Design und Implementierung eines Frameworks zur Evaluierung Phasenbasierter Agenten

R. Kapitza, Martin Geier

Juni 2000 TR-I4-00-08

Technical Report

Computer
Science Department

Operating Systems — IMMD IV

Friedrich-Alexander-University
Erlangen-Nürnberg, Germany

Design und Implementierung eines Frameworks zur Evaluierung Phasenbasierter Agenten

Studienarbeit im Fach Informatik

vorgelegt von

Rüdiger Kapitza

geboren am 10. November 1975 in Erlangen

Angefertigt am

Institut für Mathematische Maschinen und Datenverarbeitung (IV)
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	*Prof. Dr. rer. nat. Fridolin Hofmann* *Dipl. Inf. Martin Geier*

Beginn der Arbeit:	*1. Dezember 1999*
Abgabe der Arbeit:	*9. Juni 2000*

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die aktuellen Entwicklungen im Bereich der Computertechnologie zeigen einen klaren Trend auf: Immer mehr rechnergestützte Systeme unterschiedlicher Bauweise werden miteinander über Netzwerke verbunden. Den vorläufigen Endpunkt dieser Tendenz stellen Haushalts- und Mobilfunkgeräte mit einer eigenen Internetadresse dar. Als Konsequenz dieser enormen Expansion der Vernetzung steigt der Bedarf nach verteilten Anwendungen, die eine Nutzung der Ressourcen des Netzwerkes ermöglichen.

In großen Netzwerken, wie dem Internet, gibt es bedingt durch die Art und den Standort des einzelnen Teilnehmersystems starke Unterschiede in der Qualität der Netzanbindung. Viele Systeme sind nur temporär über analoge oder digitale Telefonleitungen mit dem Internet verbunden. Solche leistungsschwachen Verbindungen und überlastete Netze führen bei der Nutzung von Diensten mit erhöhtem Datenaufkommen zu langen Übertragungszeiten. Ein Beispiel das häufig in der Praxis vorkommt, ist die Kommunikation mit einer entfernten Datenbank.

Wird eine Anfrage an eine entfernt Datenbank gestellt, so kann das zu übertragende Resultat der Anfrage eine sehr große Datenmenge bedeuten. Oft werden die Daten aber nur zum Teil benötigt und müssen deshalb weiterverarbeitet werden, um das gewünschte Ergebnis zu erhalten. Versuche die Datenmenge und ihre Nachbearbeitung durch gezielte Anfragen und Einschränkungen zu reduzieren gelingt meist nur bedingt und ist letztlich von der Schnittstelle der Datenbank abhängig.

Eine Verkürzung der Übertragungszeit durch die Reduktion der Datenübertragung bietet das Konzept des mobilen Agenten. Grundidee dieses Konzeptes ist es, dass Programme dynamisch den Ort ihrer Ausführung in Abhängigkeit von ihren Zielen und den Eigenschaften des Umfelds selbstständig bestimmen können. Im Falle des Beispiels wird ein mobiler Agent sich zu dem entfernten Rechner transferieren und vor Ort die Anfragen an die Datenbank stellen. Abhängig von dem Resultat der Anfrage, ist der Agent nun selbstständig in der Lage weitere Anfragen zu stellen, einen anderen Rechner mit einer Datenbank aufzusuchen oder mit den Endergebnissen zu seinem Ursprungsort zurückzukehren. Der Vorteil dieses Konzeptes ist es, dass nicht mehr einzelne Anfragen und große Mengen von Rohdaten übertragen werden müssen, sondern nur noch das wesentlich kleinere Programm des mobilen Agenten und die erzielten Endergebnisse. Wie man an Hand dieser kurzen Darstellung erkennen kann, bietet das Konzept des mobilen Agenten weit mehr Möglichkeiten als nur eine Reduktion der zu übertragenden Datenmenge. Es wird als neuer flexibler Ansatz für verteilte Anwendungen betrachtet [Lan98]. Obwohl es schon zahlreiche Anwendungen und Systeme gibt, die auf mobilen Agenten basieren, ist ihre Funktionalität und Größe meist gering. Ein Grund dafür ist, dass mit zunehmender Komplexität der Aufgabenstellung, parallel die Größe des mobilen Agent wächst und dadurch der Vorteil einer reduzierten Datenübertragung aufgehoben wird. Eine viel diskutiertes, komplexes Anwendungsgebiet von mobilen Agenten bildet der electronic commerce.

Mit der zunehmenden Angebotsvielfalt im Internet wird es für Käufer immer schwieriger den günstigsten Preis für ein bestimmtes Produkt zu ermitteln. Anstatt die Abfrage und Bewertung der einzelnen Angebote selbst vorzunehmen, kann ein Anwender diese Aufgabe an einen mobilen Agent übertragen. Ausgestattet mit den entsprechenden Informationen über das Produkt, sucht der Agent mögliche Anbieter auf. Falls es erhältlich ist, verhandelt er über den Preis und bewertet das Angebot. Hat der mobile Agent das günstigste Angebot gefunden, so kann er das Produkt erwerben.

Sicherlich benötigt der mobile Agent nur einen Teil seiner Funktionalitäten bei einem Anbieter, der das gewünschte Produkt nicht führt oder dessen Angebot überteuert ist. Der Transport nicht verwendeter Funktionalitäten erhöht aber unnötig die Datenübertragung und mindert den Nutzen von mobilen Agenten. Einen Weg die Datenübertragung auf ein Minimum zu reduzieren und damit auch die effiziente Lösung von komplexen Aufgabenstellung zu ermöglichen, bietet das Konzept des Phasenbasierten Agenten [GH99]. Dieses Konzept teilt einen mobilen Agenten in verschiedene Phasen auf. Jede Phase repräsentiert eine abgeschlossene Teilaufgabe der Gesamtaufgabenstellung. Der Agent migriert nicht mehr als Einheit zu einem entfernten Rechner, sondern nur noch die aktive Phase. Alle anderen Phasen und eine Workflowengine, der den Ablauf des gesamten Agenten steuert, können stationär am Ursprungsort bleiben. Die aktive Phase agiert autonom vor Ort, bis sie ihre Aufgabe erfüllt hat. Die Workflowengine des Agent entscheidet dann auf Grund der Endergebisse der Phase, mit welcher Phase die Verarbeitung fortgesetzt werden soll und überträgt sie zum Zielsystem. Das Konzept des Phasenbasierten Agenten verhindert, auf Grund der expliziten Entscheidung über die Migration jeder einzelnen Phase, den Transfer von nicht benötigten Funktionalitäten und bietet die Möglichkeit komplexe Probleme auf einfache Weise zu strukturieren.

1.2 Thema und Gliederung der Arbeit

Das Ziel dieser Arbeit ist der Entwurf und die Implementierung eines Frameworks[1] für Phasenbasierte Agenten. Die Implementierung erfolgt in der Programmiersprache Java[2] und verwendet die Agentenplattform[3] Voyager [OBJ99] als Basis. Aufbauend auf diesem Framework sind verschiedene Agenten für Messzwecke zu entwickeln. Ziel der Messungen ist eine Evaluierung des Phasenbasierten Agenten und ein Vergleich dieses Konzeptes mit dem des monolithischen mobilen Agenten, in Bezug auf Ausführungsgeschwindigkeit und Datenübertragung.

Als Grundlage gibt Kapitel 2 eine erweiterte Beschreibung der in der Motivation eingeführten Konzepte, sowie eine Einführung in die verwendete Laufzeitumgebung Voyager. Im ersten Teil von Kapitel 3 wird das Design und die Implementierung der einzelnen Komponenten des Frameworks vorgestellt. Der zweite Teil des Kapitels beschreibt die Bestimmung der Messgrößen und den Entwurf verschiedener Agenten für die Messungen. Kapitel 4 stellt die Messungen vor und bewertet die erzielten Ergebnisse. Die abschließende Zusammenfassung und eine Beschreibung der möglichen Ergänzungen bilden Kapitel 5.

1. Architekturmuster, das erweiterbare Vorlagen für Anwendungen in einem Bereich zur Verfügung stellt [BRJ99].
2. Java ist eine objektorientierte Programmiersprache die von Sun Microsystems entwickelt wurde.
3. Eine spezielle Laufzeitumgebung in der Agenten ausgeführt werden können.

2 Grundlagen

2.1 Konzept des mobilen Agenten

2.1.1 Definition des mobilen Agenten

Die Basis für das aktuell stark diskutierte Konzept des mobilen Agenten bildet das Paradigma des Softwareagenten. Dieser allgemeine Überbegriff wird von Lange [Lan98] aus zwei verschiedenen Perspektiven definiert. Eine sehr einfache Definition beschreibt die Leistungen, die ein Softwareagent, im Folgenden kurz als Agent bezeichnet, gegenüber dem Benutzer erbringen soll. Die Definition fordert, dass der Agent den Benutzer gemäß seiner Bedürfnisse unterstützt. Außerdem soll er den Benutzer vertreten und in dessen Auftrag handeln. Diese Anforderungen charakterisieren den Begriff eines Agenten jedoch bei weitem nicht ausreichend. Eine zweite Definition beschreibt deshalb alle wichtigen Eigenschaften, über die ein Agent verfügen muss, um die gewünschten Leistungen zu erbringen:

Ein Agent

- ist ein Objekt oder ein Programm, das für seine Abarbeitung eine Ausführungsumgebung benötigt.
- ist reaktiv, d. h. er ist in Lage Veränderungen in seiner Umgebung zu erkennen und kann sein Verhalten entsprechend anpassen.
- ist autonom in seiner Entscheidungsfindung.
- handelt Ziel basiert.
- ist in seiner Lebensdauer zeitlich nicht beschränkt.

Folgende Eigenschaften sind oft bei Agenten zu finden, sind aber nicht zwingend erforderlich:

Ein Agent

- ist in der Lage mit anderen Agenten zu kommunizieren.
- kann den Ort seiner Ausführung jederzeit wechseln.
- lernt aus Erfahrung.

Die Mobilität eines Agenten wird in dieser Definition als optionale Eigenschaft betrachtet. Auf Grund der in der Motivation beschriebenen aktuellen Entwicklungen, steigt die Bedeutung der Mobilität von Agenten jedoch stetig und führt zu dem Konzept des mobilen Agenten.

Definition eines mobilen Agenten:

Ein mobiler Agent ist ein Objekt oder Programm, das die Fähigkeit besitzt den Ort seiner Ausführung zu wechseln. Bei einem solchen Ortswechsel werden die Daten, der Programmcode und unter Umständen der Ausführungszustand des mobilen Agenten übertragen. Sämtliche Entscheidungen, einschließlich der Migration zu einem anderen System, trifft der mobile Agent autonom.

Sicherlich stellt das Konzept des mobilen Agenten eine Erweiterung des Agentenbegriffs um die Eigenschaft der Mobilität dar, aber es ist auch eine Zusammenfassung der ursprünglich in der Definition des Agenten geforderten Eigenschaften auf den Begriff der Autonomie festzustellen. Boger zeigt an Hand eines Beispiels, wie weitreichend der Begriff der Autonomie ist [Bog99]. Ein Agent kann zur Lösung einer Aufgabe eine Liste von Rechneradressen der Reihe nach abarbeiten und sich von Rechner zu Rechner bewegen. Die gleiche Aufgabe löst ein anderer Agent indem er, basierend auf bisher gemachten Erfahrungen, die Entscheidung trifft, dass nur bestimmte Rechner zur Lösung seiner Aufgabe von Relevanz sind, die er aufsucht. Beide Fälle sind für Boger mobile Agenten, die autonom entscheiden, auch wenn die Art der Entscheidungsfindung völlig verschieden ist. Der zweite Lösungsansatz erscheint jedoch intelligenter, da er erlerntes Wissen verwendet. Wie dieses Beispiel zeigt, müssen mobile Agenten nicht intelligent handeln und damit können Intelligenz und Mobilität als orthogonale Konzepte betrachtet werden, die aber miteinander kombinierbar sind.

2.1.2 Vorteile mobiler Agenten

Die bisher in der Motivation angesprochenen Vorteile von mobilen Agenten sollen an dieser Stelle vertieft und weitere Vorteile vorgestellt werden:

<u>Verkürzung der Reaktionszeit auf Ereignisse:</u>

In vielen Anwendungsbereichen, speziell bei Echtzeitsystemen, spielt die Reaktionszeit eines Programmes eine wichtige Rolle. Oft ist die Latenz von Netzwerkverbindungen zu groß um zeitkritische Entwicklungen von einem entfernten Rechner aus zu beobachten und auf Ereignisse zu reagieren. Eine Lösung dieser Problematik bietet ein mobiler Agent, der das zu beobachtende System aufsucht. Lokal vor Ort ist er in der Lage auf Ereignisse rechtzeitig und in angemessener Weise zu reagieren.

<u>Asynchrone Bearbeitung von Aufgaben:</u>

Da bei der Migration eines mobilen Agenten nicht nur der Zustand des Agenten zum Zielsystem übertragen wird, sondern auch sein Programmcode, ist ein mobiler Agent von seinem Ursprungsort unabhängig. Diese Tatsache bietet für mobile rechnergestützte Systeme einen großen Vorteil, denn sie verfügen oft nur über eine temporäre Netzwerkanbindung auf Grund von hohen Verbindungskosten sowie technischen Gegebenheiten. Aufgaben die mit bisherigen Lösungsansätzen längere Verbindungszeiten an das Netzwerk erfordern, sind deshalb oft teuer oder nicht erfüllbar. Der Einsatz von mobilen Agenten kann die Verbindungszeiten in solchen Fällen erheblich reduzieren.

Eine Aufgabe wird an einen mobilen Agenten übertragen der seinen Ursprungsort verlässt, um sie unter Ausnutzung der Ressourcen des Netzwerks zu erfüllen. Ein mobiles System kann nach dem Transfer des Agenten zu einem anderen Rechner seine Netzwerkverbindung abbauen, denn während der Bearbeitung der Aufgabe wird die Verbindung nicht benötigt. Sobald das mobile System sich wieder mit dem Netzwerk verbindet und der mobile Agent seinen Auftrag erfüllt hat, kehrt er zu diesem mit den Ergebnissen zurück.

<u>Reduktion der Datenübertragung:</u>

Wenn ein mobiler Agent von einem Rechner zu einem anderen Rechner wechselt, muss der Zustand des Agenten und sein Programmcode übertragen werden. Ist der Agent vor Ort, kann er auf die Dienste des Zielrechners zugreifen und die erhaltenen Daten lokal verarbeiten. Nach der Erfüllung seiner Aufgabe überträgt der mobile Agent die Ergebnisse an seinem Ursprungsort oder kehrt mit den Ergebnissen dorthin zurück.

Bisher werden Anfragen und ihre Ergebnisse einzeln zwischen entfernten Kommunikationspartnern über das Netzwerk übertragen. Die Folge sind viele entfernte Zugriffe und abhängig vom Dienst, die Übertragung von großen Rohdatenmengen. Der Einsatz von mobilen Agenten verlagert das verarbeitende Programm zu den Daten, statt die Daten zur Verarbeitung zu übertragen und reduziert somit den Datentransfer.

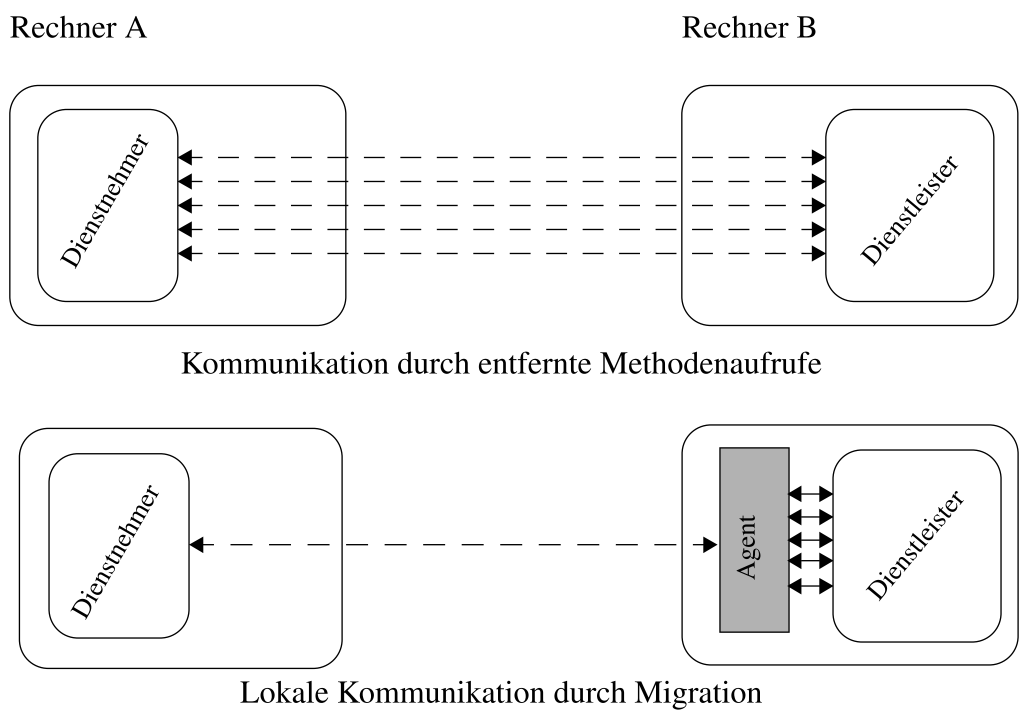

Abb. 2.1 Unterschied zwischen lokaler und entfernter Kommunikation

<u>Plattformunabhängiges Konzept für verteilte Anwendungen:</u>

Wie in der Motivation angedeutet ist mit der Zunahme der Vernetzung parallel eine steigende Heterogenität der Teilnehmersysteme verbunden. Der transparente Ablauf einer verteilten Anwendung auf verschieden Systemen erfordert aber eine einheitliche Schnittstelle. Einer der Gründe für die schnelle Verbreitung der Programmiersprache Java ist ihre Plattformunabhängigkeit, das bedeutet Programme die in Java geschrieben sind, können auf fast allen aktuellen Systemen ausgeführt werden. Die Plattformunabhängigkeit von Java basiert auf einer Laufzeitumgebung die auf dem Betriebssystem des Rechners aufsetzt, in der die Programme ausgeführt werden. Mobile Agenten benötigen ebenfalls eine solche Umgebung, die ihre Ausführung und

Migration zwischen verschiedenen Rechneren ermöglicht. Fast alle neueren Agentensysteme sind deshalb in der Programmiersprache Java implementiert oder Skriptsprachen wie Tcl [GCK+96]. Diese Tatsache ermöglicht den Einsatz von mobilen Agenten als Konzept für verteilte Anwendungen auf allen Rechnersystemen.

2.1.3 Nachteile mobiler Agenten

Das Konzept des mobilen Agenten birgt aber auch eine Reihe von Problemen in sich. Im Folgenden eine Zusammenfassung der Wichtigsten:

<u>Sicherheitsprobleme:</u>

Obwohl Sicherheitsprobleme in allen Bereichen der Netzwerkprogrammierung ein ständiges Thema sind, wird besonders bei mobilen Agenten die Sicherheit als Schwachpunkt angesehen. Damit ein mobiler Agent beispielsweise seinen Auftraggeber vertreten kann, benötigt er verschiedene, zum Teil persönliche, Informationen über seine Interessen und Gewohnheiten. Diese Informationen sind im Zustand des Agenten gespeichert. Ein mobiler Agent wird zur Erfüllung seiner Aufgabe verschiedene Rechner besuchen. Dort besitzt die Ausführungsumgebung die Kontrolle über den Agenten. Die Ausführungsumgebung ist also in der Lage den Agenten in verschiedener Weise zu manipulieren. Die im Zustand des Agenten enthaltenen Informationen können kopiert, verändert oder zerstört werden. Ähnliches ist mit dem Code eines mobilen Agenten möglich. Bisher gibt es kein Verfahren, das die Agenten vor bösartigen Zugriffen der Ausführungsumgebung schützt.

Ein mobiler Agent kann aber auch seine Ausführungsumgebung oder andere Agenten schädigen. Ein einfaches Beispiel wäre ein Agent, der soviel Betriebsmittel in Anspruch nimmt bis die Ausführungsumgebung überlastet ist. Einige Agentenplattformen verfügen bereits über Mechanismen mit denen die Rechte der mobilen Agenten eingeschränkt werden können und damit dem System einen gewissen Schutz bieten, aber diese Mechanismen befinden sich noch in der Entwicklung [AD99].

<u>Komplexität und Größe:</u>

Werden umfangreiche Aufgaben an mobile Agenten übertragen, die sie in intelligenter Weise lösen sollen, so nimmt die Größe und Komplexität von mobilen Agenten rasch zu. Boger beschreibt hohe Entwicklungskosten und aufwendige Programmieraufgaben als Nachteile von intelligenten mobilen Agenten [Bog99]. Diese Sichtweise wird hier nicht geteilt, da die Entwicklungskosten bei ähnlichen Anforderungen unter Verwendung eines anderen Konzeptes für verteilte Anwendungen vergleichbar sind. Der Vorteil einer reduzierten Datenübertragung hängt jedoch eng mit der Größe eines mobilen Agenten zusammen. Steigt die Größe eines Agenten über ein gewisses Maß an, geht dieser Vorteil verloren.

2.1.4 Anwendungsbereiche von mobilen Agenten

Das Konzept des mobile Agenten bietet auf Grund seiner Flexibilität ein sehr breites Anwendungsspektrum. Es ist als Alternative zu den bisherigen Programmierparadigmas für verteilte Anwendungen zu betrachten. Schattenseite der allgemeinen Verwendbarkeit dieses Konzeptes ist das Fehlen von spezifischen Anwendungsfällen, bei denen mobile Agenten die einzige mögliche Lösung verkörpern [KG99]. Trotz dieser Tatsache gibt es viele Anwendungsbereiche in denen mobile Agenten gegenüber traditionellen Ansätzen im Vorteil sind. Im Folgenden ein kleine Auswahl für Anwendungsbereiche dieser Art:

<u>Electronic commerce:</u>

Ein stark expandierender Bereich in der Entwicklung des Internets ist der Verkauf von Waren aller Art. Die Zunahme der Anbieter und die Vielfalt der angebotenen Produkte macht eine Marktübersicht sowie Preisvergleiche für Anwender nahezu unmöglich. Hier bietet sich die Unterstützung des Anwender durch mobile Agenten an. Der Anwender erteilt dem mobilen Agenten den Auftrag für den Kauf eines bestimmten Produktes. Der mobile Agent sucht selbstständig eine große Anzahl von Anbietern auf und führt einen Preisvergleich durch. Je nach Anbieter und Produkt ist es vorstellbar, dass der mobile Agent über den Preis verhandelt oder zusätzliche Leistungen in den Vergleich der Angebote einbezieht. Hat der Agent das beste Angebot gefunden, kann er es kaufen und bezahlen.

<u>Verteilte Informationssuche:</u>

Ein wichtiges Werkzeug zum Auffinden von Informationen im Internet sind Suchmaschinen. Bisher erzeugen Suchmaschinen ihre Indizes in dem sie große Datenmengen aus dem Internet herunterladen, um sie weiter zu verarbeiten. Durch den Einsatz von mobilen Agenten lässt sich die Menge der übertragenen Daten stark einschränken, indem sie die Webserver einzeln aufsuchen und die Daten vor Ort auswerten. Danach werden nur noch die Ergebnisse zur Suchmaschine übertragen und in den Datenbestand eingefügt.

<u>Verteilte Berechnung:</u>

Aufgaben deren Berechnung das Vielfache der Rechenleistung eines Systems über eine lange Zeitdauer benötigt, erfordern normalerweise spezielle Multiprozessorsysteme oder Rechnercluster. Mobile Agenten ermöglichen die Lösung solcher Aufgaben durch gewöhnliche Systeme. Eine Aufgabe wird dafür in verschiedene Teilaufgaben gegliedert, die jeweils von einem Agenten übernommen werden. Diese Agenten verteilen sich auf mehrere Rechner eines Netzwerkes und lösen ihre Teilaufgabe. Wird ein System für andere Anwendungen benötigt oder ist hoch belastet, kann ein Agent sogar seine Berechnung unterbrechen und auf einem geringer genutzten System fortsetzen.

