RepliXa - Eine Framework zur transparenten und anpassbaren Objektreplikation

Markus Jäger[1]
Erwin-Rommel-Str. 51
91058 Erlangen
Tel.: 09131/302301

26. April 1999

[1] msjaeger@cip.informatik.uni-erlangen.de

Zusammenfassung

Replikation ist eine wichtige Technik, um die Fehlertoleranz, Zuverlässigkeit, Verfügbarkeit und Performance eines verteilten Systems zu verbessern. Da in den letzten Jahren bei der Entwicklung verteilter Systeme ein eindeutiger Trend in Richtung objektorientierter Ansätze zu erkennen ist, sollten die Replikationstechniken diesen neuen Konzepten angepasst werden. Damit beschäftigt sich das Themengebiet der Objektreplikation.

Diese Studienarbeit beschreibt die Implementation eines Frameworks, mit dessen Hilfe Objekte transparent, flexibel und einfach repliziert werden können. Ein Ziel bei der Entwicklung des Frameworks war, daß der Programmierer seine Anwendung auf einem nicht verteilten System entwickeln kann und sie anschließend durch die Verwendung von RepliXa, wozu nur wenige Schritte nötig sind, einer verteilten Umgebung anpassen kann.

Um diese oben beschriebene Flexibilität und Anpassbarkeit zu erreichen, wurde das Konzept der Metaprogrammierung verwendet. Mit Hilfe der Metaprogrammierung kann Programmcode, der zur Anwendung gehört, von Programmcode, der für die Replikation nötig ist, getrennt werden, wodurch eine flexible Kombination von Objekte und Replikationsstrategien ermöglicht wird.

Das Framework stellt einen RMI-Mechanismus, eine aktive Replikationsstrategie und eine passive Replikationsstrategie zur Verfügung. Der RMI-Mechanismus kann verwendet werden, um an entfernten Objekten Methoden aufzurufen. Die aktive Replikationsstrategie eignet sich gut, um Objekte welche eine graphische Darstellung besitzen zu replizieren, und die passive Replikationsstrategie kann gut für kritische Anwendungen verwendet werden, da sie den Ausfall eines beliebigen Replikates toleriert.

Da aus zeitlichen Gründen nicht alle in dieser Studienarbeit beschriebenen Mechanismen implementiert werden konnten, wie zum Beispiel eine Garbage Collection für replizierte Objekte, ist das Framework noch nicht für den praktischen Einsatz geeignet und sollte deshalb nur im Rahmen wissenschaftlicher Arbeiten Verwendung finden.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Zu den wesentlichen Eigenschaften jedes technischen Systems gehören unter anderem seine Fehlertoleranz, Zuverlässigkeit, Verfügbarkeit und Performance. Eine in den letzten Jahren immer beliebter werdende Technik, mit deren Hilfe diese Eigenschaften verbessert werden sollen, stellt die Replikation dar. Die Idee der Replikation ist, kritische Systemkomponenten nicht nur einmal, sondern mehrfach im System zu integrieren. Der Grund für den vermehrten Einsatz der Replikation in den letzten Jahren liegt unter anderem darin, daß die Systeme immer schneller und leistungsfähiger werden. Denn dadurch wird es möglich, den Aufwand zur Koordinierung der Replikate des mehrfach vorhandenen Teilsystems zusätzlich zu bewältigen.

Im Rahmen dieser Studienarbeit wird nun untersucht, inwieweit sich das Konzept der Replikation mit dem Konzept der objektorientierten Programmierung verbinden läßt. Bei der objektorientierten Programmierung handelt es sich um ein in den letzten Jahren sehr populär gewordenes, programmiersprachliches Konzept, das sich hervorragend dazu eignet, komplexe Software-Systeme zu entwerfen. Einen wichtigen Teil dieser Untersuchung stellt die Frage dar, wie die hier entwickelten Mechanismen dem Anwendungsprogrammierer einfach und flexibel zur Verfügung gestellt werden können. Dem Programmierer, welcher eine Anwendung entwirft, sollte so eine einfache Möglichkeit geboten werden, Teile seiner Anwendung zu replizieren.

Im Folgenden werden zunächst die Konzepte Replikation und objektorientierte Programmierung näher betrachtet. Im Anschluß daran erfolgt eine Betrachtung der Möglichkeiten und Chancen, aber auch der Probleme, die sich durch eine Kombination der beiden Konzepte ergeben. Abschließend folgt noch ein Überblick über die restlichen Kapitel dieser Studienarbeit.

