Computer Science, University of California, Santa Barbara, June 1997
- EdEd98 Guy Eddon, Henry Eddon. Inside Distributed COM. Microsoft Press, 1998.
- Gri93 Andrew S. Grimshaw. The Mentat Computation Model, Data-Driven Support for Object-Oriented Parallel Processing. Department of Computer Science, University of Virginia, May 1993
- Gri96 Andrew S. Grimshaw, Wm. A. Wulf. Legion — A View From 50,000 Feet. Department of Computer Science, University of Virginia, August 1996
- Hunt98 Galen C. Hunt, Automatic Distributed Partitioning of Component Applications. Ph.D. Dissertation, Department of Computer Science. University of Rochester, July 1998
- Ne81 Bruce J. Nelson. Remote Procedure Call, Ph.D. Dissertation, Department of Computer Science, Carnegie-Mellon University, 1981
- OMG97 Object Management Group: CORBAservices: Common ObjectServices Specification. OMG Document formal/97-12-02, November 1997
- OMG98a Object Management Group: The Common Object Request Broker: Architecture and Specification, Rel. 2.2. February 1998. OMG Document formal/98-02-01, February 1998

- OMG98b Object Management Group: XML Metadata Interchange. OMG Document ad/98-10-05, October 1998
- OMG99 Object Management Group: CORBA Components. OMG Document ad/98-10-05, March 1999
- Sar98 Luis F. G. Sarmenta. Bayanihan: Web-Based Volunteer Computing Using Java. MIT Laboratory for Computer Science, Cambridge, April 1998
- Shm98 Shmulik London. POPCORN — A Paradigm for Global-Computing. Master Thesis, Institute of Computer Science, The Hebrew University of Jerusalem, June 1998
- Sun98 Java Remote Method Invocation - Distributed Computing for Java, Whitepaper Sun Microsystems, September 98, <http://www.javasoft.com/marketing/collateral/javarmi.html>
- Sun99 The Java Naming and Directory Interface Tutorial. Sun Microsystems, March 1999. <http://www.javasoft.com/products/jndi/tutorial/index.html>
- XML98 Extensible Markup Language (XML) 1.0. W3C Recommendation, February 1998