2.2 Erweiterung des mobilen Agenten zum Fragmentierten Agenten

Bisher wird ein mobiler Agent als ein Objekt oder eine Menge von Objekten betrachtet, die eine Migrationseinheit bilden. Dieser Ansatz verursacht bei komplexen Aufgabenstellungen große Migrationseinheiten. Dadurch geht einer der Hauptvorteile des mobilen Agentenparadigmas, die Reduktion der Datenübertragung, verloren. Eine Lösung für dieses Problem bietet die Erweiterung des mobilen Agentenkonzeptes zum Fragmentierten Agenten, unter Verwendung des Fragmentierten Objektmodells.

2.2.1 Das Fragmentierte Objektmodell in AspectIX

Das Fragmentierte Objektmodell ist eine Anpassung des traditionellen monolithischen Objektes an die Erfordernisse von verteilten Systemen. Im Gegensatz zu monolithischen Objekten können fragmentierte Objekte zeitgleich verschiedene Aufenthaltsorte innerhalb eines verteilten Systems besitzen. Diese Eigenschaft wird, bei der Middlewareplattform AspectIX [HBG+98], durch eine Teilung des monolithischen Objektes in sogenannte Fragmente erreicht. Die einzelnen Fragmente stehen miteinander in Verbindung um die Semantik eines Objektes zu repräsentieren. Jedes Fragment kann die gesamte Funktionalität des Objektes oder nur eine Teil davon implementieren. Damit ein Objekt mit einem fragmentierten Objekt in Interaktion treten kann, muss ein Fragment lokal im Adressraum des Objektes vorhanden sein. Besitzt das lokale Fragment nicht die angeforderte Funktionalität des verteilten Objektes, so wird die Anfrage transparent an ein Fragment weitergeleitet, das sie erfüllen kann. Fragmente können auch als Objekte betrachtet werden, mit dem einzigen Unterschied, dass alle Fragmente die Identität eines verteilten Objektes aufweisen.

Für die Implementierung des Fragmentierten Objektmodells wird eine Middlewareplattform benötigt, die eine einheitliche Identität des verteilten Objekts ermöglicht. Zu diesem Zweck wird eine Zweiteilung des Fragments in Fragmentinterface und Fragmentimplementation vorgenommen. Das Fragmentinterface ist ein automatisch generiertes Objekt das von der Middle-

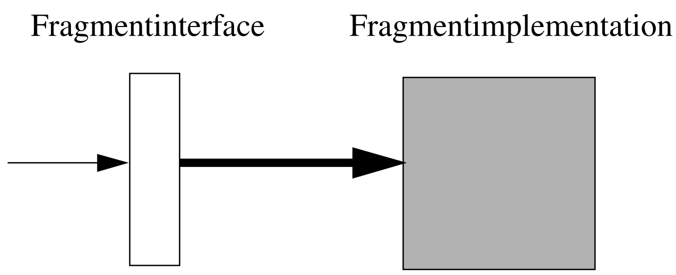

Abb. 2.2 Bestandteile eines Fragments

wareplattform bereitgestellt wird. Es dient als stabile Referenz für andere Objekte. Alle Aufrufe am Fragmentinterface werden an die Fragmentimplementation weitergeleitet. Die Implementation definiert die Semantik des Fragments. Durch die Aufteilung des Fragmentes ist es möglich die Fragmentimplementation dynamisch auszutauschen, ohne das Referenzen auf das Fragment ihre Gültigkeit verlieren.

2.2.2 Der Fragmentierte Agent

Durch die Übertragung des Fragmentierten Objektmodells auf das Programmierparadigma des mobilen Agenten entsteht das *Fragmented-Object Agent Model (FOAM)* [GH99]. Dieser Ansatz trennt sich von der Sichtweise eines mobilen Agenten als Migrationseinheit. Ein Fragmentierter Agent besteht aus einer beliebigen Anzahl von mobilen Fragmenten, die sich auf verschiedene Orte verteilen können.

Die Tatsache, dass alle Fragmente mobil sein können, beeinflusst das Design eines Fragmentierten Agenten entscheidend. Eine wichtige Rolle für die Aufteilung der benötigten Funktionalität in Fragmente spielt die Kommunikation zwischen den Fragmenten und die Größe der mobilen Fragmente. Damit die Kommunikation zwischen den einzelnen Fragmenten auf ein Minimum reduziert wird, werden eng miteinander zusammenhängende Anforderungen möglichst in einem Fragment implementiert. Mobile Fragmente enthalten im Optimalfall außerdem nur Funktionalitäten die von einer lokalen Kommunikation profitieren. Funktionalitäten die selten benötig werden oder sehr groß sind, können in eigenen immobilen Fragmenten implementiert werden.

Sicherlich erscheint der Entwurf eines Fragmentierten Agenten auf Grund seiner Anforderungen als komplizierter, doch bietet er viele Vorteile wegen seiner Flexibilität:

- *FOAM* hebt die starre Trennung zwischen dem Einsatz von entfernter Kommunkation oder mobilen Agenten auf. Beide Mechanismen können je nach Nutzen zur Lösung einer Aufgabe verwendet werden. Ob ein Fragment den Ausführungsort wechselt kann dynamisch zur Laufzeit entschieden werden. Einer der Hauptvorteile von mobilen Agenten, die Reduktion der Datenübertragung, bleibt durch die freie Skalierbarkeit der Migrationseinheit selbst bei komplexen Aufgabenstellungen erhalten.
- Ein mobiler Agent ist abhängig von seiner Laufzeitumgebung. Bei einem Systemausfall bedeutet dies den Verlust aller bisher durch den Agenten erzielten Ergebnisse. Ein Fragmentierter Agent ist normalerweise auf verschiedenen Rechner verteilt. Wichtige Informationen können deshalb durch Replikation von Fragmentimplementationen auf mehreren Rechner aufbewart werden. Die replizierten Informationen müssen über ein spezielles Protokoll konsistent gehalten werden. Fällt einer der Aufenthaltsorte des Fragmentierten Agenten aus, kann er dennoch seine Ausführung normal fortsetzen, da er auf die Replikate der verlorenen Fragmente zurückgreifen kann.
- Viele Anwendungsbereiche erfordern den Vergleich von Informationen unterschiedlicher Systeme. Bisher musste ein mobiler Agent, der aus einer Migrationseinheit besteht, jedes System nacheinander aufsuchen. Mit einem Fragmentierten Agenten können die einzelnen Informationsquellen parallel verarbeitet werden, indem sich der Agent durch die Auf-

teilung in Fragmente auf mehrere Systeme verteilen. Die Anzahl der Ausführungsorte des Fragmentierten Agenten ist dabei flexibel und wird dynamisch durch den Agenten entschieden.

2.3 Das Konzept des Phasenbasierten Agenten

Das Konzept des Phasenbasierten Agenten entwickelt, aufbauend auf dem Ansatz des Fragmentierten Agenten, ein Verfahren zu Lösung komplexer Aufgabenstellungen durch verteilte mobile Agenten. Dabei stellt dieses Konzept nicht nur ein Architekturmuster dar, sondern definiert auch eine Entwurfsmethode. Da der Entwurfsprozess eines Phasenbasierten Agenten für das Verständnis seines Aufbaues wichtig ist, wird dieser zuerst beschrieben:

Für den Entwurf eines Phasenbasierten Agenten muss die zu erfüllende Aufgabestellung in zeitlich aufeinanderfolgende Teilaufgaben, sogenannte Phasen, gegliedert werden. Eine Phase kann dabei verschiedene Anfangsparameter besitzen mit denen sie gestartet wird und sollte nach ihrer Ausführung ein Endergebnis liefern. Kann in bestimmten Situationen im Voraus keine eindeutige Abfolge der Phasen festgelegt werden oder gibt es verschiedene alternative Abläufe, so müssen Bedingungen definiert werden. Diese entscheiden zur Laufzeit, basierend auf den Endergebnissen der vorangegangenen Phasen und sonstigen Randbedingungen, mit welcher Phase die Ausführung fortgesetzt wird.

Ein Phasenbasierter Agent besteht aus einer stationären Komponente und einer beliebigen Anzahl von mobilen Fragmenten und Phasen. Da das Konzept des Phasenbasierten Agenten auf dem des Fragmentierten Agenten aufbaut, entspricht eine Phase im wesentlichen einer Fragmentimplementation. Die Aufgabe der stationären Komponente ist es, alle Phasen zu verwalten und durch eine Workflowengine den generellen Ablauf des verteilten Agenten zu steuern. Die Workflowengine implementiert zu diesem Zweck die beim Entwurf festgelegten Bedingungen zur Abfolge der Phasen und entscheidet durch sie, welche Phasen in die mobilen Fragmente eingesetzt werden. Die mobilen Fragmente bilden die aktiven Einheiten des Agenten und werden durch die Phasen gesteuert. Eine Phase entscheidet dabei unabhängig über jede Aktion des Fragments einschließlich der Migration zu einem anderen System. Hat die Phase ihre Teilaufgabe erfüllt, so übermittelt sie ihre Endergebnisse an die Workflowengine (Abb. 2.3). Die Workflowengine wertet die Ergebnisse der Phase aus und entscheidet nun, welche Phase in das mobile Fragment eingesetzt wird. Dieser Ablauf wiederholt sich immer wieder, bis die Aufgabenstellung erfüllt ist.

Im Folgenden soll nun verdeutlicht werden, warum das Konzept des Phasenbasierten Agenten vor allem bei komplexen Aufgabenstellungen eine Reduktion der Datenübertragung erreichen kann.

Bei einer komplexen Aufgabenstellung müssen meist eine Vielzahl von Funktionen implementiert werden, die nur in besondern Situationen benötigt werden. Wie aus der bisherigen Darstellung des Ablaufs eines Phasenbasierten Agenten klar wird, werden nur Phasen in ein mobiles Fragment eingesetzt und damit zu einem entfernten System übertragen, die auch tatsächlich für die Erfüllung der Aufgabenstellung benötigt werden. Bei monolithischen mobilen Agenten wird dagegen immer der gesamte Agent zu einem Zielsystem migriert und damit auch Funktionalitäten die nicht benötigt werden.

Ist eine Aufgabenstellungen hinreichend komplex, kann die Workflowengine eines Agenten relativ groß sein. Bei monolithischen Agenten ist sie immer fester Bestandteil der Migrationseinheit des Agenten und muss zu einem entfernten System transferiert werden. Dies muss bei einem Phasenbasierten Agenten nicht der Fall sein, hier besteht die Möglichkeit die Workflowengine stationär an einem Ort zu belassen. Natürlich müssen dann die Endergebnisse der Phasen zur Workflowengine übertragen werden, damit diese über den weiteren Ablauf entscheiden kann. Bleibt die Workflowengine bei komplexen Aufgabenstellungen vor Ort lässt sich eine Reduktion der Datenübertragung erreichen, da die Workflowengine wesentlich größer ist als die zu übertragenden Endergebnisse der Phasen.

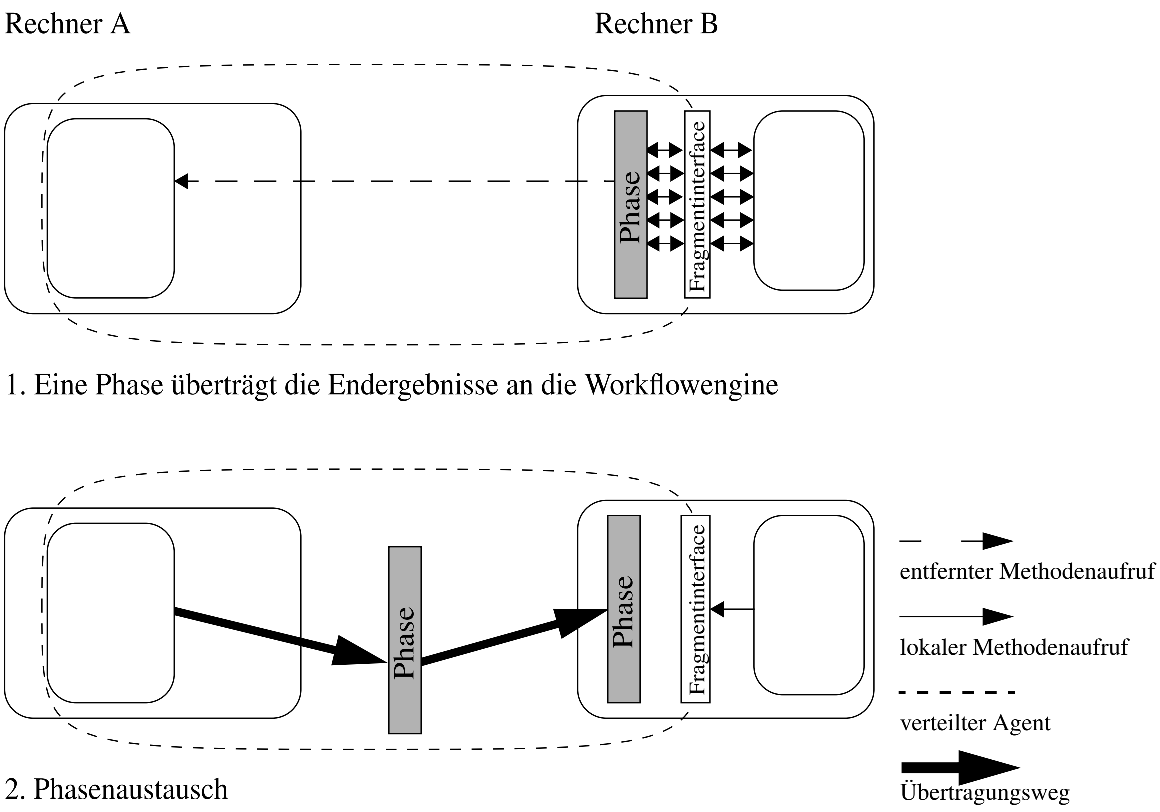

Abb. 2.3 Austausch einer Phase

Das Konzept des Phasenbasierten Agenten soll nun durch den Entwurf eines Agenten zum Kauf eines beliebigen Produktes veranschaulicht werden.

Folgenden Phasen werden bei dieser Aufgabenstellung identifiziert:

(1) Suche nach einem Anbieter

(2) Anfrage nach dem gesuchten Produkt

(3) Verhandlungen

(4) Kauf

Das Ziel der ersten Phase ist es, Adressen von verschiedenen Anbieter zu ermitteln. Hat die Phase eine oder mehrere Adressen von Anbietern gefunden, so transferiert sie sich zum System eines Anbieters und teilt der Workflowengine das Erreichen ihres Endzustandes mit. Die zweite Phase wird durch eine Fragmentimplementation realisiert, die ermittelt, ob das gewünschte Produkt bei der Firma erhältlich ist. Die Parameter dieser Phase sind die genaue Produktbezeichnung und zusätzliche Eigenschaften über die das Produkt verfügen soll. Abhängig von den Er-

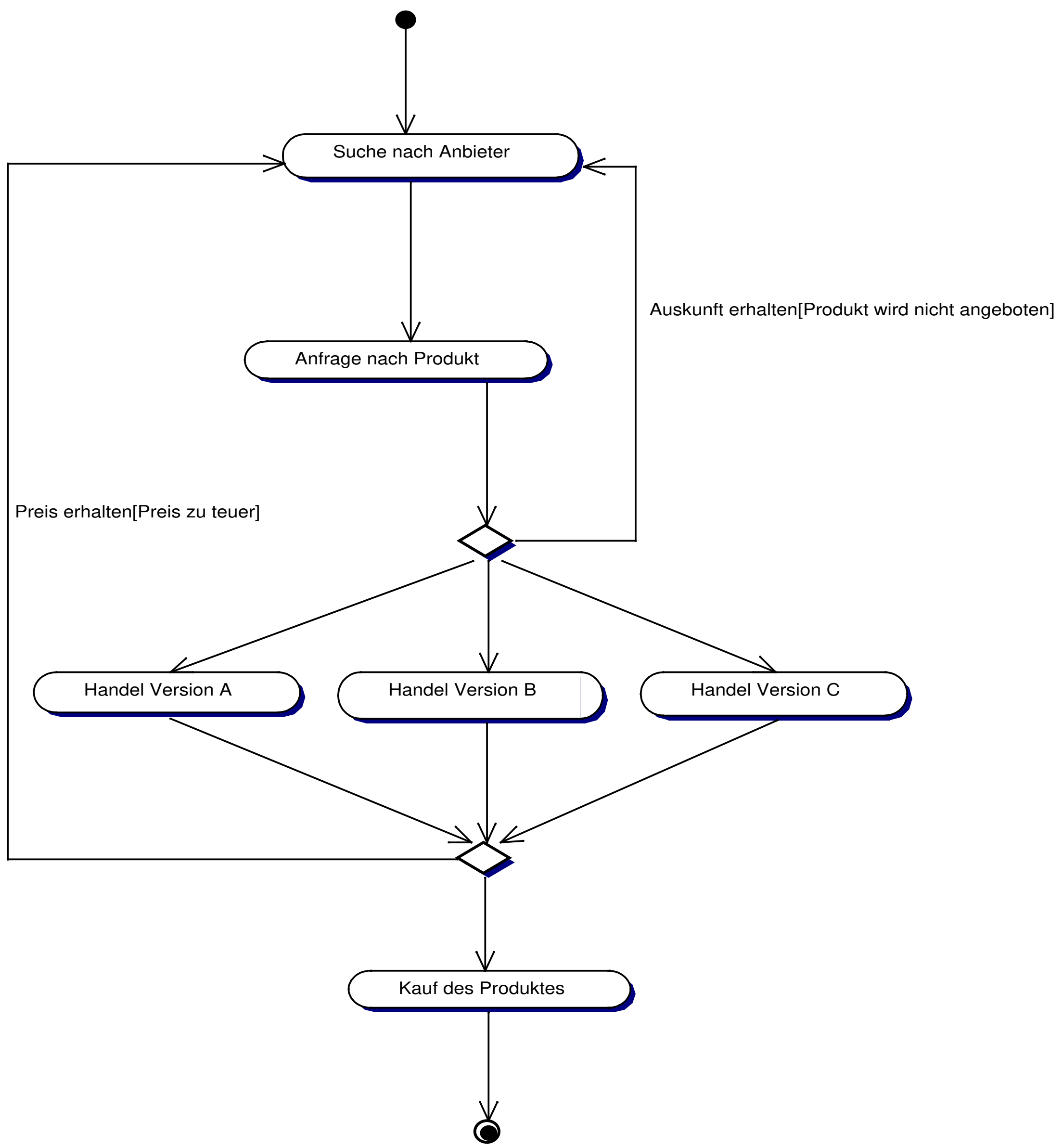

Abb. 2.4 Workflow eines Kaufagenten

gebnissen der Anfragephase wird die Workflowengine entscheiden, mit welcher Phase die Verarbeitung fortgesetzt wird. Ist das Produkt im Angebot, wird die Anfragephase durch eine Implementierung der Verhandlungsphase ersetzt.

Das in UML[1]-Notation dargestellte Aktivitätsdiagramm (Abb. 2.4) verdeutlicht den Ablauf des Workflows. Jeder Aktivitätszustand repräsentiert eine Implementation einer Phase und jeder Zustandsübergang stellt einen Phasenaustausch dar. Die Aufgabe ist erfüllt sobald der Endzustand erreicht ist.

Für den Anbieter ist der Austausch der einzelnen Phasen völlig transparent und seine Referenz auf den Agenten bleibt durch die Aufteilung des Fragments in Fragmentimplementation und Fragmentinterface stabil erhalten.

Das Aktivitätsdiagramm verdeutlicht mit den drei Phasen zur Verhandlung, dass es möglich ist eine Auswahl zwischen verschiedenen Phasen zu treffen, die grundsätzlich die gleiche Aufgabe erfüllen, sich jedoch im Detail unterscheiden. Diese Möglichkeit, verschiedene Versionen einer Phase zu entwickeln und nur die vor Ort benötigte zu transferieren, reduziert die Datenübertragung gegenüber einem monolithischen Agenten, der alle Fälle berücksichtigen muss.

Speziell bei der letzten Phase des Workflows, dem Kauf des Produktes, wird noch ein wichtiger Vorteil des Phasenbasierten Agenten offensichtlich. Konventionelle mobile Agenten enthalten immer alle nötigen Daten und Informationen zur Erfüllung ihrer Aufgabe. Bei einem Kaufagenten schließt dies unter Umständen die Kreditkartennummer ein. Solche wichtigen Informationen können von einem unbekannten System ausspioniert werden, ohne dass der Agent etwas davon merkt. Phasenbasierte Agenten verfügen immer nur über den Teil der Informationen der für die aktuelle Phase relevant ist. Im Falle des Beispiels wird die Information der Kreditkartennummer erst übertragen wenn sicher ist, dass der Kauf stattfindet. Zu diesem Zeitpunkt muss jedoch ein gewisses Vertrauen in das Anbietersystem gesetzt werden.

2.4 Der Voyager ORB[2]

Das von *ObjectSpace* entwickelte Produkt Voyager ist eine zu Java kompatible Laufzeitumgebung zur Programmierung verteilter Systeme. Es vereinigt unterschiedliche Konzepte der verteilten Programmierung und ermöglicht eine transparente Kommunikation zwischen entfernten Objekten.

Voyager wurde als Grundlage für die Entwicklung des Frameworks gewählt, da es im Gegensatz zu vielen Forschungsprojekten ein kommerzielles Produkt ist, dass sehr stabil und leistungsfähig ist. Anders als die meisten Agentenplattformen unterstützt Voyager außerdem nicht nur die Implementierung von mobilen Agenten, sondern verfügt auch über komfortable Mechanismen zur entfernten Kommunikation über Methodenaufrufe. Genau diese beiden Konzepte werden durch Fragmentierte Agenten miteinander kombiniert und für die Realisierung eines Frameworks für Phasenbasierte Agenten benötigt.

1. Unified Modeling Language
2. Object Request Broker

Dieser Abschnitt stellt die Basisfunktionalitäten von Voyager vor und beschreibt die für die Entwicklung von Phasenbasierten Agenten relevanten Mechanismen der Laufzeitumgebung. Eine vollständige Beschreibung aller Leistungen von Voyager ist der Dokumentation des Produktes zu entnehmen.

2.4.1 Entfernte Objekte

Voyager ermöglicht einen transparenten Zugriff auf entfernte Objekte durch lokale Stellvertreter, sogenannte Proxies. Im Gegensatz zu RMI[3] oder Corba[4] werden diese aber nicht zur Übersetzungszeit erstellt, sondern können dynamisch zur Laufzeit erzeugt werden. Damit ein Proxy ein entferntes Objekt vertreten kann, benötigt er die gleiche Schnittstelle wie das zu vertretende Objekt. Die Programmiersprache Java bietet die Möglichkeit Schnittstellen durch Interfaces zu definieren. Ein Interface ist eine Menge von Methodendeklarationen ohne Programmcode. Eine Variable vom Typ des Interface kann ein beliebiges Objekt als Wert annehmen, wenn das Objekt das Interface implementiert, d. h. alle Methoden des Interfaces beschreibt. Diese Eigenschaft wird bei der Erstellung von Proxies genutzt, denn sie implementieren das gleiche Interface wie das zu vertretende Objekt. Alle Methodenaufrufe leitet der Proxy weiter an das eigentliche Objekt. Befindet sich ein Objekt mit den zugehörigen Proxy in der gleichen Laufzeitumgebung, so werden alle Argumente wie bei einem normalen Methodenaufruf in Java behandelt. Sind Proxy und Objekt auf unterschiedliche Systeme verteilt, werden die Argumente des Methodenaufrufes serialisiert und übertragen. Im Falle des entfernten Methodenaufrufes entspricht das einem *pass by value*, der Übertragung der gesamten Objekte. Falls die Argumente aber sehr groß sind, kann man Maßnahmen treffen, die es ermöglichen, nur einen Proxy des Argumentes zu übertragen. Rückgabewerte von Methodenaufrufen werden vom Ursprungsobjekt an den Proxy weitergeleitet, der sie an das aufrufende Objekt übergibt. Treten Fehler bei dem entfernten Objekt auf, so werden diese ebenfalls transparent an das aufrufende Objekt über den Proxy weitergeleitet.