1.1 Replikation

Replikation zeichnet sich dadurch aus, daß einzelne Systemkomponenten nicht nur einmal, sondern mehrfach vorhanden sind. Durch diese Vorgehensweise soll eine Verbesserung der Eigenschaften des Gesamtsystems erreicht werden. Falls es sich um aktive Komponenten handelt, wie zum Beispiel Prozesse oder elektronische Schaltkreise, wird von aktiver Replikation gesprochen, ansonsten von passiver. Passiv repliziert werden Daten oder ähnliches.

1.1.1 Aktive Replikation

Abgesehen von der aktiven Objektreplikation, welche weiter unten behandelt wird, ist die aktive Replikation eine verbreitete Technik zur Realisierung von Fehlertoleranz[1]. Dies geschieht meist unter Verwendung physikalischer Redundanz. Sie ist beispielsweise in der Biologie (Säugetiere haben zwei Augen, zwei Ohren, zwei Lungenflügel usw.) zu finden. [Tan95].

Die Fehlertoleranz wird auf folgende Weise erreicht: Nach Ausfall einer Komponente werden ihre Aufgaben von den anderen mehrfach vorhanden Komponenten übernommen, wodurch ein Fehler des Gesamtsystems verhindert wird. Ein Problem ergibt sich jedoch, falls eine Komponente nicht komplett ausfällt, sondern nur falsche Ergebnisse liefert, da in diesem Fall durch evtl. aufwendige Entscheidungsprozesse festgestellt werden muß, welches Ergebnis das richtige ist [Tan95].

In dem für diese Studienarbeit relevanten Umfeld verbirgt sich hinter den aktiven Komponente im allgemeinen ein Prozeß oder ähnliches. Da die replizierten Prozesse alle die gleichen Eingabedaten erhalten und den gleichen Algorithmus ausführen, welcher als deterministisch vorausgesetzt wird, gelangen alle fehlerfreien zum gleichen Ergebnis. Dadurch können, wie oben beschrieben, fehlerhafte Prozesse toleriert werden. Ein Problem ergibt sich jedoch in Zusammenhang mit Softwarefehlern, d.h falls ein Fehler bei der Konzeption oder der Implementierung des Algorithmus gemacht wurde, kommen alle Prozesse zum gleichen falschen Ergebnis. Dieser Sachverhalt stellt einen großen Nachteil der aktiven Replikation dar, da es keinen Schutz gegen Softwarefehler gibt und sie nie vollständig ausgeschlossen werden können. Siehe hierzu auch [Bir96].

Eine Lösung diese Problems wäre, daß für jede einzelne Komponente ein eigener Algorithmus von einen separaten Entwicklerteam konzipiert und implementiert wird. Diese Vorgehensweise hat zur Folge, daß ein Fehler in der Software immer nur in einer Komponente auftritt. Da dies jedoch ein sehr kostspieliger Ansatz ist, ist er nur für sehr kritische Anwendungen (z.B. Raumfahrt) zu empfehlen.

1.1.2 Passive Replikation

Wie bereits erwähnt, ist die Hauptanwendung der passiven Replikation das Replizieren von Daten. Die Vorgehensweise ist wie folgt: Zuerst werden von den Daten Kopien angelegt, im Folgenden als Replikate bezeichnet. Diese Replikate müssen dann alle in einem konsistenten Zustand gehalten werden, d.h. falls die Daten eines Replikates geändert werden, müssen die Daten der anderen Replikate aktualisiert werden. Die einzelnen Replikationsverfahren unterscheiden sich dadurch, wie und wann die Daten der Replikate aktualisiert werden.

Zwei wichtige Bereiche, in denen passive Replikation verwendet wird, stellen Dateisysteme und *Distributed Shared Memory* (DSM) Systeme dar.

[1] Eigenschaft eines Systems, auch dann noch korrekt zu arbeiten, wenn einige seiner Komponenten fehlerhaft sind [Dud93].

1.1.2.1 Dateisysteme

Replizierte Dateisysteme werden im Allgemeinen im Rahmen eines Client-Server-Modells[2] verwendet, wobei die Server den Zugriff auf das Dateisystem anbieten und die Clients darauf zugreifen.

Diese Systeme werden aus unterschiedlichen Gründen verwendet. Zum einen erhöht sich die Fehlertoleranz, da durch die mehrfache Speicherung der Daten diese bei Ausfall nur eines Servers nicht verloren gehen. Zum anderen erhöht sich auch die Verfügbarkeit[3], da die Wahrscheinlichkeit, daß die Daten abrufbar sind, mit der Anzahl der Kopien im System steigt. Des weiteren kann, durch die Existenz mehrerer Server, eine Verteilung der Anfragelast (verursacht durch die Clients) und somit eine Verbesserung der Performance[4] erreicht werden.

Unter den Dateisystemen gibt es eher optimistisch vorgehende, wie zum Beispiel Coda [KS92], bei dem es erlaubt ist, sobald auch nur ein Server erreichbar ist, die Daten von diesem zu lesen und auf diesem zu modifizieren. Diese Vorgehensweise bringt natürlich ein hohes Maß an Verfügbarkeit mit sich, hat jedoch auch den Nachteil, daß die Daten auf den anderen Servern erst aktualisiert werden können, wenn diese wieder erreichbar sind. Sie ist als optimistisch zu bezeichnen, da sie erwartet, daß die gleichen Daten nicht auf zwei verschiedenen Servern unterschiedlich modifiziert werden, was zu Inkonsistenzen innerhalb der Replikate führen würde. Dies ist jedoch nicht gewährleistet und falls solche Inkonsistenzen innerhalb der Daten auftreten, müssen sie aufgelöst werden und es muß eine gemeinsame Version der Daten geschaffen werden. Dies ist jedoch in manchen Fällen nicht ohne die Unterstützung des Benutzers möglich, da das replizierte Dateisystem nichts über die inhaltliche Bedeutung der Daten weiß.

Mechanismen wie Replikation sollten jedoch komplett vor dem Benutzer verborgen werden, da sie seine eigentlichen Aufgaben nicht betreffen. Dies ist bei dem oben dargestellten Ansatz nicht vollständig möglich, was somit einen großen Nachteil dieser Strategie darstellt. Um diese Problematik zu umgehen, wurden andere eher pessimistische Strategien entworfen, die jedoch nicht in der Lage sind ein so hohes Maß an Verfügbarkeit zu erreichen. Ein Beispiel hierfür wäre Harp [LGJ91], bei dem es erst möglich ist, die Daten auf den Servern zu modifizieren, falls mit Sicherheit ausgeschlossen werden kann, daß Inkonsistenzen entstehen.

1.1.2.2 DSM-Systeme

DSM-Systeme werden aus mehreren Gründen verwendet: Zum einen soll damit ein gemeinsamer Speicher, wie er bei Multiprozessoren vorhanden ist, auch für Multicomputer, denen ein Kommunikationsnetz zugrunde liegt, zur Verfügung stellen werden, um darin Daten zu speichern. Ein anderer Grund ist, daß über einen gemeinsamen Speicherbereich Kommunikation zwischen den Prozessen möglich ist. Dies hat den Vorteil, daß zur Kommunikation keine separaten Mechanismen, wie beispielsweise *Message Passing*[5] benötigt werden, sondern bereits

[2] Bei dem Client-Server-Modell bietet der Server einen Dienst an, den der Client in Anspruch nehmen kann, und liefert gegebenenfalls ein Ergebnis zurück[Per93]. Da es sich um ein sehr allgemeines Konzept handelt kann es in den verschiedensten Variationen verwendet werden. Siehe hierzu [Tan92] [Tan96] [SG98].

[3] Die Verfügbarkeit gibt Auskunft darüber, in welchem Umfang ein System zur Bearbeitung einer gegebenen Aufgabe genutzt werden kann [Dud93].

[4] Unter Performance wird die Leistungsfähigkeit eines Systems verstanden.

[5] Beim Message-Passing-Modell wird die Kommunikation zwischen Prozessen durch Nachrichten die sie sich gegenseitig schicken realisiert.

bekanntes verwenden kann [SS94] [Tan95].

Eine der zentralen Fragen im Zusammenhang mit verteiltem Speicher ist, wie groß die Seiten, welche verteilt werden sollen, zu wählen sind. Bei der Verwendung von großen Verteilungseinheiten (normalerweise ein Vielfaches der Seitengröße der computereigenen Speicherverwaltung) besteht der Vorteil, daß das DSM-System besser in das Gesamtsystem integriert werden kann [Tan95]. Außerdem liegt Speicherzugriffen eine gewisse Lokalität zugrunde, d.h. zwei aufeinanderfolgende Zugriffe erfolgen meist in der selben Speicherseite, welche ja bereits zur Verfügung steht.

Große Verteilungseinheiten brauchen jedoch auch länger bis sie übertragen sind, und außerdem erhöht sich durch sie auch die Wahrscheinlichkeit des *False Sharing*s[6]. Um dieses Problem zu umgehen, wählen manche DSM-Systeme kleinere Verteilungseinheiten, wie zum Beispiel Objekte oder Variablen.