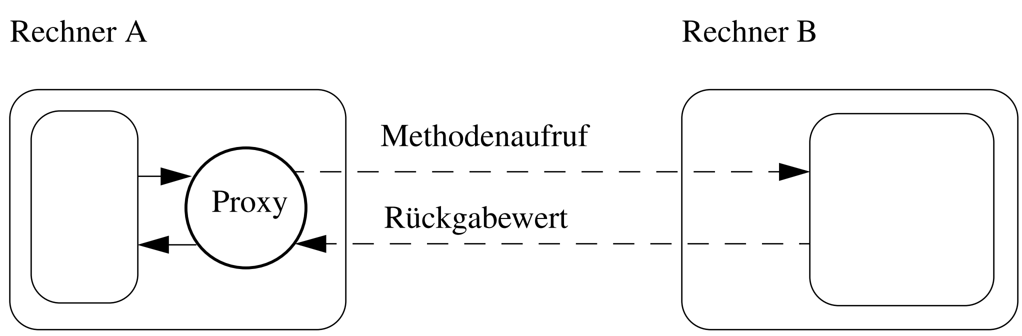

Abb. 2.5 Kommunikation über einen Proxy

3. Remote Methode Invocation
4. Common Object Request Broker Architectur

2.4.2 Namensdienst

Ein Namensdienst ermöglicht, über einen Namen, den einfachen Zugriff auf die Referenz eines entfernten Objekts. Die Laufzeitumgebung Voyager stellt über Funktionen der Klasse *Namespace* einen komfortablen Namensdienst zur Verfügung. Soll ein Objekt für entfernte Systeme erreichbar sein, wird es mittels der Methode *bind()* an einen Namen gebunden. Ein entferntes Programm kann mit dem Namen durch die Funktion *lookup()* einen Proxy auf das Objekt erhalten. Der Befehl *unbind()* entfernt den Verweis auf ein Objekte aus dem Index des Namensdienst.

2.4.3 Verteilte Garbage Collection

Nicht mehr benötigte Objekte werden durch die Garbage Collection der Java-Laufzeitumgebung automatisch gelöscht, sobald keine Referenz mehr auf sie verweisen. Dieser Mechanismus funktioniert nur für Programme, die in einer Laufzeitumgebung ausgeführt werden, da die Garbage Collection von Java keine entfernten Referenzen überprüfen kann. Voyager verfügt über eine verteilte Garbage Collection die entfernte Referenzen berücksichtigt. Jede Voyager-Laufzeitumgebung registriert alle entfernten Referenzen. In festen Zeitintervallen wird eine Zusammenfassung aller gelöschten und neu erzeugten Referenzen an die Systeme der betroffene Objekte verschickt, die sich im letzen Intervall ereignet haben. Die Empfängersysteme können aus der Zusammenfassung ermitteln, wieviele entfernte Referenzen auf ein lokales Objekt noch bestehen. Verfügt ein Objekt über keine entfernten Referenzen mehr, so kann die Voyager-Laufzeitumgebung seine eigne lokale Referenz auf das Objekt entfernen und sobald keine sonstige lokale Referenz mehr auf das Objekt verweist, wird es von der Java Garbage Collection gelöscht.

2.4.4 Dynamic Aggregation

Voyager bietet mit Dynamic Aggregation einen Mechanismus der eine beliebige Anzahl von Objekten, sogenannte Facetten, mit der Referenz eines primären Objektes verbindet. Ziel einer solchen Aggregation ist die Erweiterung der Funktionalität des primären Objektes. Die Klasse *Facets* verfügt zu diesem Zweck über verschiedene statische Methoden, die es ermöglichen einem primären Objekt neue Facetten hinzuzufügen oder auf die bereits mit dem primären Objekt verbundenen Objekte zuzugreifen. Da der Objektverbund durch die Laufzeitumgebung verwaltet wird, muss weder das primäre Objekt noch die Facette angepasst oder verändert werden. Der Verbund aus primären Objekt und Facetten ist aus Sicht des Systems eine Einheit und wird durch die Garbage Collection gelöscht sobald keine Referenz auf das primäre Objekt mehr vorhanden ist. Einen Spezialfall bilden sogenannte transiente Facetten. Sie können sofort nach der Erfüllung ihrer Aufgabe durch das System gelöscht werden. Beispiele für solche transienten Facetten sind die in Abschnitt 2.4.5 und 2.4.6 vorgestellten Facetten *Agent* und *Mobility.* Diese Facetten können einem beliebigen Objekt dynamisch Mobilität verleihen. Ansonsten lassen sich Facetten nicht wieder aus einem Aggregat entfernen und können selber nicht über eigene Facetten verfügen.

2.4.5 Objektmigration

Eine wichtige Funktionalität für die Programmierung von verteilten Systemen ist die Mobilität von Objekten. Beliebige Objekte können unter Voyager von einem System zu einem anderen verlagert werden. Eine *Mobility* Facette wird zu diesem Zweck an das betroffene Objekt angefügt. Diese Facette ermöglicht eine aus der Sicht des Objektes transparente Migration ohne das Veränderung am Objekt vorgenommen werden müssen. Das Ziel der Migration kann entweder in Form einer Adresse oder als Referenz auf ein entferntes Objekt an die Methode *moveTo()* übergeben werden. Damit auch nach der Verlagerung eines Objektes ein transparenter Zugriff auf das Objekt möglich ist, muss es indirekt über einen Proxy referenziert werden. Ein Proxy verliert auch nach der mehrfachen Migration eines Objektes nicht seine Verbindung zum Objekt. Aufrufe, die der Proxy an den ursprünglichen Ort des Objektes weiterleitet, lösen dort eine Ausnahme als Rückgabewert aus, die den neuen Aufenthaltsort des Objektes beinhaltet. Der Proxy kann den Aufruf nun an den richtigen Ort zustellen und erreicht somit das Objekt. Alle öffentlichen Methoden eines mobilen Objektes, die unter Verwendung eines Proxies aufgerufen werden können, sollten synchronisiert sein. Diese Maßnahme verhindert die Ausführung von Methoden des Objektes während seiner Migration. Das mobile Objekt wird zusammen mit allen Objekten die es referenziert, serialisiert und zum Zielort übertragen. Ist bei der Migration kein Fehler aufgetreten kann das mobile Objekt durch die Garbage Collection am Ursprungsort gelöscht werden. Bisher wurde beschrieben wie ein Objekt transparent von einem Ort zu einem anderen übertragen werden kann, oft ist es aber sinnvoll das ein Objekt über seine Migration informiert wird. Implementiert ein Objekt das Interface *IMobile*, so wird es über vier verschieden Phasen des Transfers unterrichtet. Die erste Methode *preDeparture()* wird unmittelbar vor einer Migration aufgerufen und informiert über den Aufenthaltsort und das Ziel. Da die Migration eines Objektes von außen angestoßen werden kann, bietet diese Methode die Möglichkeit eine Ausnahme auszulösen, die den Vorgang der Migration abbricht. Die nächste Methode *preArrival()* wird aufgerufen sobald das Objekt das Zielsystem erreicht hat, aber bevor es im System für andere Objekte sichtbar ist. Sobald der Transfer erfolgreich verlaufen ist, wird die Funktion *postArrival()* aufgerufen. Der letzte Methodenaufruf *postDeparture()* erfolgt an der veralteten Kopie am Ursprungsort, beispielsweise um Aufräumarbeiten zu ermöglichen.

2.4.6 Mobile Agenten

Ähnlich der *Mobility* Facette bietet Voyager eine *Agent* Facette, die ein Objekt dynamisch zu einem mobilen Agenten erweitert. Mit dem Aufruf der Methode *Agent.of()* kann einem Objekt eine *Agent* Facette angefügt werden. Der Zugriff auf die Methoden der Facette erfolgt über das Interface *IAgent*.

Wie in der Definition eines mobilen Agenten gefordert, stellt ein mobiler Agent unter Voyager eine autonome Einheit dar. Ein mobiler Agent ist deshalb in seiner Lebensdauer nicht an andere Objekte gekoppelt, d.h. er wird selbst dann von der Garbage Collection nicht gelöscht, wenn keine Referenz durch andere Objekte mehr auf ihn verweist. Der Agent kann seine Autonomie aber aufgeben, indem er die Methode *setAutonomous()* mit dem Parameter *false* aufruft. Danach wird der Agent vom System wie ein normales Objekt behandelt.

Außerdem ist ein mobiler autonomer Agent nicht nur in der Lage seinen Aufenthaltsort zu wechseln, sondern kann am Zielort auch seine Ausführung fortzusetzen. Die vollständige Migration eines mobilen Agenten besteht aus dem Transfer von Daten, Programmcode und Ausführungszustand. Voyager realisiert eine vereinfachte Variante, die sogenannte schwache Migration. Diese Art der Migration transferiert nur den Programmcode und die Daten eines mobilen Agenten. Grund für eine solche Einschränkung ist der hohe technische Aufwand der mit Migration des Ausführungszustandes verbunden ist. Die Konsequenz dieser Vereinfachung ist ein Verlust an Transparenz der Migration. Der mobile Agent muss über einen Methodenaufruf am Zielort neu gestartet werden und kann nicht innerhalb eines Methodenrumpfes von einer Anweisung zur nächsten einen Ortswechsel vollziehen. Die von der *Mobility* Facette bekannte Methode *moveTo()* wird zu diesem Zweck bei der *Agent* Facette um die Argumente für einen Neustart am Zielort erweitert. Diese Argumente bestehen aus einer öffentlichen Methode des mobilen Agenten und ihren Parametern. Der Verlust des Ausführungszustandes nach der Migration bringt aber auch noch andere Nachteile mit sich. So sollte eine Migration zum Beispiel nicht innerhalb einer Kette von rekursiven Aufrufen erfolgen, da Informationen über vorangegangenen Aufrufe nach einem Transfer verloren sind. Es ist also nicht möglich an den Anfang einer Rekursion zurückzukehren. Ist der Programmcode des Agenten auf dem Zielsystem nicht vorhanden, muss er von einem anderen Ort geladen werden. Damit das Zielsystem des Agenten weiß woher der Programmcode des Agenten geladen werden soll, verfügt die *Agent* Facette über die Methode *setResouceLoader()* mit der ein Resource Loader festgelegt werden kann, der auf den Programmcode des Agenten verweist. Bei der Ankunft des Agenten wird zuerst die *Agent* Facette geladen und falls sie über einen Resource Loader verfügt, dieser in das System integriert.

3 Design und Implementierung

3.1 Das Framework

3.1.1 Überblick

Ziel des ersten Abschnitts dieses Kapitels ist die Entwicklung eines Frameworks für das in Abschnitt 2.3 vorgestellte Konzept des Phasenbasierten Agenten. Ein Framework ist eine erweiterbare Vorlage für Anwendungen in einem bestimmten Bereich. Es definiert die Architektur der Anwendung, die jeweiligen Zuständigkeiten der einzelnen Komponenten, sowie den Kontrollfluss [GHJ+96].

Der Entwurf eines Frameworks verlangt die genaue Analyse des Anwendungsbereiches. Funktionalitäten die von allen Anwendungsfällen des Frameworks benötigt werden, sollten durch das Framework implementiert werden. Diese Maßnahme reduziert den Entwicklungsaufwand für die einzelnen Anwendungsfälle. Für die Implementierung aller anderen Funktionalitäten muss das Framework flexible Schnittstellen bieten, die eine Anpassung an die spezifischen Anforderungen der Anwendungsfälle ermöglichen.

Obwohl das Konzept des Phasenbasierten Agenten schon eine grobe Struktur der einzelnen Komponenten vorgibt, sind für den Entwurf des Frameworks die Anforderungen des Konzeptes und des Anwendungsbereiches erneut zu betrachten. Dies ist nötig da der Entwurf eines Frameworks nach den bereits beschriebenen Gesichtspunkten erfolgen sollte.

Funktionalitäten die von allen Anwendungsfällen eines Phasenbasierten Agenten in gleicher Weise benötigt werden:

- Verwaltung der Phasen
- Mobilität der Fragmente
- Übertragung der Endergebnisse einer Phase an die Workflowengine
- Transparenter Austausch von Phasen

Anforderungen und Funktionalitäten, für die das Framework Schnittstellen oder Möglichkeiten zur Erweiterung und Spezifizierung bieten sollte:

- Funktion der Phasen
- Mechanismen zur Entscheidung über einen Phasenaustausch
- Schnittstelle der Fragmente zu anderen Objekten

Unter Berücksichtigung der Aufteilung der Anforderungen lässt sich das Framework in einzelne Klassen und Komponenten gliedern, die in Abbildung 3.1 dargestellt sind. Die mit *My* beginnenden Klassen bilden dabei nur Platzhalter für die Klassen eines Anwendungsfalles des Frameworks. Im Folgenden soll noch kurz ein Überblick verschafft werden, welche Funktionen die einzelnen Komponenten erbringen. Eine detaillierter Beschreibung der Komponenten wird im weiteren Verlauf des Kapitels gegeben.

Die Verwaltung der Phasen wird durch die Methoden der Klasse *FragmentProvider* übernommen. Die Klassen *BaseAgent* und *BaseAgentMobility* realisieren, unter Ausnutzung der *Agent* Facette von Voyager, die Mobilität der Fragmente. Die zentrale Einheit des mobilen Fragments bildet ein Objekt der Klasse *FragmentAgent*. Diese Klasse ist Unterklasse von *BaseAgent* und verbindet das mobile Fragment mit dem stationären Teil des Fragmentierten Agenten. Den transparenten Austausch von Phasen, sowie eine automatische An- und Abmeldung bei einem Kommunikationspartner, ermöglichen die Methoden der Klassen *FragmentInterface* und *FragmentExchangeLock*. Jeder Anwendungsfall eines Phasenbasierten Agenten wird in der Regel aus unterschiedlichen Phasen bestehen. Die Klasse *Phase* liefert eine Vorlage, von der alle kon-

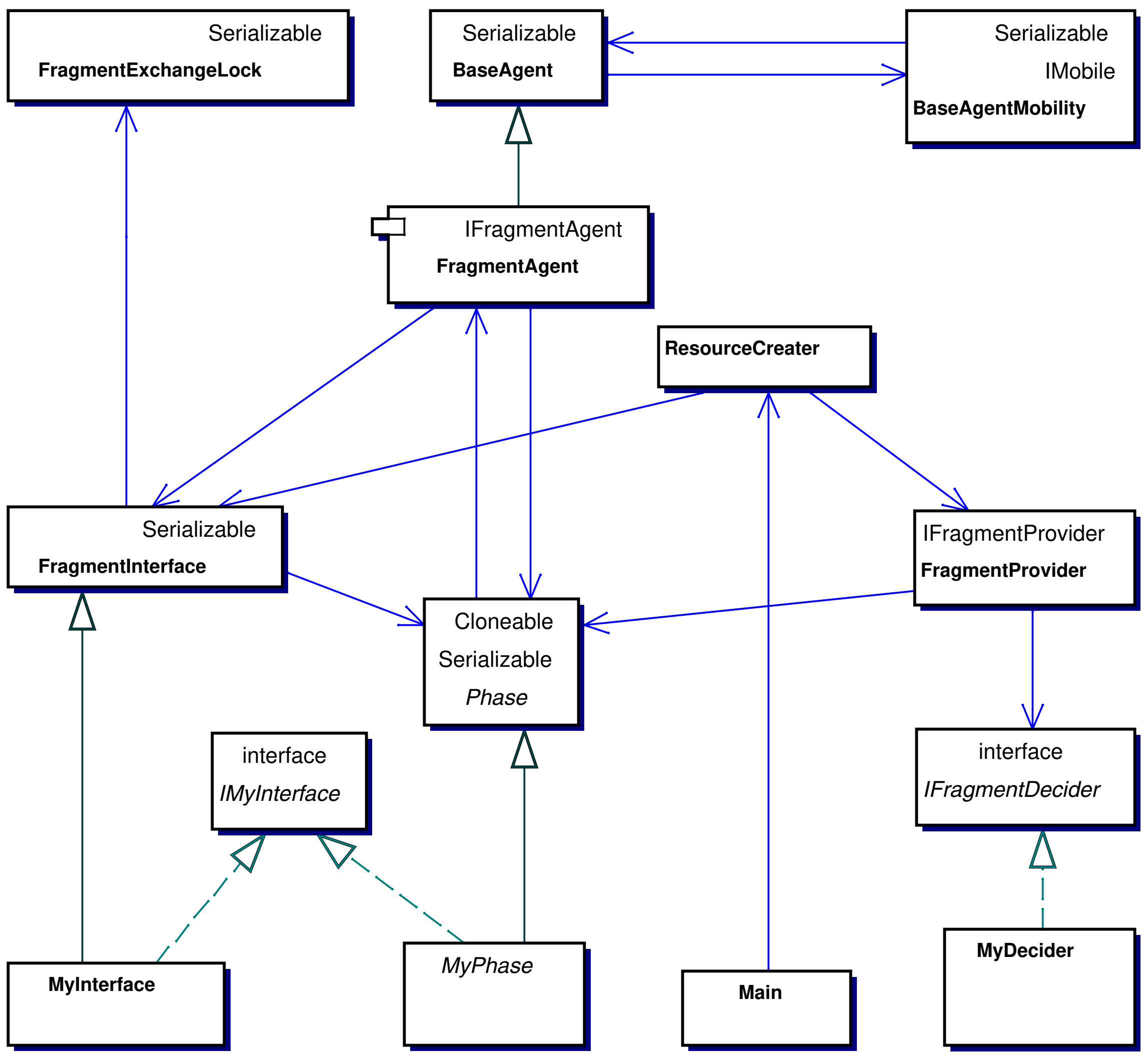

Abb. 3.1 Klassendiagramm des Frameworks

kreten Implementationen einer Phase abgeleitet werden müssen. Diese Vorlage verfügt unter anderem über Methoden, mit denen die Endergebnisse einer Phase an die Workflowengine weitergeleitet werden. Da das Konzept des Phasenbasierten Agenten im Wesentlichen keine Anforderungen an die Art und den Aufbau der Workflowengine stellt, ist die Workflowengine sehr lose mit dem Framework gekoppelt. Ein Objekt muss lediglich das Interface *IDecider* implementieren, um als Workflowengine zu fungieren. Die Schnittstelle zum Kommunikationspartner eines Fragmentes ist anwendungsspezifisch und wird durch die Ableitung der Klasse *FragmentInterface* und der Implementierung eines anwendungsbezogenen Interfaces erzeugt. Ein Anwendungsfall des Frameworks wird durch die Methode *main()* der Klasse *Main* gestartet. Diese Methode startet ihrerseits eine Voyager-Laufzeitumgebung und erzeugt ein Objekt der Klasse *ResourceCreater*. Die Aufgabe dieser Klasse ist es, alle wichtigen Objekte des Frameworks und des Anwendungsfalles zu erzeugen.

3.1.2 FragmentProvider

Ein Phasenbasierter Agent besteht mindestens aus zwei Teilen, einem mobilen Fragment und einer stationären Komponente. Der stationäre Anteil des Phasenbasierten Agenten setzt sich aus einem Objekt der Klasse *FragmentProvider*, einer Workflowengine und einer beliebigen Anzahl von Phasen zusammen. Die Aufgabe der Instanz der Klasse *FragmentProvider* ist es das mobile Fragment mit der Workflowengine zu verbinden und die Phasen zu verwalten.

Die Verwaltung der Phasen ist eingeschränkt auf die für das Konzept nötigen Funktionalitäten. Alle Phasen eines Anwendungsfalles werden zu Beginn der Ausführung eines Phasenbasierten Agenten mit der Methode *put()* an das Objekt der Klasse Fragmentprovider übergeben. Benötigt die Workflowengine den Zugriff auf verschiedene Phasen, beispielsweise um sie zu verändern, so ist dies über den Aufruf der Methode *updatePhases()* möglich. Der Rückgabewert dieser Methode ist ein Objekt vom Typ *Enumeration* und enthält die Referenzen auf alle Phasen des Anwendungsfalles.

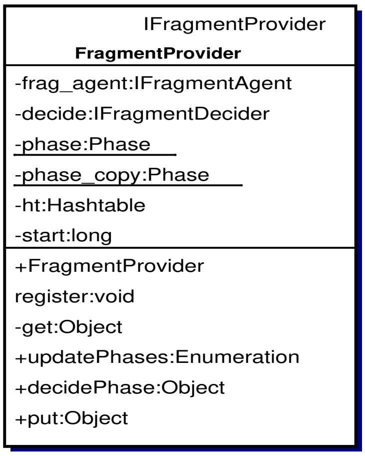

Abb. 3.2 UML-Darstellung der Klasse FragmentProvider

Um als Vermittler zwischen mobilen Fragment und Workflowengine zu dienen, implementiert die Klasse *FragmentProvider* das Interface *IFragmentProvider* mit der Methode *decidePhase()*. Dieses Interface ermöglicht es, dem mobilen Fragment über einen Proxy die Methode an einem entfernten Ort aufzurufen. Zweck dieser indirekten Verbindung ist der Austausch der Phase des mobilen Fragments. Mit den Parametern der Methode *decidePhase()* werden die Ergebnisse der aktiven Phase des mobilen Fragmentes über das Objekt der Klasse *FragmentProvider* an die Workflowengine übertragen (Abb. 3.4). Die Workflowengine wertet die Ergebnisse aus und entscheidet mit welcher Phase das mobile Fragment seine Verarbeitung fortsetzen soll. Er liefert

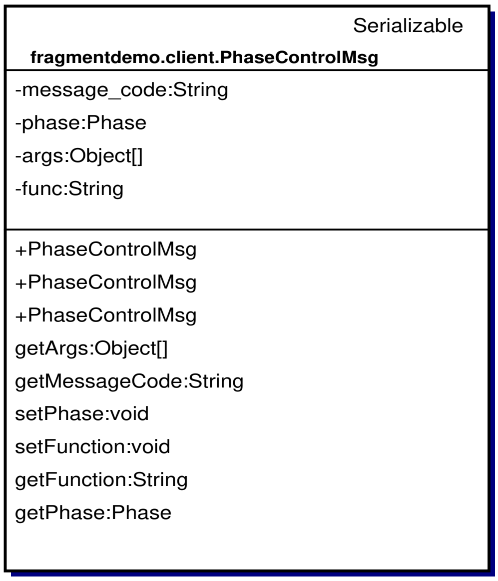

Abb. 3.3 UML-Darstellung der Klasse PhaseControlMsg

den Namen der Phase als Rückgabewert gekapselt in einem Objekt vom Typ *PhaseControlMsg* (Abb. 3.3) an die Instanz der Klasse *FragmentProvider*. Diese fragt den Namen der Phase ab und setzt eine Kopie der Phase in das Objekt der Klasse *PhaseControlMsg* ein, bevor es an das mobile Fragment als Rückgabewert geliefert wird.