Ein weiteres Problem bei DSM-Systemen ist die Verzögerung durch die Netzzugriffe bei der Aktualisierung der Daten und die durch das Gesamtsystem erzeugte Netzlast. Durch die Verwendung eines flexibleren Konsistenzmodells[7] können diese Probleme jedoch abgeschwächt werden [Mos93].

1.2 Objektorientierte Programmierung

Objektorientierte Programmierung ist ein in den letzten Jahren immer populärer werdendes programmiersprachliches Konzept, bei dem ein Softwaresystem als Sammlung kooperierender Objekte modelliert wird. Ein Objekt ist eine vom Programmierer definierte gekapselte Datenstruktur. Es besteht aus internen Daten, welche den Zustand des Objektes repräsentieren, und Methoden bzw. Operationen. Methoden können den internen Zustand verändern, den Zustand (teilweise) zurückgeben oder irgendeine andere Operation ausführen. Eine direkter Zugriff auf den internen Zustand ist nicht erlaubt, was auch als Information Hiding, also das Verbergen von Informationen, bezeichnet wird. Die einzelnen Objekte des Softwaresystems kooperieren, indem sie aneinander Methoden aufrufen. Siehe hierzu auch [Tan95] [Boo93] [Kle98].

In den letzten Jahren hat die Verwendung von Objekten bei der Entwicklung von Software immer mehr zugenommen. Einer der Hauptgründe dafür ist wohl, daß sich komplexe Programme mit Hilfe von Objekten sehr viel einfacher strukturieren und somit beherrschen lassen. Des Weiteren eignet sich der objektorientierte Ansatz besonders gut, um den Problembereich zu modellieren. Es ist also beispielsweise möglich, den Objekten die Namen der real existierenden Objekte, deren Aufgabe sie erfüllen, zuzuweisen. Diese und ähnliche Vorgehensweisen führen zu einem wesentlich besseren Verständnis des Problembereichs. Es existiert natürlich noch eine Menge weiterer Gründe, zu deren Erläuterung jedoch auf [Boo93] [Kle98] [Lar98] verwiesen werden soll.

[6]Mit *False Sharing* wird der ungünstige Fall bezeichnet, daß eine Seite zwei gemeinsam genutzte Variable A und B enthält, welche von zwei Prozessoren verwendet werden, wobei der eine Prozessor nur A verwendet und der andere nur B verwendet. In diesem Fall wird nachdem eine Variable von einem Prozeß verändert wurde, diese beim anderen Prozeß aktualisiert, obwohl dieser sie nicht benötigt. Diese ständigen unnötigen Aktualisierungen führen zu einer Verschlechterung der Performance. Siehe hierzu auch [SS94] und [Tan95].

[7]Das Konsistenzmodell legt fest, zu welchem Zeitpunkt Änderungen an den Daten für anderen Prozesse sichtbar sein müssen.

1.3 Objektreplikation

Objektreplikation stellt die Verbindung der Konzepte Replikation und objektorientierte Programmierung dar, d.h. bei den Einheiten, welche repliziert werden, handelt es sich um programmiersprachliche Objekte. Die Verbindung der beiden Konzepte ist sinnvoll, da die Kapselung der Daten in Objekten auch im Bereich der Replikation viele Vorteile mit sich bringt. Die Objektreplikation kann, wie die herkömmliche Replikation, zur Fehlertoleranz, zur Erhöhung der Zuverlässigkeit und Verfügbarkeit und zu vielem mehr eingesetzt werden.

1.3.1 Passive Replikation

Bei der passiven Replikation wird der Objektzustand als gemeinsamer Speicherbereich verschiedener Prozessoren bzw. Prozesse betrachtet. Durch diese Sichtweise ist es möglich, bereits aus dem DSM-Bereich bekannte Strategien und Konsistenzmodelle zu verwenden. Der Objektzustand umfaßt jedoch für gewöhnlich wesentlich weniger Speicher, als bei herkömmlichen DSM-Systemen als Verteilungseinheit gewählt wird.

Einige Nachteile herkömmlicher DSM-Systemen treten bei der Verwendung von Objekten nicht auf. So verkürzt sich durch den kleineren Speicherbereich, der bei der Replikation übertragen wird, die Übertragungszeit, was zu einer Verbesserung der Performance des Gesamtsystems führen kann [Tan95]. Die Wahrscheinlichkeit des *False Sharing*s verringert sich dadurch ebenfalls. Dies hat zwei Gründe, der eine liegt in der Verwendung von sehr kleinen Verteilungseinheiten, der andere Grund besteht darin, daß die Daten innerhalb eines Objektes einen semantischen Zusammenhang haben, d.h. entweder ein Prozeß braucht den Objektzustand oder er braucht ihn nicht. Es ist also sehr unwahrscheinlich, daß ein Prozeß ausschließlich eine Hälfte eines Objektzustandes und ein anderer ausschließlich die andere benötigt [Bir96].

Der Vorteil der Lokalität, der bei DSM-Systemen ja gerade durch den großen Speicherbereich gegeben war, geht bei der Verwendung von Objekten nicht verloren, sondern verstärkt sich sogar noch. Dies ist der Fall, da die Daten innerhalb eines Objektes, wie bereits festgestellt, einen starken semantischen Zusammenhang haben. Es ist also sehr wahrscheinlich, daß, falls ein Teil des Objektzustandes benötigt wird, der andere ebenfalls verwendet wird [Tan95].

Im Gegensatz zu DSM-Systemen ist es bei der Objektreplikation leider nicht so einfach möglich, diese in die rechnereigene Speicherverwaltung zu integrieren. Dies liegt daran, daß die Größe der Objektzustände weder mit der Größe einer Speicherseite übereinstimmt, noch bei allen Objekten gleich ist. Eine relativ einfache, aber leider äußerst ineffiziente Methode besteht darin, auf eine physikalische Speicherseite immer nur ein Objekt zu plazieren [Tan95]. Eine andere Vorgehensweise wäre, daß der zur Replikation der Daten nötige Programmcode einfach in den Methoden des Objektes integriert wird. Dies sollte jedoch aus mehreren Gründen vermieden werden. Der wichtigste ist wohl, daß in einer Methode der Programmcode zur Manipulation der Daten stehen sollte, also der Programmcode, der in direktem Zusammenhang mit der späteren Arbeitsweise des Programmes steht. Der Programmcode für die Replikation hat damit aber nichts zu tun, er ist lediglich für die Verteilung der Anwendung auf verschiedene Rechnersysteme relevant, was die Arbeitsweise und Aufgaben der Anwendung nicht betrifft. Ein weiterer Grund ist, daß es bei diesem Ansatz sehr schwierig ist, für bereits existierende Objekte eine neue Replikationsstrategie hinzuzufügen oder eine bereits bestehende zu ändern [Gar98].

Eine andere vielversprechendere Lösung ist es, dem Programmierer einen einfachen Mechanismus zur Verfügung zu stellen, mit dem er für die Objekte seiner Anwendung Strategien bezüglich ihres Zugriffs, und somit ihrer Verwendung, festlegen kann. Es ist dann auch möglich, die Anwendung auf einem einzigen Rechner zu entwerfen, und erst im Nachhinein durch ändern der Strategie für bestimmte Objekte auf mehrere aufzuteilen. Das oben beschriebene Ändern oder Hinzufügen einer Strategie zu Objekten ist dann auch problemlos möglich. Um eine derartige Entkoppelung der Replikationsstrategie von den Anwendungsobjekten zu erreichen, ist allerdings die Unterstützung des Laufzeitsystems[8] der Programmiersprache nötig (z.B. für das Abfangen des Zugriffs auf die Daten) [Gar98].

1.3.2 Aktive Replikation

Das Ziel der aktiven Objektreplikation ist das gleiche wie bei der passiven, nämlich den internen Objektzustand der verschiedenen Replikate in einem konsistenten Zustand zu halten. Es werden hierbei jedoch völlig andere Mechanismen verwendet. Es werden nicht die Daten der Objekte repliziert, sondern die Methodenaufrufe an den Objekten. Dies ist ausreichend, da alle Modifikationen an den Daten ohnehin über Methoden erfolgen. Die Replikation der Methodenaufrufe wird folgendermaßen realisiert: Falls eine Methode an einem Objekt aufgerufen wird, wird diese abgefangen und automatisch an allen Replikaten aufgerufen [KG96].

So einfach diese Idee ist, soviele Probleme bringt sie auch mit sich. Da auf die aktive Replikation und ihre Probleme im weiteren Verlauf noch des öfteren eingegangen wird, folgt an dieser Stelle nur ein kurzes Beispiel: Die Methoden müssen alle deterministisch ablaufen und an jedem Objekt in der gleichen Reihenfolge aufgerufen werden, um Inkonsistenzen zu vermeiden [Gar98].

Eine Möglichkeit, dem Anwendungsprogrammierer einen einfachen und flexiblen Einsatz dieser Art der Replikation zu ermöglichen, besteht darin, wie bei der letzten Methode der passiven Replikation vorzugehen, da dadurch eine saubere Trennung von Anwendungsobjekten und Replikationsstrategien erreicht werden kann. Es ist dann allerdings auch wieder die Unterstützung des Laufzeitsystems nötig (z.B. für das Abfangen der Methodenaufrufe).