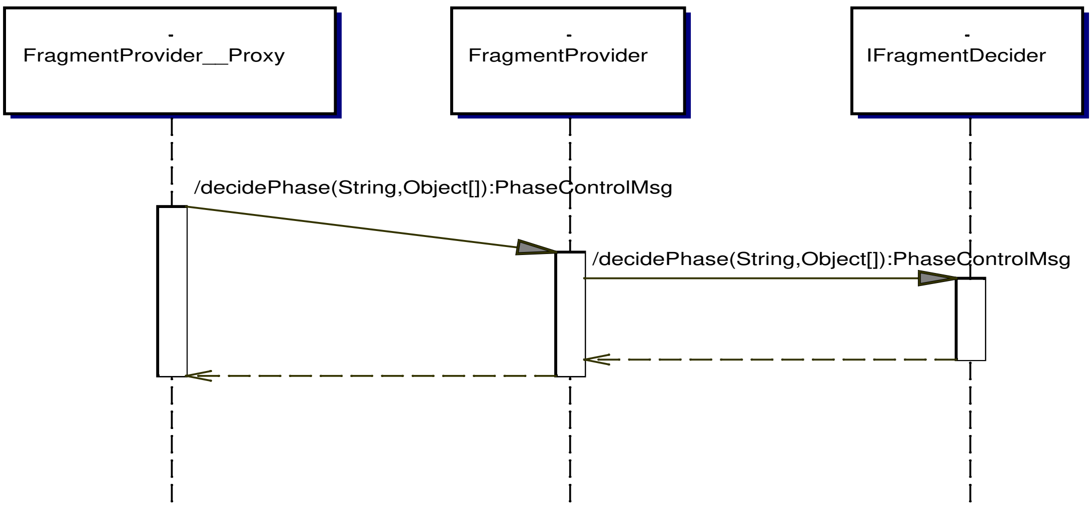

Abb. 3.4 Interaktionsdiagramm eines Phasenaustausches

Im Gegensatz zu einer direkten Verbindung zwischen mobilen Fragment und Workflowengine bietet die vorgestellte Strukturierung den Vorteil, dass sie die Verwaltung und den Transport der Phasen in das Framework verlagert. Der Workflowengine muss lediglich entscheiden, mit welcher Phase und Methode das mobile Fragment seine Verarbeitung fortsetzen soll und braucht deshalb nur lose mit dem Framework verbunden werden.

3.1.3 Mobilität der Fragmente

Eine elementare Grundeigenschaft eines Phasenbasierten Agenten ist die Mobilität seiner Fragmente. Mobilität im Sinn einer einfachen Objektmigration ist jedoch nicht ausreichend, denn ein mobiles Fragment soll seine Verarbeitung nach einem Transfer zu einem anderen System autonom fortsetzen können wie ein mobiler Agent. Voyager verfügt mit der in Abschnitt 2.4.6 vorgestellen *Agent* Facette über die Möglichkeit ein Objekt um die Eigenschaften eines mobilen Agenten zu erweitern. Die Facette erfüllt damit alle Anforderungen die an die Migration eines mobilen Fragments gestellt werden und kann zur Realisierung der Mobilität von Fragmenten verwendet werden.

Ein mobiles Fragment besteht, wie in Abschnitt 2.2.1 des Grundlagenkapitel beschrieben, nicht aus einem einzigen Bestandteil, sondern setzt sich aus einem Fragmentinterface und einer Fragmentimplementation zusammen. Bei dem Entwurf eines Frameworks für Phasenbasierte Agenten stellt sich die Frage, welches der beiden Elemente die Mobilität des Fragments verwirklichen soll. Im ersten Ansatz wird man die Fragmentimplementation durch die *Agent* Facette erweitern, weil sie der eigentliche Funktionsträger eines Phasenbasierten Agenten ist, und entscheidet wann und wohin eine Migration stattfindet. Eine Phase kann also direkt über ihre *Agent* Facette die Migration zu einem anderen System einleiten. Obwohl dieser Ansatz naheliegend ist, weist er einen entscheidenden Nachteile auf:

Im Verlauf der Erfüllung einer Aufgabe wird die Fragmentimplementierung eines mobilen Fragmentes mehrmals gewechselt. Die ausgetauschten Phasen werden nicht mehr gebraucht und können von der Garbage Collection gelöscht werden. Eine *Agent* Facette ist eine komplexe Komponente. Sie besteht aus mehreren Objekten und Proxies, die Voyager zusammen mit der Referenz auf das primäre Objekt erzeugt und verwaltet. Die Eigenschaften jeder Phase, durch eine *Agent* Facette zu erweitern, verursachen einen unnötig hohen Aufwand, da nicht die einzelnen Phasen die Eigenschaften und Funktionalitäten eines mobilen Agenten benötigen, sondern nur der Verbund aus Fragmentinterface und Phase. Leider erlaubt es die Dynamic Aggregation auch nicht bereits erzeugte Facetten an andere Objekte zu übertragen.

Die Erweiterung des Fragmentinterfaces durch eine *Agent* Facette ist der zweite Ansatz. Das Fragmentinterface bleibt während der gesamten Ausführungszeit eines Phasenbasierten Agenten unverändert. Die Mobilität und Autonomie eines Fragments kann also mit einer einzigen *Agent* Facette ermöglicht werden. Schwachpunkt dieser Idee ist die Funktionsweise der Facette. Die Migration des Fragments kann zwar von der aktiven Phase des Fragments angestossen werden, aber die Methode, mit der am Zielort die Ausführung fortgesetzt wird, muss vom Fragmentinterface sein, dem Objekt das durch die Facette erweitert wird. Sicherlich ist es kein Problem nach einer Migration Methodenaufrufe des Fragmentinterfaces an die Phase weiterzuleiten, doch sind verschiedene Nachteile damit verbunden. Alle Methoden die nach einer Migration aufgerufen werden sollen, müssen öffentlich sein. Ein Kommunikationspartner erhält über das Fragmentinterface somit Zugriff auf Methoden, die nicht zur normalen Schnittstelle des Fragments gehören, sondern fragmentintern sein sollten. Sieht man von diesem Sicherheitsrisiko ab, schränkt der Ansatz auch die Flexibilität ein. Alle Methoden mit denen nach einem Transfer die Ausführung wieder aufgenommen werden soll, müssen durch jede Phase implementiert und vom Fragmentinterface aufgerufen werden. Soll eine Phase am Zielort seine Ausführung mit einer speziellen Methode fortsetzen, müssen alle Phasen und das Fragmentinterface erweitert werden.

Die Mängel der einzelnen Lösungsansätze sind zu groß, um sie direkt einsetzen zu können. Ziel ist die Kombination der Vorteile beider Ansätze. Eine Phase muss mit einer beliebigen öffentlichen Methode am Zielort seine Ausführung fortsetzen können, ohne dass eine Anpassung anderer Objekte erforderlich ist. Damit nicht unnötig Ressourcen verbraucht werden, darf ein mobiles Fragment nur über eine einzige *Agent* Facette verfügen. Erster Schritt zum entgültigen Ansatz ist die Verlagerung der Mobilität in eine dritte Komponente, die aus den Klassen *BaseAgent* und *BaseAgentMobility* besteht. Das Objekt der Klasse *BaseAgentMobility* erweitert seine Funktion direkt bei der Erzeugung durch eine *Agent* Facette und implementiert das Interface *IMobile*. Es kapselt alle Voyager abhängigen Details und referenziert das Objekt der Klasse *BaseAgent*. Der Grund für ein Verlagerung der voyagerspezifischen Funktionalitäten in ein gesondertes Objekt ist vor allem die Entwicklung eines von der Agentenplattform möglichst unabhängigen Frameworks.

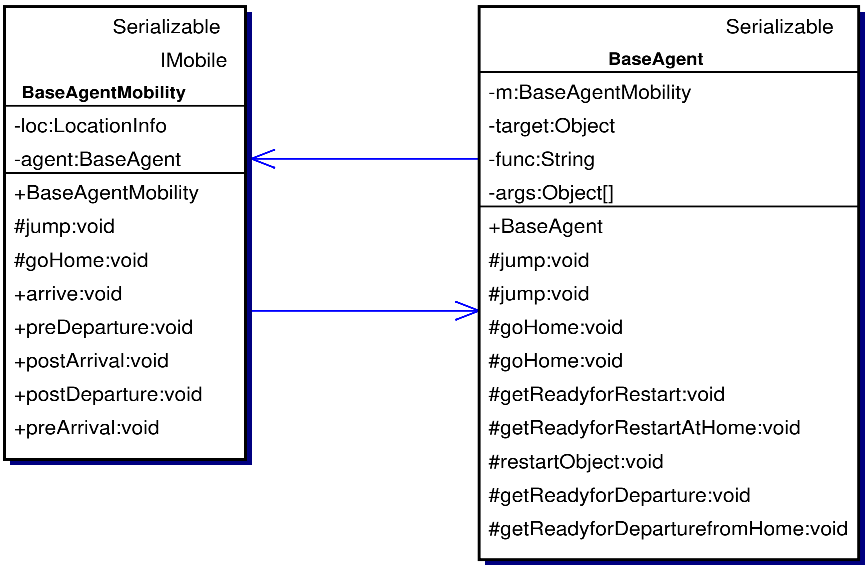

Abb. 3.5 UML-Darstellung der Klassen BaseAgent und BaseAgentMobility

Die Klasse *BaseAgent* verfügt über eine Methode *jump()* ähnlich der *moveTo()* Methode der *Agent* Facette. Im Unterschied zu *moveTo()* wird bei *jump()* zusätzlich zu einem Methodennamen und optionalen Parametern ein Objekt des Fragments übergeben, mit dem am Zielort die

Ausführung wieder aufgenommen werden soll. Damit die Wiederaufnahme der Verarbeitung mit einer öffentlichen Methode eines beliebigen Objekts möglich ist, müssen verschiedene Schritte vor und nach einer Migration ausgeführt werden.

Vor einer Migration werden alle Parameter des Methodenaufrufes *jump()* gespeichert. Anschließend leitet das Objekt der Klasse *BaseAgentMobility*, durch den Aufruf der Methode *moveTo()* mit dem Zielort und dem Namen der eigenen Methode *arrive()* als Parameter, die eigentliche Migration ein. Sobald das mobile Fragment zum Zielort vollständig übertragen ist, ruft die Voyager-Laufzeitumgebung die Methode arrive() des Objektes der Klasse *BaseAgentMobility* auf und startet damit das mobile Fragment am Zielort. Diese Methode ruft nun die Methode *restartObject()* der Klasse *BaseAgent* auf. Mit Hilfe des Reflexion-API der Java Standardklassenbibliothek startet diese Methode das vor der Migration gespeicherte Objekt. Ein direkter Aufruf des Objektes ist nicht möglich, da der Typ des Objektes und die Methode zur Übersetzungszeit nicht feststehen. Es muss also dynamisch aus dem Namen der Methode und den restlichen vor der Migration gespeicherten Parametern ein Objekt der Klasse *Methode* des Reflexion-API erzeugt werden [Fla98]. Dieses Objekt verfügt über eine Methode *invoke()* die den Aufruf des Objektes mit den gespeicherten Parametern ermöglicht.

Das Objekt der Klasse *BaseAgentMobility* ruft im Verlauf der Migration zwei Methoden an der Instanz der Klasse *BaseAgent* auf. Die erste Methode *getReadyforDeparture()* informiert über die unmittelbar bevorstehende Migration. Die zweite *getReadyforRestart()* signalisiert nach Ankunft am Zielort, dass die Ausführung wieder aufgenommen wird. Diese Methoden werden für verschiedene Abläufe, die mit der Migration eines Fragmentes verbunden sind, benötigt.

3.1.4 FragmentAgent

Der Verbund aus *BaseAgent* und *BaseAgentMobility* stellt die nötigen Mechanismen für die Mobilität eines Fragments zur Verfügung, eine Verbindung zum Fragmentinterface und der aktiven Phase stellt er jedoch nicht her. Diese Aufgabe übernimmt die Klasse *FragmentAgent*. Sie ist Unterklasse von *BaseAgent* und bildet den Vermittler zwischen den Komponenten eines mobilen Fragments und dem stationären Anteil eines Phasenbasierten Agenten.

Entscheidet eine Phase, dass ihr Endzustand erreicht ist, leitet die Instanz der Klasse *FragmentAgent* die Endergebinsse an die stationäre Komponente des Phasenbasierten Agenten weiter. Die Rückantwort der Instanz der Klasse *FragmentProvider* enthält die neue Phase und weitere Informationen mit welcher Methode die Phase aufgerufen werden soll. Die neue Phase wird durch das Objekt der Klasse *FragmentAgent* mit dem Fragmentinterface und dem Kommunikationspartner des mobilen Fragments verbunden und anschließend aufgerufen. Die Klasse *Frag-*

mentAgent überlagert die Methoden *getReadyforDeparture()* und *getReadyforRestart()* seiner Oberklasse, um das Fragmentinterface über den Migrationsprozeß zu informieren und entsprechende Maßnahmen zu treffen.

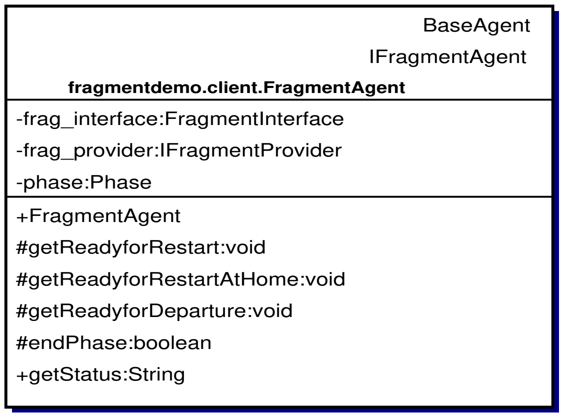

Abb. 3.6 UML-Darstellung der Klasse FragmentAgent

3.1.5 Fragmentinterface

Ein Fragmentinterface ist die Schnittstelle eines mobilen Fragments. Alle Methodenaufrufe an dieser Schnittstelle werden an die aktive Phase des mobilen Fragments weitergeleitet. Das Fragmentinterface nimmt somit die Rolle eines Stellvertreters ein, der eine stabile und immer gültige Referenz für den Kommunikationspartner bietet und den Zugriff auf die Phase kontrolliert. Zweck der Kontrolle ist es einen für den Kommunikationspartner des Fragments transparenten Austausch der Phasen zu ermöglichen.

Die Schnittstelle des Fragmentinterface ist für jeden Anwendungsfall eines Phasenbasierten Agenten unterschiedlich. Das Framework kann also nur eine Basisklasse für das Fragmentinterface bereitstellen, die es mit den anderen Komponenten des mobilen Fragments verbindet und über Mechanismen zum Austausch der Phase verfügt. Eine weitere Aufgabe, die das Fragmentinterface außerdem übernehmen sollte, ist die An- und Abmeldung des Fragments bei seinen Kommunikationspartnern.

Kern der Problematik eines transparenten Phasenaustausches ist die Existenz mehrere Aktivitätsträger, die parallel, direkt oder indirekt, auf die aktive Phase des Fragments zugreifen können. Indirekte Zugriffe erfolgen dabei über das Fragmentinterface durch einen Kommunikationspartner des Fragments, während direkte durch die Phase selbst erfolgen. Sicherlich ist es vorstellbar das beide Arten von Zugriffen einen Phasenaustausch einleiten können, doch ist die Koordinierung der Aktivitätsträger in diesem Fall äußerst schwierig. Es wurde deshalb festgelegt, dass ein mobiles Fragment nur über einen Aktivitätsträger verfügt. Dieser ist in der Lage einen Phasenaustausch einzuleiten.

Hat die aktive Phase eines Fragments ihren Endzustand erreicht und wird ausgetauscht, muss das Fragmentinterface alle Methodenaufrufe des Kommunikationspartners verzögern, bis sie an die neu eingesetzte Phase weitergeleitet werden können. Würde man auf die Blockierung der Methodenaufrufe verzichten, könnte es sein, dass Anfragen an die bisher aktive Phase gestellt werden, die sie nicht mehr bearbeiten kann.

Die Programmiersprache Java verfügt mit dem Schlüssewort *synchronized* über einen Monitormechanismus der einen kritischen Bereich vor nebenläufigen Zugriffen schützt. Kennzeichnet man die Methoden des Fragmentinterface mit diesem Schlüsselwort, so kann der Kommunikationspartner immer nur genau eine Methode des Fragmentinterfaces ausführen. Alle Methodenaufrufe durch andere Aktivitätsträger des Kommunikationspartners werden blockiert bis dieser Methodenaufruf beendet ist. Erreicht die Phase ihren Endzustand und der Aktivitätsträger des mobilen Fragments leitet den Phasenaustausch ein, muss das Fragmentinterface mit einer expliziten *synchronized*-Anweisung für Zugriffe durch den Kommunikationspartner gesperrt werden. Dies ist nötig da der Aktivitätsträger nicht über das Fragmentinterface die Methoden der Phase aufruft und somit schon einen exklusiven Zugriff auf die Phase hat, sondern direkt Methoden der Phase aufrufen kann. Ist der Phasenaustausch vollzogen wird die Sperrung aufgehoben. Alle folgenden Methodenaufrufe werden dann vom Fragmentinterface an die neu Phase weitergeleitet. Der Austausch der Phase ist somit für den Kommunikationspartner transparent.

Problematisch an dieser Verfahrensweise ist die Möglichkeit, das zum Zeitpunkt der *synchronized*-Anweisung das Fragmentinterface schon durch einen Methodenaufruf blockiert sein kann. Die Anweisung wird in einem solchen Fall in die interne Warteschlange des Objektes eingeordnet bis die Methode das Fragmentinterface frei gibt. Aber selbst danach ist nicht gesichert, dass sie den Zugriff auf das Fragmentinterface bekommt, wenn eine Reihe von anderen Aktivitätsträgen bereits auf den Zugriff warten. Die Konsequenz ist, dass nach Erreichen der Endbedingung einer Phase ein Kommunikationspartner noch Methoden der Phase aufrufen kann. Diese Methoden sind unter Umständen nicht vollständig implementiert, da sie im Aufgabenbereich der folgenden Phase liegen. Die Aufrufe solcher Methoden erfolgen dann zwar in der richtigen logischen Reihenfolge, aber zum falschen Zeitpunkt, da das mobile Fragment noch nicht bereit für die Fortsetzung der Verarbeitung mit diesen Methoden ist.

Einen einfachen Ausweg bietet die Möglichkeit, auch dem Aktivitätsträger des Fragments nur einen indirekten, synchronisierten Zugriff über das Fragmentinterface auf die Phase zu gestatten. Leider sind damit ähnliche Nachteile verbunden wie in Abschnitt 3.1.3 bereits beschrieben. Jede Methode einer Phase, die einen Phasenaustausch einleiten kann, muss sowohl von jeder anderen Phase als auch vom Fragmentinterface implementiert werden. Dies ist unter anderem nötig damit das Fragmentinterface jede Phase direkt referenzieren kann. Aus der Sicht eines Anwendungsprogrammierers bedeutet dieser Ansatz also eine Reihe von Restriktionen, die er beachten muss. Mit diesem Verfahren der vollständigen Synchronisierung der Zugriffe auf die Phase eines mobilen Fragments wird leider auch die Ausführungsgeschwindigkeit der Kommunikation zwischen mobilen Fragment und seinem Partner reduziert. Es wird deshalb ein weiterer Ansatz betrachtet:

Grundlage des Ansatzes ist ein Read/Writer-Muster [Lea 98]. Dieses Muster teilt die Zugriffe auf eine Ressource in zwei Gruppen ein: Modifizierende Operationen, die eine Veränderung an der Ressource vornehmen und nicht modifizierende bzw. lesende Operationen. Modifizierende Operation erhalten immer exklusiven Zugriff auf die Ressource. Konkurrierende Operationen die Inkonsistenzen auslösen können, werden mit dieser Maßnahme ausgeschlossen. Die parallele Ausführung mehrerer lesender Operationen wird jedoch zugelassen, da sie sich gegenseitig nicht negativ beeinflussen können.

Im Falle eines mobilen Fragments werden alle Methodenaufrufe des Kommunikationspartners am Fragmentinterface als Leseoperationen betrachtet. So ist es möglich das die Aktivitätsträger des Kommunikationspartner mehrere Methoden des Fragmentinterfaces gleichzeitig aufrufen können. Ein Phasenaustausch wird als modifizierende Operationen eingestuft, die nur von dem Aktivitätsträger der Phase durchgeführt werden kann.

Anders als im ersten Ansatz bei dem alle Zugriffe gleichwertig sind, haben in diesem Fall modifizierende Operationen eine höhere Priorität als lesende. Befinden sich eine oder mehrere lesende Operationen in der Verarbeitung und ein modifizierender Aufruf erfolgt, so werden die lesenden Operationen noch normal beendet, aber keine neuen lesenden Operationen mehr zugelassen. Im Gegensatz zu der einfacheren Variante eines synchronisierten Fragmentinterfaces kann also nicht eine Reihe von beliebigen neuen Methodenaufrufen an der Phase erfolgen, wenn sie ihren Endzustand erreicht hat.

Bisher wurden die allgemeinen Mechanismen erläutert wie ein transparenter Phasenaustausch ermöglicht werden kann. Nun sollen kurz die einzelnen Klassen und einige zusätzliche Aufgaben die das Fragmentinterface übernimmt, vorgestellt werden.

Die Klasse *FragmentInterface* bildet eine Vorlage des Frameworks für das tatsächliche Fragmentinterface eines Phasenbasierten Agenten. Ein Anwendungsentwickler muss von dieser Klasse eine Unterklasse bilden und zusätzlich das Interface seines Fragments implementieren. Jede Operation die mit dem Read/Writer-Muster verwalten werden soll, muss unmittelbar vor ihrer Ausführung die Methode *beforeInterfaceAction()* der Klasse *FragmentInterface* aufrufen um den Zugriff auf die Phase zu erlangen. Nach Abschluss der Operation wird der Zugriff durch den Aufruf der Methode *afterInterfaceAction()* wieder abgegeben. Die Klasse *FragmentInterface* leitet diese Methodenaufrufe an eine Instanz der Klasse *FragmentExchangeLock* weiter, die das Read/Writer-Muster realisiert. Diese Trennung zwischen dem Fragmentinterface und der Klasse *FragmentExchangeLock* mit dem Koordinierungsmechanismus bildet, einerseits eine sinnvolle Verteilung der Aufgaben im Sinne einer objektorientierten Programmierung und erlaubt es andererseits den Koordinierungsmechanismus bei Bedarf mit geringem Aufwand auszutauschen.

Ein wichtiger Punkt, der bisher noch nicht erläutert wurde, ist wie die Verbindung zwischen einem mobilen Fragment und seinem Kommunikationspartner auf- und abgebaut wird. Im vorliegenden Entwurf wird von der Annahme ausgegangen, dass ein mobiles Fragment seinen Kommunikationspartner kennt. Trifft ein mobiles Fragment auf einem Zielsystem ein, so erhält das Fragmentinterface über den Namensdienst von Voyager eine Referenz auf den Kommunikationspartner.Damit der Kommunikationspartner nun über das mobile Fragment informiert wird, ruft das Fragmetinterface die Methode *register()* des Kommunikationspartners auf. Im Unterschied zu einem fragmentierten Objekt, kann eine Referenz auf ein Fragmentinterface eines Phasenbasierten Agenten seine Gültigkeit verlieren, da das mobile Fragment zu einem anderen Ort migriert. In einem solchen Fall muss das mobile Fragment dem Kommunikationspartner mitteilen, dass seine Referenz ungültig wird. Dies geschieht mit der Methode *unregister()* des Fragmentinerfaces sobald ein Ortwechsel des Fragments eingeleitet wird.