1.4 Kapitelübersicht

Nachdem jetzt ein umfangreicher Überblick über den Themenkreis der Replikation gegeben wurde, wird im nächsten Kapitel näher auf die Ziele und Rahmenbedingungen dieser Studienarbeit eingegangen. Im dritten Kapitel wird das Grundsystem erläutert, welches die Aufgabe hat, Objekte mit Replikationsstrategien zu verbinden. Im vierten und fünften Kapitel werden die zur Verfügung gestellte aktive und passive Replikationsstrategie vorgestellt. In Kapitel sechs

[8] Höhere Programmiersprachen bieten zur Programmierung häufig komplexe Ausdrucksmittel an, welche nicht direkt in eine kurze Befehlsfolge der Maschinensprache übersetzt werden können (Zugriff auf komplexe Datenstrukturen, rekursive Prozeduren), oder die von Werten abhängen, die erst während der Ausführung des Programmes ermittelt werden (z.B. die dynamische Speicherverwaltung). Für die Bearbeitung solcher Operationen werden Prozeduren in der Maschinensprache geschrieben, welche dann vom übersetzten Programm aufgerufen werden. Die Zusammenfassung solcher Prozeduren für einen Übersetzer werden als Laufzeitsystem bezeichnet. Siehe [Dud93].

erfolgen einige Schlußbetrachtungen und im Anhang sind diverse Beispielprogramme zur Verwendung der Mechanismen zu finden.

Kapitel 2

Ziele und Rahmenbedingungen

In diesem Kapitel werden zuerst die bei der Implementierung der im Rahmen dieser Studienarbeit entwickelten RepliXa Bibliothek angestrebten Ziele dargelegt. Danach wird erläutert, wie und mit welchen Mitteln, d.h. Rahmenbedingungen wie Programmiersprache und Interpreter, versucht wurde, diese zu erreichen. Zum Schluß erfolgt eine Beschreibung des zugrunde gelegten System- und Fehlermodells.

Zuerst aber noch ein kurzer Exkurs zur Entstehung des Namens RepliXa: Da der Bereich, mit dem sich diese Studienarbeit vornehmlich beschäftigt, die Replikation ist und zur praktischen Realisierung der verschiedenen Replikationsstrategien der Java-Interpreter metaXa Verwendung fand, erschien eine Kombination aus Replikation und metaXa angemessen, also RepliXa.

2.1 Ziele von RepliXa

Hauptziel von RepliXa ist es, dem Anwendungsprogrammierer, im Folgenden als Programmierer bezeichnet, eine einfache, transparente, anpassbare und offene Möglichkeit der Objektreplikation zur Verfügung zu stellen.

Einfachheit
Dem Programmierer sollte eine möglichst einheitliche und kleine Schnittstelle zur Nutzung der Dienste von RepliXa zur Verfügung stehen.

Transparenz
Es sollte möglich sein Objekte zu entwickeln, ohne die Kenntnis darüber, ob sie später repliziert werden oder nicht, d.h. die Replikation sollte in einer für die Objekte transparenten Art erfolgen.

Anpassbarkeit
Die Anpassung einer Applikation an eine neue Ausführungsumgebung sollte nicht zu viele Änderungen erfordern.

Offenheit
Es sollte problemlos möglich sein, neue Replikationsstrategien in RepliXa zu integrieren.

Bei dem Versuch, ein System mit den oben genannten Eigenschaften zu entwerfen, haben sich folgende Teilaufgaben ergeben:

- Am Anfang stand die Frage, wie das selektive Binden von Replikationsstrategien an Objekte realisiert werden könnte. Mit selektivem Binden ist gemeint, daß für verschiedene Objekte festgelegt werden kann, ob und wie sie repliziert werden, wodurch die oben genannte Transparenz und Anpassbarkeit erreicht werden sollte.

- Danach mußte geprüft werden, welche Basismechanismen zur einfacheren Implementierung der einzelnen Replikationsstrategien zur Verfügung gestellt werden sollten. Hier ergab sich im Wesentlichen die Notwendigkeit eines entfernten Methodenaufrufs (*Remote Method Invocation* RMI), um die Kommunikation der einzelnen Replikate untereinander zu vereinfachen. Es wurde schnell klar, daß der Mechanismus des selektiven Bindens auch hier Verwendung finden würde.

- Nun, nachdem ein gewisser Rahmen geschaffen wurde, war es nötig, die verschiedenen Replikationsstrategien, die implementiert werden sollten, genauer festzulegen. Es sollten eine aktive (Kapitel 4) und eine passive (Kapitel 5) Strategie zur Verfügung gestellt werden.

- Zu guter Letzt mußte eine Möglichkeit gefunden werden, die obigen Überlegungen so zu verbinden, daß sich ein offenes System mit einer einfachen Schnittstelle ergibt.

Nachdem diese Teilaufgaben gelöst waren, konnte begonnen werden, zuerst das Grundsystem (Kapitel 3) und dann die Replikationsstrategien zu implementieren.

2.2 Die Programmiersprache Java

Java ist eine C-ähnliche Sprache, die von der US-Firma Sun Microsystems entwickelt wurde und im Gegensatz zu C jedoch objektorientiert ist und interpretiert wird. In [Fla96] ist eine Beschreibung ihrer wesentlichen Eigenschaften zu finden. Hier werden nur ein paar der wichtigsten erläutert:

Einfach
Da sie der Syntax von C++ sehr ähnlich ist, ist es mit etwas Erfahrung in C, C++ oder Pascal nicht schwer, die Grundzüge der Sprache zu erlernen. Zudem wurden viele Elemente von C++, welche diese Sprache unnötig komplizierten und nicht zwingend erforderlich waren, weggelassen.

Objektorientiert
Java ist vollkommen objektorientiert, abgesehen von einfachen Typen wie Integer oder Boolean, und folgt deshalb den üblichen Kriterien objektorientierter Programmiersprachen wie Abstraktion, Kapselung von Daten und Methoden, Vererbung und Polymorphismus.

Interpretiert
Um eine hohe Flexibilität zu erreichen, werden die Java-Klassen zuerst in einen plattformunabhängigen Bytecode übersetzt und anschließend durch einen Java-Interpreter auf der entsprechenden Plattform ausgeführt.

Multithreaded
Im Gegensatz zu den meisten anderen Programmiersprachen verfügt Java über eine eingebaute Sprachunterstützung für Threads. Dadurch sind diese sehr einfach zu erzeugen, zu verwalten und zu koordinieren. Jedes Objekt kann als Monitor verwendet werde, d.h. manche Methoden können nur in gegenseitigem Ausschluß abgearbeitet werden. Das Konzept des Monitors wurde dahingehend erweitert, daß auch Teile von Methoden, die nicht notwendigerweise zur Klasse selbst gehören, mit in den gegenseitigen Ausschluß aufgenommen werden können.

Garbage Collection
Aufgabe der Garbage Collection ist es, Objekte, die nicht mehr benötigt werden, automatisch zu löschen. Dies stellt eine Erleichterung für den Programmierer dar, da dies sonst mit zu seinen Aufgaben zählen würde.

Durch die Verwendung des Java-Interpreters metaXa war die Programmiersprache Java bereits fest vorgegeben. Wenn von Performance-Überlegungen abgesehen wird, entstanden hierdurch jedoch keine Nachteile. Ganz im Gegenteil erwies sich die Unterstützung im Bereich Multithreading als sehr vorteilhaft.

2.3 Der Java-Interpreter metaXa

Zuerst erfolgt eine Beschreibung der Eigenschaften und Möglichkeiten des metaXa-Interpreters, welche im Wesentlichen übernommen wurde aus [KG96] [Gol97] [GK97a] [GK97b] und [GK98]. Im Anschluß daran wird darauf eingegangen, warum und wie sie zur Realisierung von RepliXa verwendet wurden.

2.3.1 Reflexion und Metaprogrammierung

MetaXa ist ein erweiterter Java Interpreter, der Reflexion und Metaprogrammierung in Java ermöglicht. Golm und Kleinöder schreiben hierzu in [GK97b] folgendes:

> „Reflexion ist die Fähigkeit eines Berechnungssystems, ‚Aussagen über sich selbst zu treffen und auf sich selbst einzuwirken‘. Metaprogrammierung ist eine Methode, um Programmteile, die der Lösung des Anwendungsproblems dienen (funktionaler Code) von Programmteilen zu trennen, die Überwachungs und Steuerungsaufgaben wahrnehmen (nicht-funktionaler Code). Die Aufteilung in Basisebene (funktionaler Code) und Metaebene (nicht-funktionaler Code) ermöglicht es, das Ausführungsverhalten von Basisebenen-Objekten in einer für die Basisebenen-Objekten transparenten Art zu kontrollieren. Um diese Kontrolle zu ermöglichen, müssen einige Aspekte der Berechnung reifiziert werden. Unter Reifikation versteht man das Verfügbar machen von Dingen, die normalerweise nicht Bestandteil des Programmiermodells sind. Zum Beispiel ist die Operation eines Methodenaufrufs in den meisten