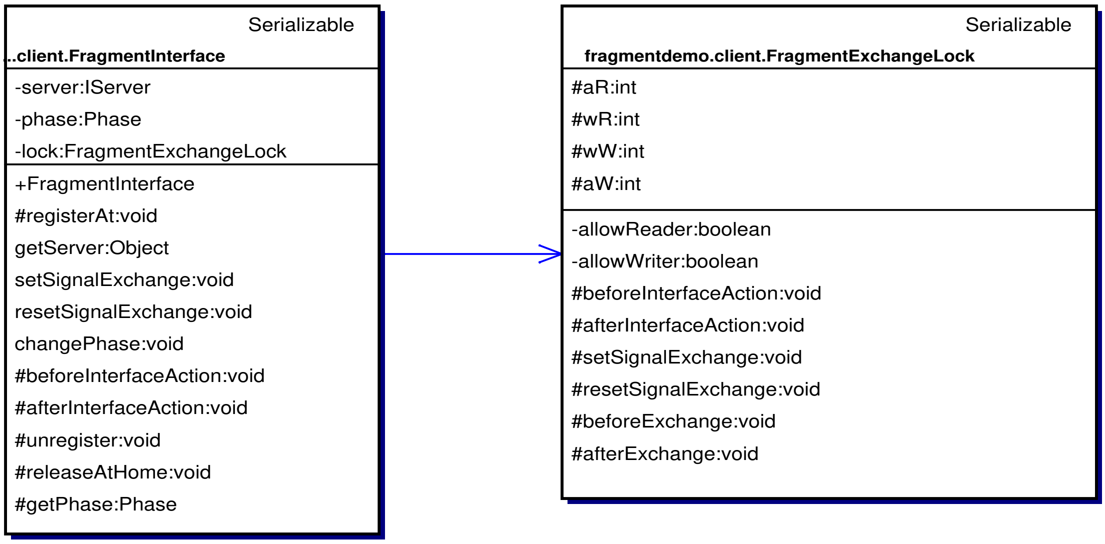

Abb. 3.7 Klassendiagramm des Fragmentinterface

3.1.6 Phase

Innerhalb eines mobilen Fragments definiert eine Phase alle Methoden, die zur Erfüllung einer Teilaufgabe der Gesamtaufgabenstellung eines Phasenbasierten Agenten nötig sind. Da jede Phase anwendungsspezifisch ist, verfügt das Framework mit der Klasse *Phase* lediglich über eine Vorlage für konkrete Implementierungen.

Im Unterschied zu anderen Komponenten, die bei Beginn der Ausführung eines Phasenbasierten Agenten fest miteinander verbunden sind, wird eine Phase dynamisch nach Bedarf in ein mobiles Fragment eingefügt. Damit die Einbindung einer Phase in das mobile Fragment weitgehend transparent für den Anwendungsentwickler ist, stellt die Klasse *Phase* Methoden bereit, die automatisch bei einem Phasenwechsel an der neuen Phase aufgerufen werden, um es mit dem Fragment zu verbinden. Eine Phase benötigt aus zwei Gründen eine Verbindung zum mobilen Fragment. Einerseits um einen Ortswechsel des Fragments einzuleiten, andererseits um das Erreichen ihres Endzustandes an die Workflowengine zu signalisieren. Da die Zugriffe einer Phase auf die Komponenten des mobilen Fragments so eng begrenzt sind, bietet es sich an die Verbindung zum Framework in der Basisklasse *Phase* zu kapseln und nur indirekte Zugriff auf das mobile Fragment über Methodenaufrufe zu ermöglichen. Diese Maßnahme hat den Vorteil, das Phasen sehr einfach zu implementieren sind. Der Anwendungsentwickler muss sich keine Gedanken über die Bestandteile des mobilen Fragments machen, und braucht lediglich ein Methode des Oberklasse seiner Phase aufrufen, um die entsprechende Aktion auszuführen.

Die Migration des mobilen Fragments kann die Phase mit der Methode *jump()* auslösen. Als Parameter muss lediglich der Name der Methode übergeben werden mit der die Ausführung der Phase am Zielort fortgesetzt werden soll. Ähnlich verhält es sich bei der Methode *endPhase()*, mit der die Endergebnisse der Phase, in Form eines Objektvektors, an die Workflowengine übertragen werden. Die Klasse *Phase* leitet diese Aufrufe lediglich an das Objekt der Klasse *FragmentAgent* weiter. Andere Vorteile der Kapselung sind, dass bei einer Veränderung der Komponenten des Frameworks keine Veränderungen an den Phasen vorgenommen werden müssen und keine Zugriffe der Phasen auf interne Methoden des Frameworks erfolgen können.

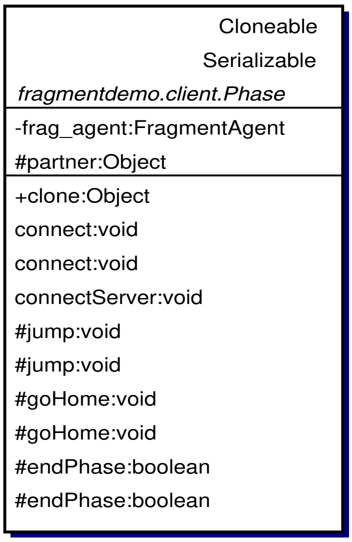

Abb. 3.8 UML-Darstellung der Klasse Phase

Die Phase eines mobilen Fragments ist aber normalerweise nicht nur mit den Komponenten des Fragments verbunden, sondern auch mit einem Kommunikationspartner. Bei dem Entwurf des Frameworks wurde von der vereinfachenden Annahme ausgegangen, dass es jeweils nur einen bestimmten Kommunikationspartner pro Aufenthaltsort gibt. Diese Annahme ist zum Beispiel in einem Verkaufsscenario, bei dem es einen Verkäufer, den Kommunikationspartner und einen Interessenten, das mobile Fragment gibt, vollkommen ausreichend. Da das Fragmentinterface bereits bei der Ankunft des Phasenbasierten Agenten sich bei dem Kommunikationspartner anmeldet und somit eine Referenz auf den Kommunikationspartner besitzt, macht es keinen Sinn, den Aufbau der Verbindung zum Kommunikationspartner der Phase zu überlassen. Die Klasse *Phase* verfügt stattdessen über eine Variable, deren Wert im Falle eines Phasenwechsels oder einer Migration von den Komponenten des mobilen Fragments mit einer Referenz auf den Kommunikationspartner neu gesetzt wird. Ein Anwendungsprogrammierer kann also davon ausgehen, dass, sobald eine Phase ihre Ausführung beginnt, egal ob nach einer Migration oder einem Phasenwechsel, eine Referenz auf den Kommunikationspartner vorhanden ist.

3.1.7 IFragmentDecider

Eine Workflowengine muss wie in Abschnitt 3.1.2 beschrieben, lediglich das Interface *IFragmentDecider* implementieren, um in das Framework eingebunden werden zu können. Das Interface deklariert die zwei Methoden, *connect()* und *decidePhase()*. Bei Beginn der Ausführung eines Phasenbasierten Agenten erhält die Workflowengine durch die Methode *connect()* eine Referenz auf das Objekt der Klasse *FragmentProvider*, mit deren Methoden sie den Zugriff auf die Phasen erhält. Vordringliche Zweck des Interfaces ist es jedoch die Übermittlung der Endergebnisse einer Phase an die Workflowengine zu ermöglichen, damit entschieden werden kann, mit welcher Phase das mobile Fragment seine Verarbeitung fortsetzt. Zu diesem Zweck deklariert das Interface die Methode *decidePhase()*, die als Parameter den Namen der letzten aktiven Phase und einen Vektor von Objekten erhält. Wie eine Workflowengine die Endergebnisse verarbeitet und entscheidet welche Phase als nächste ausgeführt wird, ist nicht festgelegt. Der Rückgabewert der Methode ist ein Objekt vom Typ *PhaseControlMsg*. Dieses Objekt enthält den vollständigen Namen der Phase und der Methode, sowie die zugehörigen Parameter, mit denen die neue Phase des mobilen Fragments gestartet werden soll.

3.1.8 Konfiguration des Frameworks

Nachdem alle Komponenten und Schnittstellen des Frameworks besprochen wurden, stellt sich die Frage, wie das Framework mit den einzelnen Klassen der Anwendungsfälle verbunden wird. Es sollte ein sehr einfacher und flexibler Mechanismus sein, der keine erneute Übersetzung der Anwendung erfordert, falls eine oder mehrere Klassen, die eine Schnittstelle des Frameworks implementieren, ausgetauscht werden.

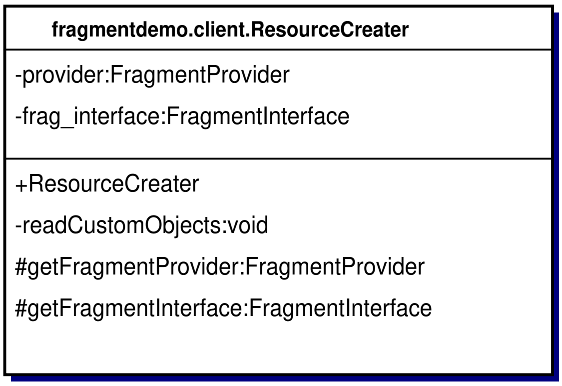

Abb. 3.9 UML-Darstellung der Klasse ResourceCreater

Eine Java-Laufzeitumgebung verfügt zur einfachen Konfiguration der virtuellen Maschine über den Mechanismus der Java System Properties. Eine Property besteht aus einem Schlüssel der mit einem Wert verbunden ist. Der Wert eines Schlüssels kann zu jeder Zeit abgefragt oder verändert werden. Außerdem können auch sehr einfach neue Properties eingeführt werden, beispielsweise indem man sie bei Beginn der Ausführung einer Anwendung auf der Befehlszeile übergibt.

Für die Konfiguration des Frameworks wird dieser Mechanismus der System Properties genutzt, um Informationen über die anwendungsspezifischen Klassen an das Framework weiterzugeben. Alle Namen von anwendungsspezifischen Klassen werden, verbunden mit festgelegten Schlüsseln, bei Beginn der Ausführung an die Java-Laufzeitumgebung übergeben und sind dann von jeder Komponente des Frameworks global zugänglich. Speziell werden diese Informationen aber durch die Instanz der Klasse *ResourceCreater* (Abb. 3.9) genutzt, da dieses Objekt alle wichtigen anwendungsspezifischen Objekte erzeugt.

3.2 Anwendungsfälle für Messungen

Dieses Kapitel beschreibt den Entwurf mehrerer Anwendungsfälle für Messzwecke unter Verwendung des bereits vorgestellten Frameworks. Zielsetzung der Messungen ist es, das Konzept des monolithischen Agenten, mit dem Ansatz des Phasenbasierten Agenten in Bezug auf Ausführungsgeschwindigkeit und Datenübertragung zu vergleichen. Das Konzept des Phasenbasierten Agenten ist vor allem für komplexere Aufgabenstellung entwickelt worden, deshalb sollen die Messungen außerdem ein Abschätzung ermöglichen, ab welcher Problemgröße der Einsatz von Phasenbasierten Agenten sinnvoll ist.

3.2.1 Bestimmung der Messgrößen

Bevor auf die einzelnen Agenten und ihren Aufbau eingegangen wird, muss geklärt werden, welche Messgrößen es gibt und wie die Messdaten ermittelt werden.

Der Vergleich der Ausführungsgeschwindigkeit beider Konzepte kann auf einfache Weise durch Zeitnahmen erfolgen. Es muss dafür nur festgelegt werden zu welchem Zeitpunkt der Ausführung eine Messung beginnt und wann ihr Ende erreicht ist. Der Ablauf eines monolithischen Agenten kann in drei Schritte eingeteilt werden:

(1) Zuerst migriert der mobile Agent zu einem entfernten Rechner.

(2) Anschließend tritt er vor Ort mit dem Zielsystem oder anderen Agenten in Interaktion.

(3) Hat der mobile Agent seine Aufgabe erfüllt, kehrt er abschließend zu seinem Ursprungsort zurück.

Der Ablauf des Phasenbasierten Agenten unterscheidet sich von dem eines monolithischen Agenten nur insofern, das die strikte Trennung zwischen Migration und Interaktion aufgehoben wird. Es findet bei Beginn der Ausführung des Phasenbasierten Agenten nur eine Teilmigration statt. Im Verlauf der Ausführung des Agenten werden je nach Bedarf weitere Phasen nachgeladen.

Für die Messungen wurde von der Annahme ausgegangen, dass sich die Art und Weise wie eine Aufgabenstellung gelöst wird, unabhängig von dem verwendeten Konzept ist. Lediglich die Aufteilung der Verantwortlichkeiten auf einzelne Komponenten kann variieren. Die Ausführungsgeschwindigkeit eines monolithischen Agenten vor Ort unterscheidet sich deshalb nicht von der einer aktiven Phase eines Phasenbasierten Agenten. Aus diesem Grund wird auf eine Implementierung der Interaktionen am Zielsystem verzichtet, da sie für den Vergleich der Konzepte nicht relevant ist. Wird der Lösungsweg von dem eingesetzten Konzept nicht beeinflusst, so sind auch die erzielten Endergebnisse identisch und müssen ebenfalls nicht berücksichtigt werden. In welcher Form der Agent zu seinem Ursprungsort zurückkehrt, wird durch keines der beiden Konzepte festgelegt und kann stark variieren.

Auf Grund dieser Annahmen kann die Zeitnahme auf die Migration des Agenten zum Zielort beschränkt werden, da alle anderen Abschnitte der Ausführung eines Agenten entweder vergleichbar sind oder nicht durch die Konzepte festgelegt werden. Eine Messung beginnt unmittelbar bevor der Agent seine Migration ausführt und endet wenn sie vollständig abgeschlossen ist. Die Migration eines monolithischen Agenten ist vollständig abgeschlossen, sobald er seine Ausführung am Zielort fortsetzt. Ein Phasenbasierter Agent kann seine Ausführung aber schon nach der Migration einer Phase fortsetzen. Ob die Migration aber vollständig abgeschlossen ist hängt von dem Verlauf der Ausführung des Agenten ab. Erfordert es die Situation werden nach Bedarf weitere Phasen migriert. Für die Messungen werden alle möglichen Fälle betrachtet, d. h. es wird im Verlauf einer Messung jede Phase zum Zielsystem übertragen und mit Beginn ihrer Ausführung auf diesem System erfolgt eine Zeitnahme. Dieses Verfahren ermöglicht einen direkten Vergleich der Transferzeit zwischen dem Konzept des monolithischen Agenten und dem des Phasenbasierten Agenten bei verschiedenen Situationen.

Die Zeitnahme für die Messungen erfolgt auf dem Ursprungsrechner der Agenten, um nicht die unterschiedlichen Systemzeiten verschiedener Rechner abgleichen zu müssen. Sobald ein monolithischer Agent seine Ausführung beginnt oder eine Phase eingesetzt ist, wird eine Nachricht an den Ursprungsort des Agenten gesendet und die Zeitnahme erfolgt. Da ein Phasenbasierter Agent aus einer ganzen Reihe von Phasen bestehen kann, werden für eine Messung der einzelnen Transferzeiten des Phasenbasierten Agenten je nach Anzahl der Phasen mehr Nachrichten übertragen. Diese Nachrichten beeinflussen die Messungen nicht, da jede Phase bei erreichen ihres Endzustandes ihre Endergebnisse an die Workflowengine übertragen muss. Im Rahmen der Messungen wird diese Übertragung der Endergebnisse als Nachricht zur Zeitnahme genutzt, es müssen also keine zusätzlichen Nachrichten außer der für das Konzept des Phasenbasierten Agenten nötigen Nachrichten übertragen werden. Sicherlich kann die Laufzeit der Nachricht je nach Übertragungsrate des Mediums eine gewisse Dauer beanspruchen aber diese Laufzeit ist für beide Konzepte gleich und führt deshalb keine Verzerrung zwischen den Messergebnissen der beiden Konzepte herbei.

Das Konzept des Phasenbasierten Agenten ist vor allem für komplexe Aufgabenstellungen entwickelt worden, unter der Annahme das oft nur ein Teil der Funktionen des Agenten benötigt werden. Trifft diese Annahme zu, kann man davon ausgehen das ein Phasenbasierter Agent eine Aufgabe meist schneller erfüllen kann, als ein monolithischer Agent, da bei einer Migration eines Phasenbasierten Agenten weniger Programmcode und Daten übertragen werden müssen.

Um diese Behauptungen durch Messungen zu überprüfen, benötigt man sehr große und komplexe Agenten. Ein derzeit verwendetes Maß, um die Größe von mobilen Agenten bei Messungen anzugeben, ist die Anzahl ihrer Programmzeilen [DNM+99]. Der Nachteil dieser Maßeinheit ist, dass die Größe der zu übertragenden Daten damit nur sehr ungenau wiedergegeben werden. Faktoren wie der Inhalt der Programmzeilen und die Art der Entwicklungsumgebung können die tatsächliche Größe des Agenten stark beeinflussen. Es wurde deshalb eine andere Maßeinheit gewählt. Ein monolithischer Agent bzw. jede Phase eines Phasenbasierten Agenten verfügt über einen Bytevektor. Die Größe dieses Vektors kann sehr einfach verändert werden und gibt an wieviele Daten mindestens übertragen werden. Der absolute Betrag der übertragenen Daten wird mit diesem Verfahren jedoch nicht gemessen, da weder der Programmcode des Agenten noch sein sonstiger Zustand durch die Messgröße berücksichtigt werden. Bei der Verwendung dieser Messgröße muss deshalb darauf geachtet werden, dass der Programmcode des Agenten so gering wie möglich gehalten wird und die Größe und Struktur des Programmcodes für alle Messfälle ähnlich ist. Werden beide Bedingungen eingehalten, können Verzerrungen durch die nicht berücksichtigten Faktoren vernachlässigt werden.

3.2.2 Basisklassen der Anwendungsfälle

Als Voraussetzungen für aussagekräftige und vergleichbare Messungen müssen alle Anwendungsfälle eine sehr ähnliche Codebasis aufweisen, da sie nicht in die Größenangabe des Agenten eingeht, aber die Übertragungszeiten der Anwendungsfälle beeinflusst. Es wurde deshalb eine Reihe von Basisklassen entworfen, die als Grundlage aller Messfälle dient.

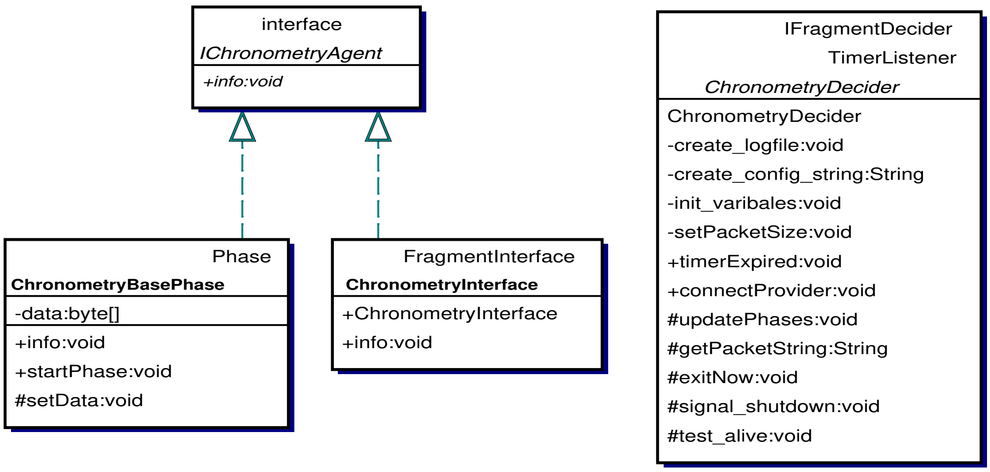

Abb. 3.10 Klassendiagramm der Basisklassen

Die Schnittstelle eines Agenten spielt im Rahmen der Messungen eine untergeordnete Rolle und wird bei allen Anwendungsfällen durch eine Instanz der Klasse *ChronometryInterface* repräsentiert. Diese Unterklasse von *FragmentInterface* implementiert das Interface *IChronometryInterface*, welches nur über eine einzige Methode für Demonstrationszwecke verfügt und bei einer echten Anwendung die Methoden der Schnittstelle des Agenten definieren sollte. Als

Grundlage aller Phasen der Anwendungsfälle erbt die Klasse *ChronometryBasePhase* von der Klasse *Phase* des Frameworks und implementiert ebenfalls das Interface *IChronometryInterface*. Diese Klasse gibt eine Startmethode vor, mit der alle Phasen aufgerufen werden. Die Kernfunktion der Klasse *ChronometryBasePhase* bildet jedoch eine Methode mit der man die Agentengröße, einen Bytevektor, festlegen kann. Die letzte Basisklasse bildet *ChronometryDecider*, eine Vorlage für die Workflowengines der verschiedenen Anwendungsfälle. Aufgabe dieser Vorlage ist es, Methoden für den Ablauf der Messungen und die Protokollierung der Messergebnisse zur Verfügung zu stellen.

3.2.3 Der Phasenbasierte Agent

Für die Messungen wurde, aufbauend auf den bereits vorgestellten Basisklassen, ein Phasenbasierter Agent entwickelt. Um möglichst genau bestimmen zu können unter welchen Umständen der Einsatz von Phasenbasierten Agenten Vorteile gegenüber monolithischen Agenten bietet, sollte der Ablauf des Agenten sich aus einer möglichst hohen Anzahl von Phasen zusammensetzen. Betrachtet man das in Abschnitt 2.3 des Grundlagenkapitels vorgestellte Beispiel eines Kaufagenten mit maximal vier aufeinanderfolgenden Phasen, so erscheint ein Phasenbasierter Agent dessen Ablauf aus zehn Phasen besteht als ausreichend.

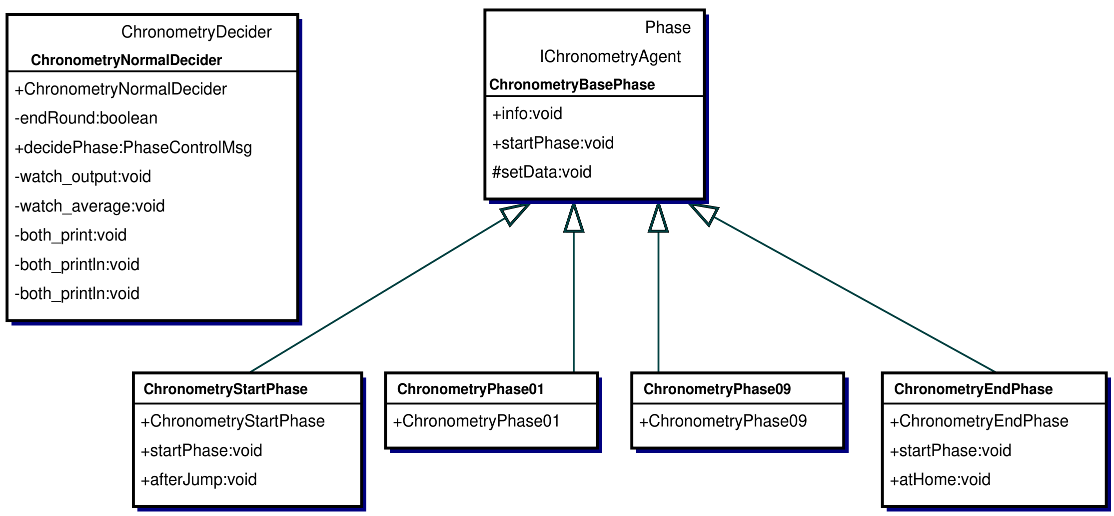

Abb. 3.11 Klassendiagramm des Phasenbasierten Agenten

Die erste Phase wird noch am Ursprungsort des Agenten in das mobile Fragment eingesetzt und ist ein Objekt der Klasse *ChronometryStartPhase*. Sobald diese Phase aktiviert wird, migriert das mobile Fragment zum Zielort. Nach der Ankunft am Zielort hat die Phase ihre Aufgabe erfüllt und signalisiert der Workflowengine ihren Endzustands. Im Folgenden werden dann neun verschiedene Phasen eingesetzt, die jeweils sofort nach Beginn ihrer Ausführung ihren Austausch einleiten. Es wurde für jede Phase eine eigene Unterklasse von *ChronometryBasePhase* gebildet, um sicher zustellen, dass mit jedem Phasenwechsel ein Codetransfer stattfindet. Die letzte in das mobile Fragment eingesetzte Phase bildet eine Instanz der Klasse *Chronometry-*

EndPhase. Diese Phase ist für die Messungen nicht relevant, da die Zeitnahme unmittelbar vor ihrer Übertragung zum mobilen Fragment abgeschlossen wird und sie lediglich die Rückkehr des mobilen Fragments zu seinem Ursprungsort koordiniert.

Die Workflowengine des Phasenbasierten Agenten wird repräsentiert durch eine Instanz der Klasse *ChronometryNormalDecider* und regelt nicht nur die bereits beschriebene Abfolge der Phasen, sondern protokolliert auch den Zeitpunkt ihres Austausches. Der Beginn der Messung ist unmittelbar bevor die erste Phase ihre Ausführung aufnimmt. Sobald eine Phase ihren Austausch eingeleitet hat, und am Ursprungsort die Methode *decidePhase()* der Workflowengine aufgerufen wird, erfolgt eine Zeitnahme. Das Ende einer Messung ist erreicht, sobald die letzte Phase eingesetzt wird, da nur die Migration zum Zielort gemessen werden soll. Ein Ablauf des Phasenbasierten Agenten liefert also zehn Messwerte, wie Abblidung 3.13 verdeutlicht.

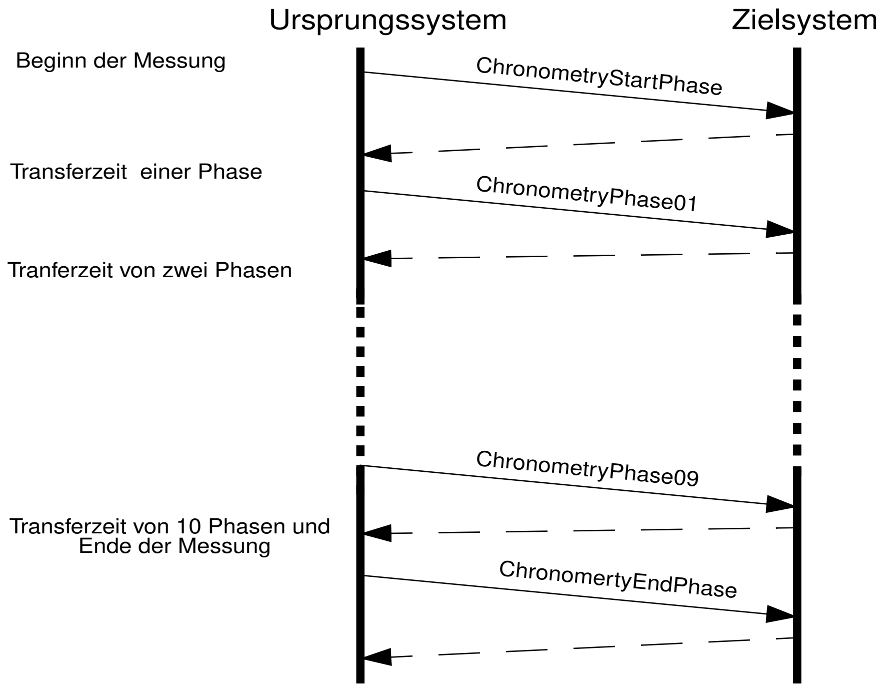

Abb. 3.12 Ablauf einer Messung des Phasenbasierten Agenten

3.2.4 Der monolithische Agent

Ein monolithischer Agent kann vereinfacht als ein Phasenbasierter Agent mit einer einzigen Phase betrachtet werden. Dieser Ansatz wurde angewandt, um einen monolithischen Agenten unter Verwendung des Frameworks und der bereits vorgestellten Basisklassen zu entwickeln. Der Vorteil dieser Vorgehensweise ist eine vergleichbare Codebasis beider Agentenkonzepte und somit eine optimale Voraussetzung für vergleichbare Messungen. Leider verfügt dieser Ansatz auch über gewisse Nachteile, da das Framework speziell für Phasenbasierte Agenten konzipiert ist und die Messergebnisse eines monolithischen Agenten negativ beeinflussen könnte.

Aus diesem Grund wird dargelegt, warum ein monolithischer Agent der unter Verwendung des Frameworks erstellt wird, sich nicht oder nur unwesentlich von einem direkt entwickelten monolithischen Agent unterscheidet.

Ist ein monolithischer Agent sehr komplex wird er nach einer Migration, je nach Situation, seine Verarbeitung mit unterschiedlichen Methoden fortsetzen, die unter Umständen durch verschiedene Objekte des Agenten implementiert sind. Die *Agent* Facette von Voyager ermöglicht es jedoch nur Methoden nach einer Migration aufzurufen die von dem primären Objekt der Facette implementiert werden. Diese Restriktion wird durch den flexiblen Mechanismus des Frameworks, mit dem mit einer beliebigen öffentlichen Methode des Agenten nach einer Migration die Ausführung fortgesetzt werden kann, aufgehoben (vgl. Abschnitt 3.1.3). Die Realisierung der Mobilität der Fragmente, des Frameworks kann also ohne Zweifel für die Implementierung eines monolithischen Agenten übernommen werden. Die verschiedenen Komponenten eines monolithischen Agenten müssen durch eine zentrale Klasse miteinander verbunden werden, wie das durch die Klasse *FragmentAgent* erfolgt. Ein monolithischer Agent sollte über eine öffentliche Schnittstelle verfügen, um den Zugriff auf seine Komponenten zu begrenzen. Genau diese Funktion übernimmt die Klasse *ChronometryInterface* bzw. die Unterklasse *Chronomety-BaseInterface*. Der einzige Mechanismus, der normalerweise nicht von einem komplexen monolithischer Agenten benötigt wird, ist der transparente Austausch von Phasen. Dieser zusätzliche Mechanismus ist aber in einer einzigen Klasse realisiert und fällt bei großen, komplexen Agenten nicht ins Gewicht.

Basierend auf den bisher gemachten Annahmen kann also ein monolithischer Agent für Messzwecke, unter Verwendung des Frameworks, entwickelt werden. Der Ablauf eines monolithischer Agenten ist dem eines Phasenbasierten Agenten sehr ähnlich. Zuerst wird eine Phase in ein mobiles Fragment eingesetzt, anschließend leitet diese Phase die Migration zum Zielort ein. Sobald die Phase wieder ihre Ausführung am Zielort aufnimmt, meldet sie das Erreichen ihres Endzustandes an die Workflowengine. Die Messung beginnt unmittelbar bevor die Phase ihre Ausführung beginnt und endet sobald sie ihren Endzustand an die Workflowengine signalisiert hat. Die Rückkehr des monolithischen Agenten wird durch eine zweite Phase realisiert, die nach Abschluss der Messung eingesetzt wird und deshalb nicht in das Messergebnis eingeht. Es müssen keine neuen Klassen für die Phasen des monolithischen Agenten entwickelt werden, da die Klassen *ChronometryStartPhase* und *ChronometryEndPhase* des Phasenbasierten Agenten genau den beschriebenen Ablauf bereits ermöglichen. Die Workflowengine bildet ein Objekt der Klasse *ChronometryMultiDecider*. Sie definiert nur den Ablauf des monolithischen Agenten und protokolliert die Messergebnisse.

3.2.5 Der erweiterte monolithische Agent

Das Ziel dieses Anwendungsfalles ist es eine Abschätzung zu ermöglichen, welchen Zeitanteil der Austausch der Phasen an der Migration eines Phasenbasierten Agenten hat und welchen der Transfer des Programmcodes. Der Anwendungsfall des monolithischen Agenten wurde zu diesem Zweck übernommen und eine neue Startphase entwickelt. Diese Phase ist ein Objekt der Klasse *ChronometryMultiPhase* und referenziert nach ihrer Erzeugung weitere neun Objekte.

Als Objekte wurden die bereits vorgestellten neun identischen Phasen des ersten Anwendungsfalles gewählt. Bei einer Migration der Phase müssen also insgesamt zehn Objekte und ihre Klassen zum Zielort übertragen werden. Ein Vergleich der Messwerte des monolithischen Agenten mit denen des erweiteren monolithischen Agent liefert eine Aussage, welchen Zeitaufwand der Transfer der neun zusätzlichen Klassen benötigt. Ein zweiter Vergleich der Messergebnisse des erweiterten monolithischen Agenten mit denen des Phasenbasierten Agenten zeigt welchen Zeitaufwand der Austausch der Phasen verursacht.

3.2.6 ChronometryServer

Für die Migration eines der vorgestellten Agenten werden auf dem Zielsystem eine Voyager-Laufzeitumgebung und ein Kommunikationspartner benötigt. Beide Anforderungen werden durch die Klasse *ChronometryServer* erfüllt, die als eigenständige Applikation auf dem Zielrechner ausgeführt wird. Im Rahmen der Messungen muss ein Agent sich nur bei dem Kommunikationspartner an- und abmelden können, um zu demonstrieren, dass er vor Ort seine Ausführung wieder aufgenommen hat. Die Klasse *ChronometryServer* implementiert zu diesem Zweck das Interface *IServer*.

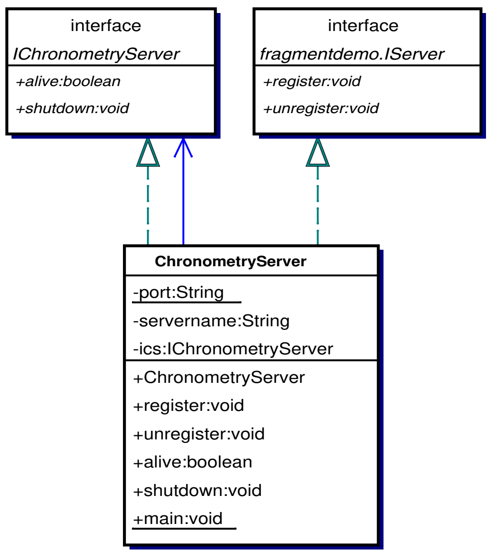

Abb. 3.13 UML-Darstellung der Klasse ChronometryServer

Um sicherzustellen, dass bei jeder Messung der gesamte Programmcode des Agenten zum Zielsystem übertragen wird und nicht die Klassen noch lokal im Speicher des Zielsystems vorliegen, muss die virtuelle Maschine von Java nach jeder Messung neu gestartet werden. Dies ist notwendig da die Version 1.2.2 der Java-Laufzeitumgebung bisher noch keine Möglichkeit bietet Klassen die einmal von der virtuellen Maschine geladen Klassen wurden zu entladen [GJS98]. Der Ablauf einer gesamten Messreihe kann sich aus mehreren hundert einzelnen Messungen zusammensetzen und sollte deshalb möglichst automatisiert sein. Aus diesem Grund implementiert die Klasse *ChronometryServer* zusätzlich das Interface *IChronometryServer*. Die

Methode *shutdown()* dieses Interfaces ermöglichen es einer entfernten Workflowengine, über einen Proxy das Programm und damit die komplette virtuelle Maschine zu beenden. Der Neustart der virtuellen Maschine erfolgt über ein Perlskript. Ob das Zielsystem nach einem Neustart für eine weitere Messung bereit ist, kann mit der Methode *alive()* festgestellt werden.

4 Messungen

4.1 Rahmenbedingungen der Messungen

Die Zeitdauer der Migration eines Agenten kann auf Grund von äußeren Faktoren, wie die Belastung der beteiligten Systeme oder des allgemeinen Netzwerkverkehrs, starken Schwankungen unterliegen. Jede Messung wurde deshalb mehrmals wiederholt und der Durchschnittswert berechnet. Diese Maßnahme richtet sich vor allem gegen kurzzeitige Belastungsspitzen. Typischerweise unterliegt der Datenverkehr innerhalb eines Netzwerkes aber auch längerfristigen, tageszeitlichen Schwankungen. Alle Messungen einer Messreihe wurden deswegen zur gleichen Tageszeit und in unmittelbarer Abfolge durchgeführt.

Eine Messreihe setzt sich aus den Messungen der drei entwickelten Agenten zusammen, deren Größe im Verlauf einer Messreihe schrittweise gesteigert wird. Die Gesamtgröße eines Agenten verteilt sich im Rahmen der Messungen immer gleichmäßig auf alle seine Phasen. Bei einer Agentengröße von 100 Kilobyte bedeutet das also, dass die einzelne Phase des phasenbasierten Agenten 10 Kilobyte groß ist.

Für alle Messungen wurde die Version 1.2.2 der Java Laufzeitumgebung und die Version 3.0 der Voyager Agentenpattform verwendet. Die genauen Messwerte jeder Messreihe können dem Anhang entnommen werden.

4.2 Transferzeitmessung über eine 100 Mbit-Verbindung

Für die erste Messreihe wurde ein lokales Netzwerk gewählt, bei dem Ursprungs- und Zielsystem der Agenten, direkt über eine Leitung mit 100 Mbit/s Datenrate verbunden sind. Die Größe der Agenten ist im Verlauf der Messreihe in 100 Kilobyte Schritten gesteigert worden, bis zu einem Maximum von 1000 Kilobyte. Jede Messung wurde zehn Mal wiederholt und der Durchschnitt der Messwerte in Abbildung 4.1 dargestellt.

Anfangsgröße der Agenten	0k
Maximale Größe der Agenten	1000k
Schrittbreite	100k
Anzahl der Wiederholungen einer Messung	10

Tab. 4.1 Parameter der 100 Mbit-Messreihe

Die als One bezeichnete kurz gestrichelte Linie repräsentiert die Messergebnisse des monolithischen Agenten mit einer einzigen Phase. Die Messwerte des erweiterten monolithischen Agenten werden durch die lang gestrichelte Linie wiedergegeben. Die Migration eines Phasenbasierten Agenten kann abhängig von der Situation nach dem Transfer von ein bis zehn Phasen abgeschlossen sein. Jede dieser Möglichkeiten wird durch eine eigene Linie wiedergegeben.

Greift man als Beispiel die Messwerte der Agenten bei einer Größe von 100 Kilobyte heraus, so ist aus dem Diagramm zu erkennen, dass der erweiterte monolithische Agent die längste Transferzeit mit mehr als 2000 Millisekunden hat. Der Messwert des monolithischen Agenten liegt zwischen den Transferzeiten der fünfen und sechsten Phase des Phasenbasierten Agenten, das bedeutet, sobald nur fünf oder weniger Phasen transferiert werden müssen, erzielt man durch den Einsatz von Phasenbasierten Agenten eine Reduktion der Transferzeit.

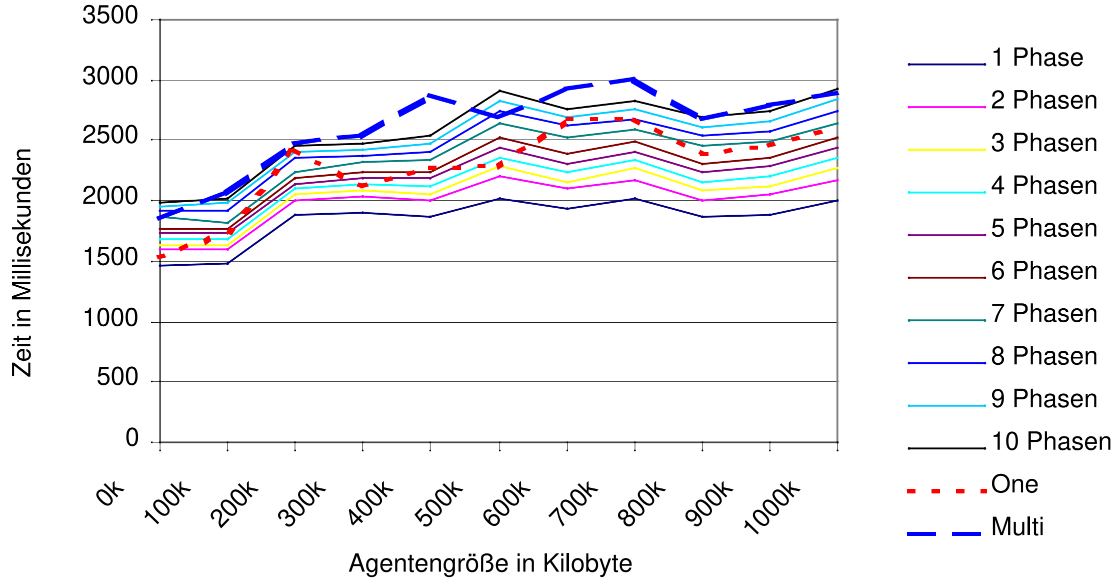

Abb. 4.1 Transferzeitmessung über eine 100 Mbit-Verbindung

Auswertung der Messreihe:

Betrachtet man die erste Messung bei der jeweils nur die Agenten ohne zusätzliche Daten migriert wurden, so erscheinen die Transferzeit von 1500 bis 2000 Millisekunden relativ hoch. Um zu verstehen wie diese Zeitspanne zustande kommt, muss der Ablauf einer Migration analysiert werden.

Sobald ein Agent seinen Transfer einleitet, erzeugt die Voyager-Laufzeitumgebung intern, dynamisch mehrere Proxies, die für die Migration von Agenten benötigt werden. Dieser Vorgang ist relativ zeitaufwendig, weil für die Erzeugung eines Proxies alle öffentlichen Methoden des zugehörigen Objektes bestimmt und nachgebildet werden müssen. Anschließend werden die Objekte des Agenten serialisiert und zum Zielsystem übertragen. Vor Ort werden sie wieder deserialisiert und der Programcode des Agenten nachgeladen. Der Programmcode der Agenten setzen sich je nach Anwendungsfall aus 20 bis 30 Klassen zusammen, mit einer Gesamtgröße von 30 bis 45 Kilobyte.

Da die Datenrate der Verbindung sehr hoch ist, kann die reine Übertragung der Daten nur eine sehr kleine Zeitspanne in Anspruch nehmen. Der Transfer der Klassen wird jedoch über zahlreiche separate Verbindungen abgewickelt, die aufgebaut und verwaltet werden müssen, was verhältnismäßig aufwendig ist. Die hohe Transferzeit der Agenten ohne zusätzliche Daten wird also in erster Linie durch die voyagerinternen Vorgänge verursacht.

Ungeachtet der starken Schwankungen der Messergebnisse im Verlauf der Messreihe, ist eine Zunahme der Transferzeit mit steigender Größe der Agenten zu erkennen. Ein Vergleich des ersten Messwertes mit dem letzten Messwert des Phasenbasierten Agenten, bei einem vollständigem Transfer aller Phasen, ergibt jedoch nur eine Zeitdifferenz von 945 Millisekunden. Dies entspricht einer Zunahme der Transferzeit von 48%, obwohl die Größe des Phasenbasierten Agenten um ein Vielfaches gesteigert wurde. Als Gründe für die geringe Zunahme der Transferzeit ist vor allem die hohe Bandbreite der Verbindung zu sehen, aber auch der geringe Aufwand, der für die Serialisierung und Übertragung des Bytevektors nötig ist.

Die Messergebnisse des Phasenbasierten Agenten bei einer vollständigen Migration von zehn Phasen unterscheiden sich nur unwesentlich von den Ergebnissen des erweiterten monolithischen Agenten. Der Zeitaufwand für den Phasenaustausch und die dafür nötigen entfernten Methodenaufrufe sind also innerhalb eines lokalen Netzwerkes bei hoher Datenrate sehr gering.

Dies gilt offensichtlich nicht für den Transfer von Objekten und Programmcode. Die Transferzeiten des erweiterten monolithischen Agenten liegen im Durchschnitt mehr als 300 Millisekunden über den Werten des monolithischen Agenten, egal wie groß der Bytevektor war. Diese Zeitdifferenz wird durch den Transfer der zusätzlichen Objekte und Klassen des erweiterten monolithischen Agenten verursacht. Erstaunlich ist jedoch die Höhe der Differenz, bedenkt man, dass die zusätzlichen Klassen mit weniger als 15 Kilobyte bei einer maximalen Agentengröße von 1000 Kilobyte eine Verlängerung der Transferzeit von mehr als 11% bezogen auf die Messwerte des monolithischen Agenten verursachen.

Zum Schluss der Auswertung muss noch die Kernfrage der Messungen beantwortet werden: Wann ist der Einsatz von Phasenbasierten Agenten im lokalen Netz sinnvoll?

Zu diesem Zweck wurde der prozentuale Zeitgewinn bzw. -verlust bei der Migration einer bestimmten Anzahl von Phasen gegenüber der Transferzeit des monolithischen Agenten in Abbildung 4.2 dargestellt. Aus dieser Abbildung kann man beispielsweise bei einer Agentengröße von 100 Kilobyte erkennen, dass die Zeitersparnis bei einem Transfer von nur einer Phase ca. 15% ausmacht. Benötigt der Phasenbasierte Agent mehr als fünf Phasen um seine Aufgabe zu erfüllen, so übersteigt die Transferzeit des Phasenbasierten Agenten die des monolithischen Agenten. Im Extremfall, bei einer Migration von 10 Phasen, ist die Transferzeit des Phasenbasierten Agenten um ca. 15% länger. Für den effektiven Einsatz von Phasenbasierten Agenten ist es aber wichtig, dass möglichst erst bei einem Übertragung von fast allen Phasen die Transferzeit des monolithischen Agenten übertroffen wird. Diese Bedingung ist im wesentlichen bei einer Agentengröße von mehr als 500 Kilobyte erreicht.

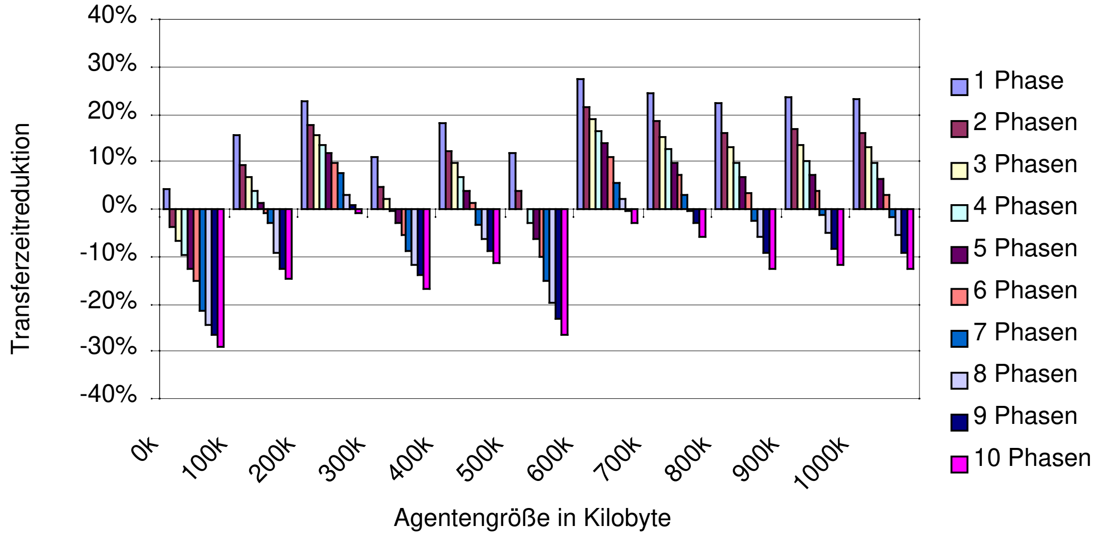

Abb. 4.2 Transferzeitreduktion bei eine 100 Mbit-Verbindung

Die bereits erwähnten Schwankungen der Messwerte sind bei den Werten der Agentengröße von 500 und 600 Kilobyte besonders auffällig. Die Ursache wird in der wechselnden Belastung der Testsysteme gesehen. Solche Schwankungen können sich auf die Messergebnisse der einzelnen Agenten unterschiedlich auswirken, da die Messungen der drei Agenten nicht parallel, sondern hintereinander durchgeführt wurden.

4.3 Transferzeitmessung über eine 10 Mbit-Verbindung

Nachdem bei der ersten Messreihe die Transferzeiten der Agent mit zunehmender Größe relativ schwach gestiegen sind, wurde bei der zweiten Messreihe eine Verbindung mit einer geringeren Datenrate von 10 Mbit/s gewählt. Das Ausgangssystem der Agenten war dabei wieder direkt mit dem Zielsystem verbunden. Alle Parameter der ersten Messreihe wurden beibehalten, um einen direkten Vergleich der Messreihen zu ermöglichen.

Anfangsgröße der Agenten	0k
Maximale Größe der Agenten	1000k
Schrittbreite	100k
Anzahl der Wiederholungen einer Messung	10

Tab. 4.2 Parameter der 10 Mbit-Messreihe

Die gemessenen Transferzeiten wurde wie bei der ersten Messreihe dargestellt und sind Abbildung 4.3 zu entnehmen.

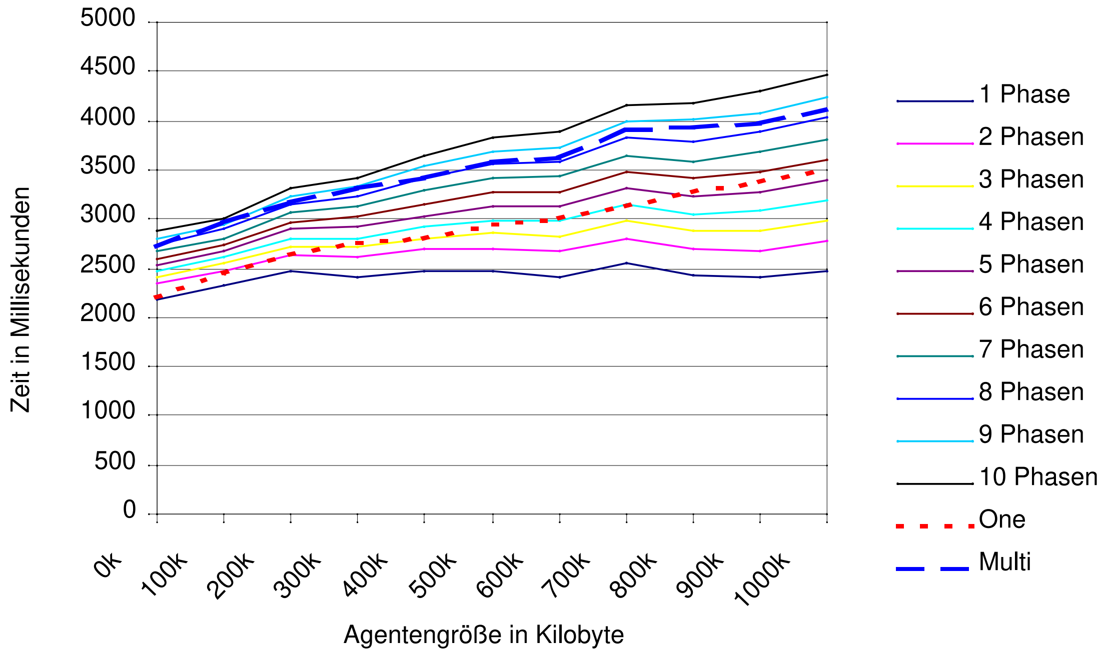

Abb. 4.3 Transferzeitmessung über eine 10 Mbit-Verbindung

Auswertung der Messreihe:

Mit zunehmender Größe der Agenten ist ein Anstieg der Transferzeiten deutlich zu erkennen. Diese begründet sich in der geringeren Datenrate der Verbindung.

Ein Vergleich der Transferzeiten des monolithischen Agenten der ersten Messreihe mit den Messwerten dieser Messreihe zeigt jedoch, dass die Übertragungszeiten im Durchschnitt nur um 27% zugenommen haben und damit weit niedriger sind als auf Grund der 10-fach geringeren Datenrate der Verbindung zu erwarten war. Dies deutet daraufhin, dass die zur Verfügung stehende Datenrate unter Umständen in beiden Fällen nicht genutzt werden kann und der Hauptanteil der Transferzeit durch Abläufe innerhalb der Voyager-Laufzeitumgebung verursacht wird. Die zweite Annahme trifft sicherlich zu, da die Transferzeiten der Agenten ohne zusätzliche Daten um ein Vielfaches höher sind, als für die reine Übertragung ihres Zustandes und des zugehörigen Programmcodes nötig wären.

Im Verlauf der Messreihe vergrößern sich die Abstände zwischen den einzelnen Messlinien des Phasenbasierten Agenten stetig. Die Differenz der Messlinien aufeinanderfolgender Phasen bildet die Transferzeit der letzten Phase. Die Vergrößerung der Abstände zwischen den Messlinien aufeinaderfolgender Phasen bedeutet also eine Zunahme ihrer Transferzeit. Diese Entwicklung ist eine Konsequenz der Zunahme der Agentengröße, da die Gesamtgröße des Phasenbasierten Agenten sich gleichmäßig auf alle Phasen verteilt. Aus der Abbildung 4.3 ist also deutlich zu erkennen, je größer die Phasen werden, desto länger dauert ihr Transfer.

Die Messwerte des erweiterten monolithischen Agenten liegen bei dieser Messreihe deutlich unter den Transferzeiten des Phasenbasierten Agenten bei einer vollständigen Übertragung aller Phasen. Im Gegensatz zur ersten Messreihe kann der Zeitaufwand für den Austausch der Phasen also nicht mehr völlig vernachlässigt werden.

Sehr auffällig ist der große Abstand zwischen der Messlinie des monolithischen Agenten und der Messlinie des erweiterten monolithischen Agenten. Die Messwerte des monolithischen Agenten liegen im Durchschnitt mehr als 600 Millisekunden unter den Werten des erweiterten monolithischen Agenten, obwohl sie sich in ihrer Größe nur um 15 Kilobyte unterscheiden. Dieser Abstand bedeutet bei einem Transfer des monolithischen Agenten von 1000 Kilobyte Größe mehr als 18% Zunahme der Transferzeit. Der Unterschied zwischen dem monolithischen Agenten und dem erweiterten monolithischen Agenten hat sich also gegenüber der ersten Messreihe noch verstärkt.

Trotz der unerwartet hohen Aufwandskosten für den Transfer der zusätzlichen Klassen des Phasenbasierten Agenten kann durch den Einsatz des phasenbasierten Konzeptes eine Reduktion der Transferzeit erreicht werden. Beträgt die Größe des Agenten 1000 Kilobyte, so wird bis zu einer Übertragung von fünf Phasen die Transferzeit reduziert. Auf Grund der Entwicklung der Messwerte ist anzunehmen, dass der Nutzen des phasenbasierten Konzeptes bei noch größeren Agenten weiter steigt. Ist die Größe des Agenten jedoch geringer als 800 Kilobyte, so übertrifft die Transferzeit des Phasenbasierten Agenten die des monolithischen Agenten schon nach einem Transfer von ein bis vier Phasen und kann eine Steigerung der Transferzeit von mehr als 30% bedeuten.

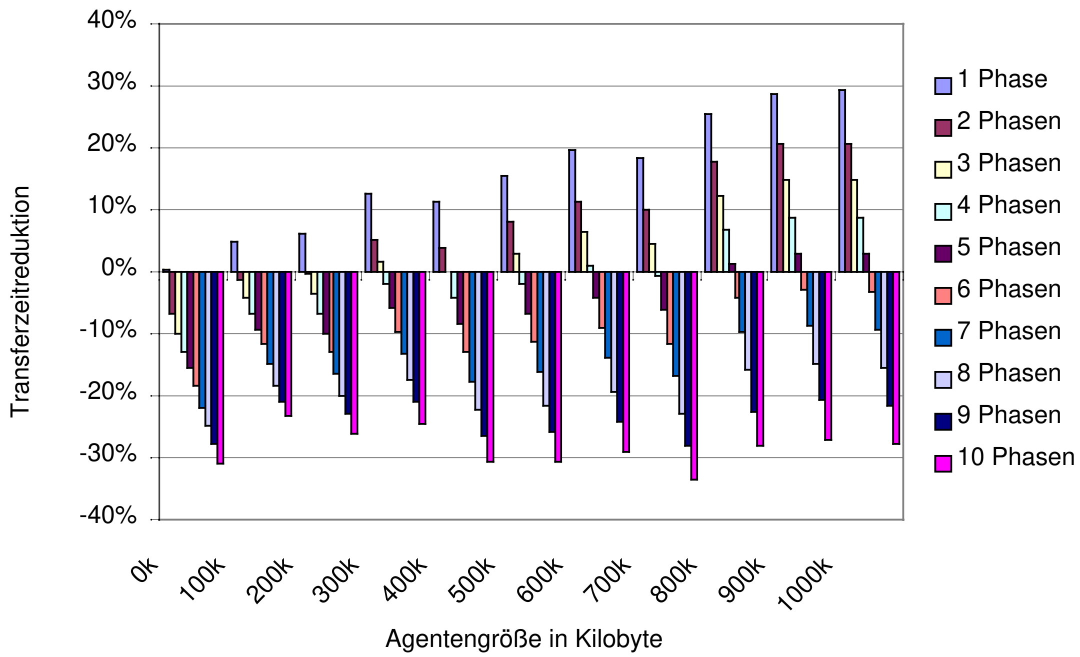

Abb. 4.4 Transferzeitreduktion bei einer 10 Mbit-Verbindung

4.4 Transferzeitmessung über eine ISDN[1]-Verbindung

Nachdem bereits bei Verbindungen mit großer Datenrate der Einsatz von Phasenbasierten Agenten eine Reduktion der Transferzeit bewirken kann, sollte mit der folgenden Messreihe ermittelt werden, ob dies auch für Verbindungen mit geringerer Datenrate gilt. Beide Teilnehmersysteme wurden deshalb über eine ISDN-Leitung miteinander verbunden. Die Datenrate einer solchen Verbindung beträgt ohne Kompression maximal 64 kBit/s. Die Parameter der Messung sind Tabelle 4.1 zu entnehmen.

Anfangsgröße der Agenten	0k
Maximale Größe der Agenten	90k
Schrittbreite	10k
Anzahl der Wiederholungen einer Messung	3

Tab. 4.3 Parameter der ISDN-Messreihe

Die Darstellung der Messergebnisse erfolgt auf die gleiche Weise wie bei den bisherigen Messreihen in Abbildung 4.5.

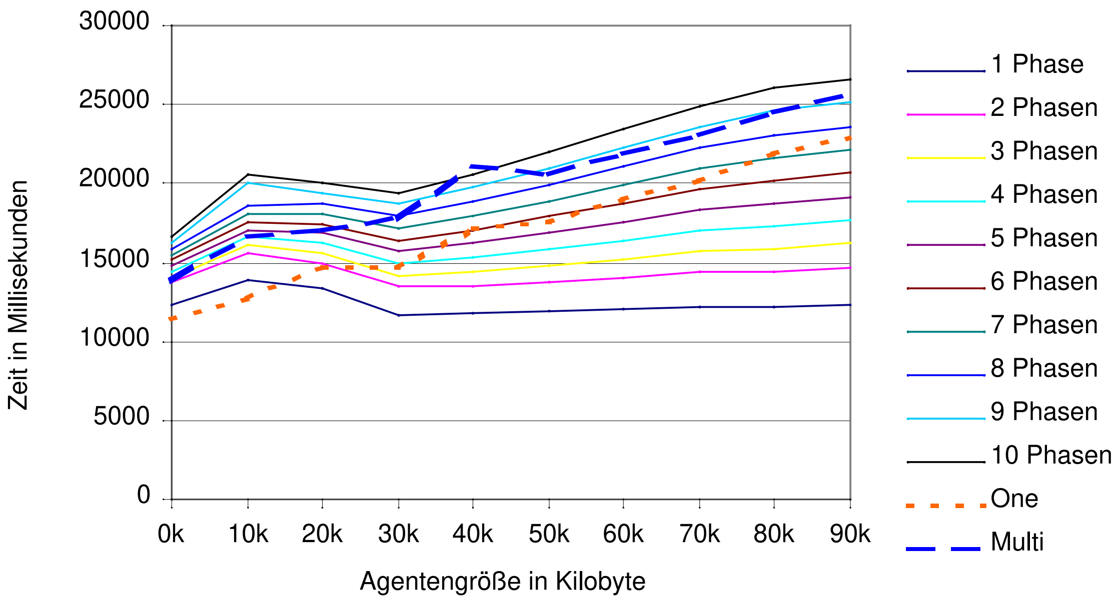

Abb. 4.5 Transferzeitmessung über eine ISDN-Verbindung

1. Integrated Services Digital Network

Auswertung der Messreihe:

Die Werte der ersten Messung verdeutlichen die wesentlich geringere Datenrate der Verbindung durch Transferzeiten zwischen 11000 bis mehr als 17000 Millisekunden. Mit steigender Größe der Agenten ist bei dieser Messreihe noch deutlicher die Zunahme der Transferzeit zu erkennen. Vergleicht man wieder den ersten und letzten Messwert des Phasenbasierten Agenten bei vollständigem Transfer aller Phasen, so ergibt sich eine Zunahme der Transferzeit um 60%.

Auffällig ist das lokale Maximum der Messwerte des Phasenbasierten Agenten bei einer Größe von 10 Kilobyte. Die Ursache muss wieder in einer Belastungsspitze der beteiligten Systeme oder des Netzwerkes gesehen werden, da zumindest auch der Messwert des erweiterten monolithischen Agenten stärker erhöht ist.

Die Transferzeiten des erweiterten monolithischen Agenten liegen bei dieser Messreihe etwa in gleicher Höhe wie die Transferzeiten des Phasenbasierten Agenten nach nur neun Phasen und damit klar unter den Messwerten des Phasenbasierten Agenten bei einem vollständigen Transfer aller Phasen. Der Zeitaufwand für den Austausch der Phasen ist also deutlich sichtbar. Als Ursache für die höheren Transferzeiten des Phasenbasenbasierten Agenten werden die entfernten Methodenaufrufe für den Phasenaustausch gesehen, da alle sonstigen Vorgänge nicht durch die geringere Datenrate der Verbindung beeinflusst werden.

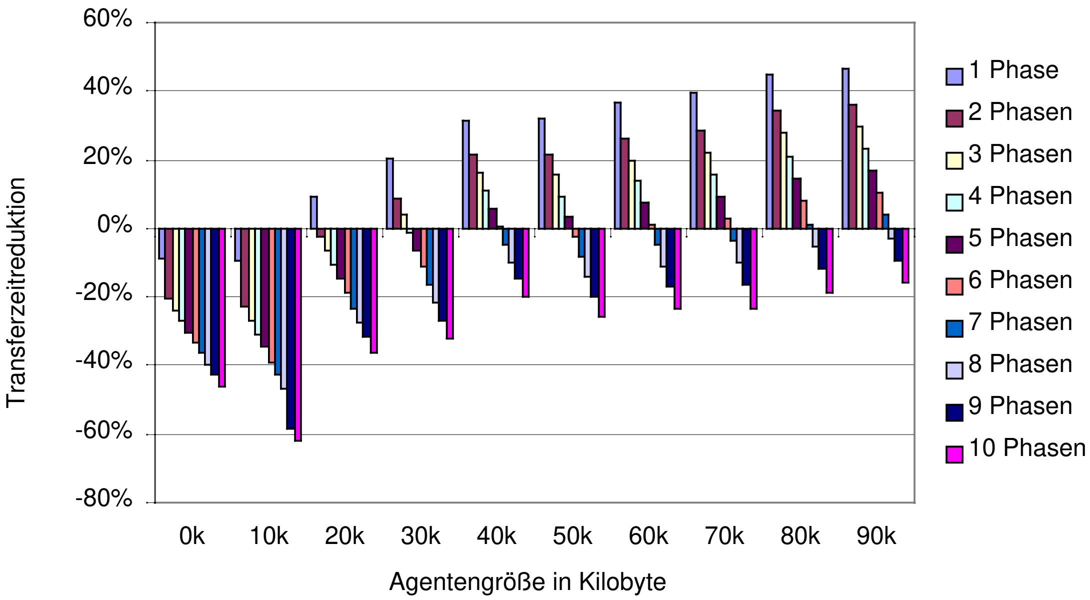

Abb. 4.6 Transferzeitreduktion bei einer ISDN-Verbindung

Da im Rahmen der Messungen alle Phasen des Phasenbasierten Agenten die gleiche Größe haben, sollten ihre Transferzeiten im wesentlichen identisch sein. Die Transferzeit einer Phase kann Abbildung 4.5 entnommen werden, indem man den Abstand zwischen den Messwert dieser Phase und der vorangegangenen Phase bestimmt. Die Annahme wird im Wesentlichen durch die Messwerte bestätigt. Lediglich die Transferzeit der ersten und zweiten Phase unterscheiden sich deutlich sichtbar von den restlichen Phasen. Bei der ersten Phase ist die Erklärung für die längere Transferzeit der zusätzliche Transport von verschiedenen Objekten des mobilen Fragments und des zugehörigen Programcodes. Die einzige wirkliche Ausnahme bildet die zweite Phase. Ihre Transferzeit ist immer etwas größer als die der folgenden Phasen. Die Gründe für diese Anomalie muss in voyagerinternen Vorgängen begründet sein, da der Ablauf des Phasenaustausches bei jeder Phase gleich ist.

Der stärkere Anstieg der Transferzeit mit zunehmender Agentengröße zeigt sich auch in der Darstellung (Abbildung 4.6) der Transferzeitreduktion. Die Zeitverluste bei einer Agentengröße von weniger als 30 Kilobyte sind mit bis zu 61% deutlich größer als bei den Messungen mit hoher Datenrate. Dies kann auf die geringere Datenrate der Verbindung zurückgeführt werden, da der Transfer der zusätzlichen Klassen für die einzelnen Phasen und die entfernte Kommunikation des Phasenbasierten Agenten mehr Zeit beanspruchen. Ist der Agent größer als 50 Kilobyte, so ist der zusätzliche Aufwand des Phasenbasierten Agenten im Verhältnis zur Transportzeit des monolithischen Agenten gering und der Einsatz des Phasenbasierten Agenten sinnvoll. Bei einer Agentengröße von 90 Kilobyte kann der Einsatz des Phasenbasierten Agenten eine Zeitersparnis von mehr als 40% bedeuten. Ein höhere Transferzeit als bei einem monolithischen Agenten wird auch erst bei einer Übertragung von mehr als sieben Phasen erreicht.

4.5 Transferzeitmessung über eine Modemverbindung

Die Ergebnisse der ISDN-Messung haben gezeigt, dass der Einsatz von Phasenbasierten Agenten vor allem bei Verbindungen mit geringer Bandbreite Vorteile bietet. Aus diesem Grund wurde für die letzte Messreihe eine Modemverbindung gewählt. Die Datenrate einer solchen Verbindung liegt mit maximal 56 kBit/s noch unter der einer ISDN-Verbindung. Doch auch diese Datenrate wird in der Praxis selten erreicht. Die tatsächlich Datenrate der Verbindung betrug im Rahmen der Messungen 44 kBit/s. Außerdem wird diese Transfergeschwindigkeit nur in eine Richtung erreicht, während die Datenrate in die Gegenrichtung 33.6 kBit/s beträgt. Die Verbindung der beiden Teilnehmersysteme führt vom Ursprungssystem über eine Modemverbindung zu einem Vermittlungsrechner. Von dort gelangen die Daten über drei weitere Systeme zum Zielsystem. Da die Modemverbindung eine wesentlich geringere

Anfangsgröße der Agenten	0k
Maximale Größe der Agenten	100k
Schrittbreite	10k
Anzahl der Wiederholungen einer Messung	3

Tab. 4.4 Parameter der Modemmessreihe

Datenrate besitzt als die folgenden Verbindungsstrecken, sollte der Zeitaufwand, der erforderlich ist die Daten zwischen den drei Vermittlungsrechnen zu transportieren, weitgehend vernachlässigbar sein. Die maximale Größe der Agenten wurde auf 100 Kilobyte beschränkt, da bereits bei dieser Größe die Transferdauer unter optimalen Bedingungen über 14 Sekunden liegt. Alle sonstigen Parameter der Messung sind aus Tabelle 4.4 zu entnehmen.

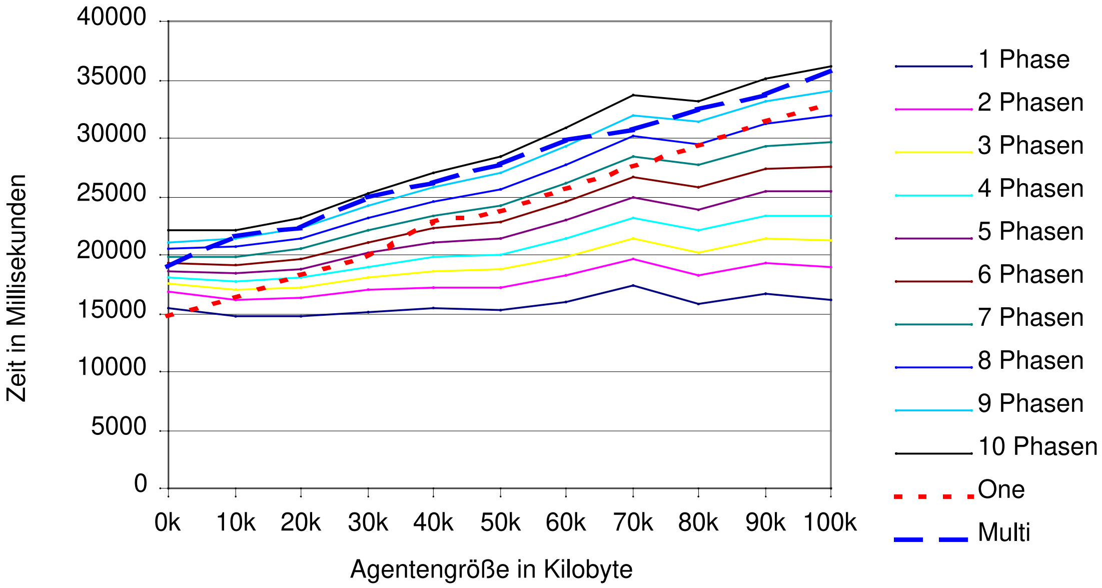

Abb. 4.7 Transferzeit über eine Modemverbindung

Auswertung der Messreihe:

Vergleicht man die Transferzeiten der ersten Messung mit den Ergebnissen der ISDN-Messreihe, so wird die geringere Datenrate der Modemverbindung deutlich. Bei der Übertragung des monolithischen Agenten ohne zusätzliche Daten macht diese gegenüber den Messwerten der ISDN-Messreihe eine Zunahme der Transferzeit von 29% aus. Der Anstieg der Transferzeit des Phasenbasierten Agenten bei einem vollständigem Transfer aller Phasen und wachsender Gesamtgröße ist in etwa mit der Zunahme der Transferzeiten bei der ISDN-Messreihe vergleichbar.

Die Zeitdauer der Übertragung der einzelnen Phasen ist sehr gleichmäßig. Die einzigen Ausnahmen bilden wieder die erste und zweite Phase. Ansonsten ist bei dieser Messreihe sehr eindeutig zu erkennen, dass mit steigender Agentengröße der Einsatz eines Phasenbasierten Agenten immer mehr Zeitersparnis erbringen kann, vorausgesetzt, dass nicht alle Phasen transferiert werden müssen.

Mit dem Diagramm der Transferzeitreduktion (Abb. 4.8) kann sehr einfach bestimmt werden, dass bis zu einer Größe von 40 Kilobyte der Nutzen des Konzeptes relativ eingeschränkt ist. Übersteigt die Größe des Agenten jedoch 70 Kilobyte, werden die Zeitverluste im Verhältnis zur Transferzeit für den Transfer des gesamten Phasenbasierten Agenten immer geringer und der Einsatz wird immer effektiver.

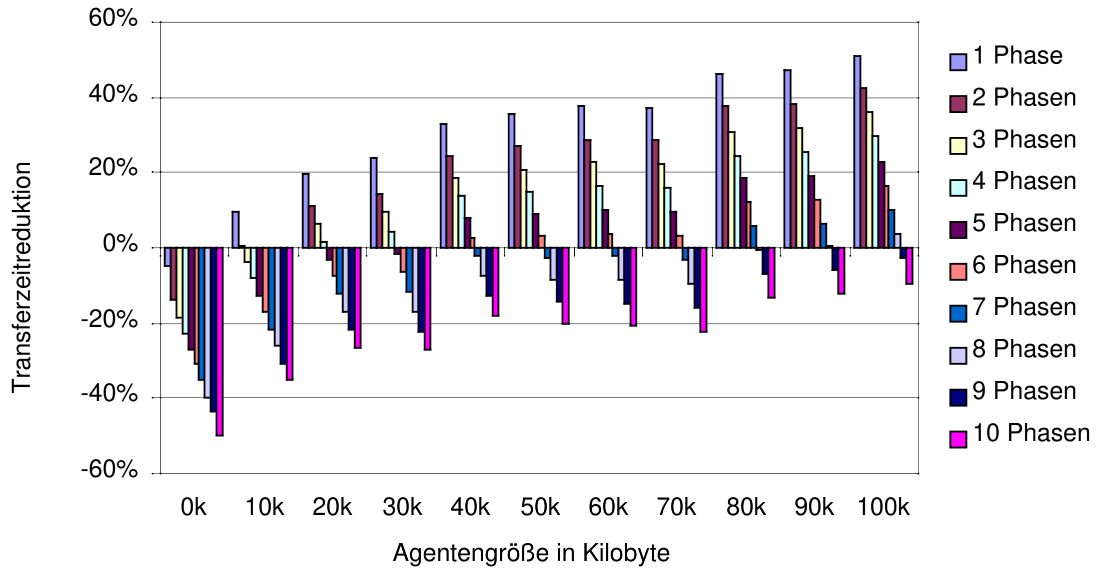

Abb. 4.8 Transferzeitreduktion bei einer Modemverbindung

4.6 Zusammenfassung der Messergebnisse

Nachdem die Messwerte der einzelnen Messreihen bereits ausführlich diskutiert wurden, soll nun eine übergreifende Bewertung und Zusammenfassung der Ergebnisse erfolgen.

Die Messwerte der ersten beiden Messreihen bei Verbindungen mit hoher Datenrate haben gezeigt, dass unter diesen Voraussetzungen die reine Übertragung der Daten nicht den Hauptanteil der Transferzeit verursacht, sondern Abläufe die vor und nach der Übertragung der Daten erfolgen. Diese Annahme wird durch die hohen initialen Transferzeiten der Agenten und deren anschließend verhältnismäßig geringen Zunahme bei steigender Agentengröße bestätigt.

Ein Vergleich der Transferzeiten zwischen dem monolithischen Agenten und dem erweiterten monolithischen Agenten hat sehr deutlich gezeigt, dass die Vorgänge für den Transfer des Programmcodes hier besonders zeitaufwendig sind. Dies begründet sich in der separaten Übertragung jeder einzelnen Klasse. Verfügt ein Agent also über 30 bis 40 Klassen, so wird die entsprechende Zahl an Verbindungen aufgebaut und die Transfers vorgenommen. Leider erfasst die gewählte Messgröße genau diesen Umstand nicht, sondern gibt lediglich die Größe des Zustandes des Agenten wieder. Der Transfer des Zustandes eines Agenten erfordert jedoch einen relativ geringen Aufwand, da die Übertragung über eine einzige Verbindung erfolgt. Eine Konsequenz dieser Erkenntnis ist, dass ein realer Agent mit dem zugehörigen Programmcode bei einem Transfer über Verbindungen mit hoher Datenrate wesentlich höhere Transferzeiten haben muss, als im Rahmen der Messungen festgestellt worden ist, da er aus mehr Klassen besteht.

Ein weiterer Faktor der bisher nicht im Rahmen der Messungen berücksichtigt wurde, ist die Leistung der Teilnehmersysteme und der Java Laufzeitumgebung. Da die Abläufe innerhalb der Teilnehmersysteme einen großen Anteil der Transferzeit verursachen, kann durch den Einsatz von leistungsfähigen Systemen die Transferzeit stark verkürzt werden.

Nachdem aufgezeigt wurde, welche Problematiken mit der Messgröße und den erzielten Ergebnissen verbunden sind, stellt sich die Frage, welche Aussagekraft der Vergleich zwischen den Messwerte des monolithischen und des Phasenbasierten Agenten hat.

Zuerst muss angemerkt werden, dass der Phasenbasierte Agent gegenüber dem monolithischen Agenten vor allem wegen seiner zusätzlichen Klassen schlechter abschneidet. Im realen Einsatz verfügt ein Phasenbasierter Agent meistens über weniger Programcode der transferiert werden muss, als ein monolithischer Agent, selbst bei einem Transfer aller Phasen, da die Workflowengine nicht migriert wird. Der Vergleich gibt unter den gestellten Bedingungen Auskunft über den absoluten Minimalnutzen, der unter realen Bedingungen wahrscheinlich bei weitem übertroffen wird.

Die Messungen wurden weiterhin nur innerhalb eines lokalen Netzwerkes durchgeführt. Das reale Einsatzgebiet von mobilen Agenten ist aber das Internet. Verfügt ein Agent also über einen größeren Programmcode und wird über Verbindungen migriert, die hohen Belastungen unterliegen, so übertreffen die Transferzeiten die hier gemessenen Werte.

Diese Annahmen führen zu dem Schluss, dass die Reduktion der Transferzeit auch bei Verbindungen mit großer Datenrate wesentlich höher ausfällt, als in den Messungen gezeigt wurde und der Einsatz von Phasenbasierten Agenten sinnvoll ist. Eine klare Aussage wie groß und komplex ein Agent sein sollte um das Konzept des Phasenbasierten Agenten effizient einzusetzen, kann auf Grund der beschriebenen Ergebnisse nicht gemacht werden.

Eine ganz andere Situation bietet sich im zweiten Teil der Messungen bei Verbindungen mit geringer Datenrate. Die Transferzeiten der Agenten nehmen bei steigender Agentengröße wesentlich schneller zu als bei den Messreihen mit hoher Datenrate und sind außerdem auch im Durchschnitt um ein Vielfaches höher.

Da die Abläufe innerhalb der Teilnehmersysteme für die Mirgration eines Agenten nicht von der geringereren Datenrate beeinflusst werden, muss die Übertragung der Daten die Zunahme der Transferzeit verursachen. Die für eine Migration nötigen Abläufe nehmen dann auf Grund der hohen Transferzeiten nur noch einen geringen Anteil der Transferzeit ein.

Eine Konsequenz dieser Erkenntnisse ist, dass die gewählte Messgröße die Transferzeit eines realen Agenten bei niedrigen Bandbreiten wesentlich besser erfasst als bei hohen Datenrate, da sie die zu übertragenden Daten misst, die den Hauptanteil der Transferzeit verursachen.

Natürlich muss auch hier darauf hingewiesen werden, dass bei einem realen Agenten gleicher Größe, die Transferzeit gegenüber den vorgestellen monolithischen und Phasenbasierten Agenten noch zunehmen kann, da die Übertragung des Programmcodes auch bei geringen Datenrate aufwendiger ist als der Transfer seines Zustandes.

Dennoch kann anhand der Messreihen klar die Aussage getroffen werden, dass der Einsatz von Phasenbasierten Agenten bei niedrigen Datenrate, ab einer Agentengröße von 80 Kilobyte die Transferzeit entscheidend reduzieren kann, solange nicht alle Funktionalitäten des Agenten am Zielsystem benötigt werden.

5 Zusammenfassung und mögliche Ergänzungen

5.1 Zusammenfassung

Die Hauptaufgabe dieser Arbeit bestand in dem Entwurf und der Implementierung eines Frameworks für Phasenbasierte Agenten. Aufbauend auf diesem Framework wurden verschiedene Agenten für Messzwecke entwickelt. Ziel der Messungen war ein Vergleich des monolithischen mobilen Agenten mit dem Konzept des Phasenbasierten Agenten in Bezug auf Transferzeit und Datenübertragung.

Die für das Verständnis der Arbeit nötigen Grundlagen sind Inhalt von Kapitel 2. Hier wurde vor allem das Konzept des mobilen Agenten erklärt und die Weiterentwicklung dieses Konzeptes zum Phasenbasenbasierten Agenten dargestellt. Den zweiten Teil des Grundlagenkapitels bildet die Beschreibung der Voyager-Laufzeitumgebung, die als Basis für die Entwicklung des Frameworks diente.

In Abschnitt 3.1.1 wurde zuerst analysiert welche Anforderungen das Framework erfüllen sollte und anschließend erläutert durch welche Komponenten des Frameworks die entsprechende Anforderungen erfüllt werden. Die einzelnen Komponenten sind im weiteren Verlauf von Abschnitt 3.1 detailliert beschrieben und ihr Aufgabenbereiche und Funktion innerhalb des Framework verdeutlicht worden. Dabei wurden Lösungswege für wichtige Aspekte des Phasenbasierten Konzeptes entwickelt, wie der Austausch von Phasen und die Funktionsweise des Fragmentinterface, aber auch frameworkspezifische Problematiken gelöst, wie der Entwurf geeigneter Schnittstellen für die einfache Erstellung von Anwendungsfällen.

Nachdem mit der Entwicklung des Frameworks die Grundlagen für den Entwurf von Anwendungsfällen gelegt war, wurde in Abschnitt 3.2.1 die Rahmenbedingungen für die Entwicklung der Agenten bestimmt. Im Folgenden wurden ein Phasenbasierter Agent und zwei monolithische Agenten entwickelt. Die beiden monolithischen Agenten unterscheiden sich in ihrem Aufbau und ermöglichen durch den Vergleich ihrer Messwerte eine bessere Bewertung der Messergebnisse.

Die Beschreibung der Messungen und die Bewertung der erzielten Ergebnisse bilden den Abschluss der Arbeit. Die Rahmenbedingungen der Messungen wurden dabei zu Beginn des Abschnitts festgelegt, um einen Vergleich der Ergebnisse der Messreihen untereinander zu ermöglichen. Es wurden vier Messreihen bei Verbindungen mit unterschiedlichen Datenrate vorgestellt und ihre Messwerte analysiert. Die Ergebnisse der Messreihen wurden ausführlich in Abschnitt 4.6 bewertet und zusammengefasst.

5.2 Mögliche Ergänzungen

Das Framework bietet bisher die Möglichkeit auf einfache Art und Weise Phasenbasierte Agenten zu entwickeln, die aus einem mobilen Fragment und einer stationären Komponente bestehen. Einer der Grundgedanken des phasenbasierten Agentenkonzeptes ist aber die Verteilung des Agenten über eine ganze Reihe von Systemen, durch eine beliebige Anzahl von Fragmenten. Um die Erzeugung und Verwaltung von mehreren parallel verarbeitenden Fragmenten für einen Phasenbasierten Agenten zu ermöglichen, müssen folgende Modifikationen und Erweiterungen des Frameworks vorgenommen werden:

(1) Zuerst muss geklärt werden, welche Komponenten der Anwendungsfälle in der Lage sein sollen, Fragmente zu erzeugen. Dafür kommen zwei Bestandteile der Anwendungsfälle in Frage. Die Phase eines bereits vorhandenen Fragments und die Workflowengine des Agenten. Um einem Anwendungsentwickler möglichst viel Freiraum zu lassen, sollten beide Komponenten neue Fragmente erzeugen können. Das Framework muss dafür an verschiedenen Stellen verändert werden. Stößt die Phase eines bereits vorhandenen Fragments die Erzeugung eines neuen Fragments an, so wird durch das Objekt der Klasse *FragmentAgent* die bereits aktive Phase über ihre *clone()* kopiert und nach der Erzeugung der restlichen Objekte ausgeführt. Leitet die Workflowengine die Erzeugung eines neuen Fragments ein, so ruft er eine Methode der Klasse *FragmentProvider* auf, mit dem Namen und der Methode der Phase des neuen Fragments als Parameter. Die Methode der Klasse *FragmentProvider* erzeugt dann die einzelnen Objekte des Fragments und ruft die aktive Phase auf.

(2) Um mehrere Fragmente durch das Framework verwalten zu können, muss jedem Fragment ein fester Identifikator zugeordnet werden, der über das Interface *IDecider* an die Workflowengine weitergegeben wird. Die Workflowengine kann dann auf Grund dieses Identifikators die Phasen unterscheiden und entsprechend ihrer Ergebnisse reagieren. Neue Fragmente müssen sich natürlich, egal auf welche der beiden Wege sie erzeugt wurden, bei der Workflowengine registrieren. Hat die Workflowengine die Erzeugung eines neuen Fragments selbst ausgelöst, so erhält sie als Rückgabewert der Erzeugung den Identifikator des neuen Fragments. Wird die Erzeugung eines neuen Fragments durch eine Phase eingeleitet, muss das Objekt der Klasse *FragmentAgent* des zugehörigen Fragments den Identifikator des neuen Fragments an die Workflowengine übermitteln.

(3) Besteht ein phasenbasierter Agent aus einer Vielzahl von mobilen Fragmenten, so müssen die Phasen dieser Fragmente die Möglichkeit haben miteinader zu kommunizieren, um eine Aufgabe effektiv erfüllen zu können. Eine Phase wird in der Regel nicht einzeln mit einer anderen Phase kommunizieren, sondern mit einer ganzen Gruppe von Phasen oder sogar allen aktiven Phasen. Das Framework sollte also über Mechanismen verfügen, die eine Gruppenkommunikation ermöglichen. Da die Phasen in ihrer Funktion sehr unterschiedlich sein können, sind sicher nicht alle Nachrichten für eine Phase relevant. Eine Phase sollte sich deshalb nur für eine bestimmte Klasse von Nachrichten registrieren können. Mit dieser Art der Gruppenkommunikation kann auch der Workflowengine die Möglichkeit geboten werden Nachrichten an die Fragmente zu schicken, denn bisher kann die Workflowengine nur auf ein Fragment Einfluss nehmen, wenn die aktive Phase des Fragments ihren Endzustand erreicht hat und ausgetauscht wird. Es gibt aber Fälle bei denen es sehr wichtig ist, dass die Workflowengine eine Phase sofort benachrichtingen kann. Zum Beispiel wenn die Applikation, die den Phasenbasierten Agenten gestartet hat beendet werden soll und deshalb alle Fragmente zu ihrem Ursprungsort zurückkehren müssen.

Ein wichtiges Ergebnis der Messungen war, dass die Übertragung des Programcodes der Agenten einen sehr hohen Anteil der Transferzeit einnimmt. Dies liegt vor allem an der separaten Übertragung jeder einzelnen Klasse. Im Falle des Frameworks bedeutete das den Auf- und Abbau von Verbindungen für mehr als 30 Transfers. Um die Transferszeit des Programmcodes zu reduzieren wäre es deshalb sinnvoll, die Klassen einer Phase bzw. des Frameworks in jeweils ein Java Archiv zu verpacken. Die Übertragung eines solchen Archives benötigt nur den Auf- und Abbau einer einzigen Verbindung.

Das Problem an dieser Idee ist die Funktion der *Agent* Facette von Voyager. Sie ermöglicht es nur einen einzigen Resource Loader zu spezifizieren, der nicht verändert oder gewechselt werden kann. Würde man alle Klassen in ein Archiv verpacken, muss man in Kauf nehmen, dass immer der gesamte Programmcode eines Phasenbasierten Agenten zum Zielsystem übertragen wird. Dies ist aber nicht mit dem Konzept des Phasebasierten Agenten vereinbar deshalb sollte ein Mechanismus entwickelt werden, der einem mobilen Fragment ermöglicht dynamisch weitere Resource Loader in die Voyager-Laufzeitumgebung zu integrieren.

Eine Erweiterung, die sich nicht auf das Desgin des Frameworks bezieht, ist die Entwicklung einer besseren Maßeinheit für die Größe eines mobilen Agent. Beide in dieser Arbeit vorgestellen Einheiten haben zu große Mängel um reale Aussagen über die Transferzeit eines Agenten bestimmter Größe und Komplexität zu liefern. Die Lösung des Problems wird in der Entwicklung einer Standardklasse gesehen. Eine solche Klasse wird so entworfen, dass sie der durchschnittlichen Klasse der Java-Klassenbibliothek entspricht. Um herauszufinden wie ein solche Klasse beschaffen sein sollte, müssen verschiedene Kriterien wie Anzahl der Programmzeilen, Anzahl der Variablen und Methoden bestimmt werden. Anschließend kann auf Grund dieser Kriterien der Programmcode der Java-Klassenbibliothek statistisch ausgewertet werden. Die so entwickelte Klasse kann dann als Maßeinheit dienen, da sie alle Faktoren die für die Migration eines realen Agenten wichtig sind erfasst.

Anhang A - Messwerttabellen

	0k	100k	200k	300k	400k	500k	600k	700k	800k	900k	1000k
1 Phase	1472	1486	1877	1898	1863	2022	1941	2016	1860	1880	1998
2 Phasen	1598	1595	2003	2031	1995	2211	2096	2179	2007	2045	2179
3 Phasen	1638	1639	2049	2081	2050	2296	2162	2265	2085	2122	2265
4 Phasen	1685	1686	2100	2138	2115	2364	2230	2336	2159	2203	2350
5 Phasen	1728	1732	2144	2192	2184	2440	2303	2413	2235	2283	2440
6 Phasen	1767	1770	2193	2242	2240	2525	2382	2487	2311	2364	2528
7 Phasen	1869	1810	2239	2317	2344	2643	2527	2596	2454	2482	2648
8 Phasen	1912	1922	2352	2376	2413	2748	2620	2682	2538	2577	2745
9 Phasen	1945	1978	2402	2426	2474	2827	2689	2752	2615	2666	2841
10 Phasen	1985	2016	2450	2481	2533	2904	2754	2832	2693	2744	2930
One	1537	1756	2428	2127	2272	2295	2674	2673	2393	2456	2600
Multi	1847	2076	2470	2545	2886	2699	2927	3018	2679	2794	2887

Tab. A.1 Messwerte der 100 Mbit-Messreihe

Tab. A.2 Messwerte der 10 Mbit-Messreihe

	0k	**100k**	**200k**	**300k**	**400k**	**500k**	**600k**	**700k**	**800k**	**900k**	**1000k**
1 Phase	2187	2323	2468	2404	2476	2475	2413	2545	2434	2412	2471
2 Phasen	2342	2473	2639	2609	2686	2696	2667	2803	2692	2684	2770
3 Phasen	2413	2543	2722	2709	2799	2851	2815	2975	2874	2881	2977
4 Phasen	2478	2605	2802	2807	2916	2992	2976	3144	3049	3080	3187
5 Phasen	2533	2669	2893	2912	3032	3127	3128	3312	3227	3280	3391
6 Phasen	2598	2728	2970	3017	3157	3267	3277	3478	3411	3475	3602
7 Phasen	2676	2804	3063	3119	3289	3406	3427	3645	3588	3673	3815
8 Phasen	2743	2891	3155	3231	3414	3562	3588	3833	3782	3885	4035
9 Phasen	2803	2952	3233	3328	3532	3691	3733	3995	4004	4074	4244
10 Phasen	2874	3014	3313	3425	3652	3830	3882	4164	4185	4295	4457
One	2196	2443	2628	2751	2795	2933	3006	3120	3270	3380	3493
Multi	2711	2964	3160	3320	3407	3582	3628	3905	3926	3978	4122

Tab. A.3 Messwerte der ISDN-Messreihe

	0k	10k	20k	30k	40k	50k	60k	70k	80k	90k
1 Phase	12350	13895	13299	11701	11751	11876	11988	12197	12130	12325
2 Phasen	13722	15544	15000	13438	13527	13782	14035	14389	14404	14726
3 Phasen	14079	16072	15616	14178	14419	14810	15222	15697	15863	16213
4 Phasen	14445	16596	16224	14923	15306	15851	16402	17024	17331	17698
5 Phasen	14797	17057	16857	15661	16184	16877	17579	18325	18775	19169
6 Phasen	15155	17608	17477	16415	17080	17894	18739	19623	20214	20645
7 Phasen	15516	18082	18130	17189	17981	18920	19899	20936	21656	22120
8 Phasen	15880	18581	18746	17962	18863	19946	21105	22247	23116	23629
9 Phasen	16225	20044	19353	18699	19746	20974	22276	23557	24565	25105
10 Phasen	16584	20522	20001	19440	20624	22002	23434	24863	26012	26597
One	11367	12676	14684	14729	17188	17519	18993	20189	21938	22973
Multi	13871	16637	17025	17759	21125	20595	21844	23043	24436	25691

Tab. A.4 Messwerte der Modemmessreihe

	0k	**10k**	**20k**	**30k**	**40k**	**50k**	**60k**	**70k**	**80k**	**90k**	**100k**
1 Phase	15438	14808	14684	15143	15404	15241	15949	17322	15785	16606	16207
2 Phasen	16763	16224	16245	16949	17275	17257	18181	19642	18296	19302	19014
3 Phasen	17463	16959	17129	17984	18575	18692	19790	21411	20251	21432	21144
4 Phasen	18058	17670	17985	19025	19780	20087	21380	23156	22121	23397	23294
5 Phasen	18677	18430	18834	20109	21000	21472	22981	24941	23946	25361	25434
6 Phasen	19252	19139	19684	21129	22215	22857	24590	26686	25810	27317	27584
7 Phasen	19872	19884	20539	22154	23420	24247	26160	28446	27680	29282	29729
8 Phasen	20517	20629	21394	23169	24625	25627	27750	30211	29540	31236	31869
9 Phasen	21112	21379	22244	24214	25825	27052	29355	31956	31405	33196	34004
10 Phasen	22042	22104	23104	25249	27030	28437	30925	33721	33220	35147	36198
One	14701	16354	18279	19838	22881	23646	25572	27537	29357	31353	33043
Multi	18965	21496	22266	24908	26162	27723	29895	30740	32383	33768	35708

Literaturverzeichnis

[Lan98] Danny B. Lange: “Mobile Objects and Mobile Agents: The Future of Distributed Computing?”, The European Conference on Object-Oriented Programming ’98, 1998.

[GH99] M. Geier, F. J. Hauck: “FOAM: Fragmented Objects for the Implementation of Mobile Agents”, Tech. Report TR-I4-99-04, IMMD IV, Univ. Erlangen-Nürnberg, 1999.

[BRJ99] G. Booch, J. Rumbaugh, I. Jacobson: “Das UML-Benutzerhandbuch”,Addison-Wesley-Longman, 1999.

[OBJ99] ObjectSpace: “Voyager ORB 3.0 Developer Guide”, http://www.objectspace.com , 1999.

[Bog99] M. Boger: “Java in verteilten Systemen: Nebenläufigkeit, Verteilung, Persistenz”, dpunkt-Verlag, 1999.

[GCK+96] R. Gray, G. Cybenko, D. Kotz, D. Rus: ”Mobile Agents: Explanations and Examples”, Manning Publishing, 1997.

[AD99] Ad Astra: “Jumping Beans White Paper”, http://www.JumpingBeans.com, 1999.

[KG99] D. Kotz, B. Gray: “Mobile Agents and the Future of the Internet”, ACM Operating Systems Review, No. 33(3), 1999.

[HBG+98] M. Geier, M. Steckermeier, U. Becker, F. J. Hauck, E. Meier, U. Rastofer: “Support for mobility and replication in the AspectIX architecture”, The European Conference on Object-Oriented Programming ’98, 1998.

[GHJ+96] E. Gamma, R. Helm, R. Johnson, John Vlissides: “Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software”,Addison-Wesley, 1996.

[Fla98] D. Flanagan:“Java in a Nutshell”, O’Reilly, 1998.

[Lea97] L. Doug: “Concurrent programming in Java: Entwurfsprinzipien und Muster”, Addison-Wesley-Longman, 1997.

[DNM+99] P. Dasgupta, N. Narasimhan, Louise E. Moser, P.M. Melliar-Smith : “MAgNET: Mobile Agents for Networked Electronic Trading”, IEEE Transactions on Knowledge and Data Engineering, Vol. 11, No. 4, 1999.

[GJS98] J. Gosling, B. Joy, G. Steele: “Java Language Specification“, http://java.sun.com/docs/books/jls/html/index.html, 1998.

Abbildungsverzeichnis

Tabellenverzeichnis