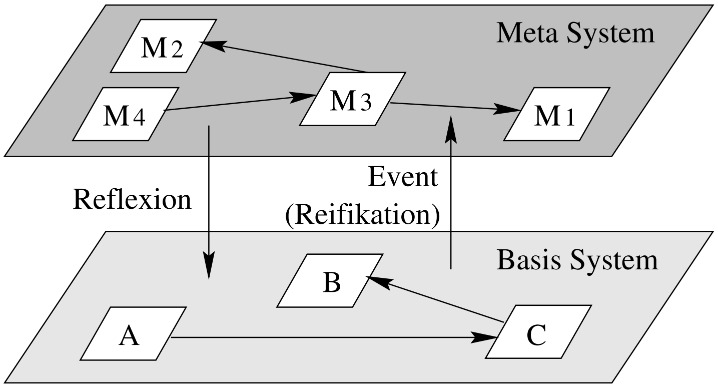


Abbildung 2.1: Zusammenhänge zwischen Reflexion, Reifikation, Basis und Meta System

> objektorientierten Programmiersprachen nur implizit verfügbar und der Mechanismus ‚Methodenaufruf' damit auch nicht änderbar.
>
> Es wird zwischen struktureller Reflexion und Reflexion des Berechnungsvorganges (*behavioral reflection*, Verhaltensreflexion) unterschieden. Strukturelle Reflexion reifiziert strukturelle Aspekte eines Programms, wie Datentypen, Methodennamen oder Vererbungsbeziehungen. Ein Beispiel für strukturelle Reflexion ist das Java 1.1 Reflection API. Verhaltensreflexion reifiziert Operationen des Laufzeitsystems, wie Methodenaufrufe, Variablenzugriffe oder Objektinstantiierungen. MetaXa ist ein System, welches sowohl strukturelle als auch Verhaltensreflexion unterstützt."

Die Zusammenhänge zwischen Reflexion, Reifikation, Basis- und Meta-System sind in Abbildung 2.1 dargestellt.

2.3.2 Binden von Meta-Objekten an Objekte

Die Reflexion und Metaprogrammierung wird in metaXa dadurch erreicht, daß Objekte der Metaebene (Meta-Objekte) an Objekte der Basisebene (Basis-Objekte) gebunden werden. Das Binden kann selektiv, d.h. nur an bestimmte Objekte, und dynamisch, d.h. zur Laufzeit, durchgeführt werden. Es gibt außerdem die Möglichkeit, Meta-Objekte an Referenzen und Klassen zu binden (Siehe hierzu auch Abbildung 2.2). Nachdem das Meta-Objekt an ein Basis-Objekt gebunden wurde, können ausgewählte Mechanismen des Laufzeitsystems umdefiniert werden, wodurch eine spezialisierte Ausführungsumgebung für das Basis-Objekt hergestellt wird. Die Mechanismen des Laufzeitsystems werden umdefiniert, indem das Meta-Objekt dem Laufzeitsystem mitteilt, daß es Interesse an bestimmten Events hat, d.h. es registriert sich für diese Events. Das Meta-Objekt kann dann für diese Events das Standardverhalten ändern und somit in das Geschehen eingreifen.

Nachdem ein Meta-Objekt an ein Basis-Objekt gebunden wurde, stehen folgende Events zur Verfügung:

- Eingehende Methodenaufrufe (am Basis-Objekt wird eine Methode aufgerufen)
- Ausgehende Methodenaufrufe (das Basis-Objekt ruft an einen anderen Objekt eine Methode auf)

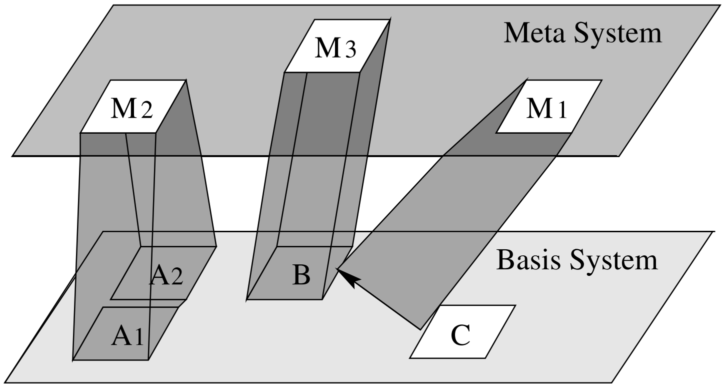


Abbildung 2.2: Meta-Objekte können an Klassen, Objekte und Referenzen gebunden werden.

- Variablen-Zugriff (auf den internen Objektzustand wird zugegriffen)
- Objekt-Erzeugung (ein neues Objekt wird durch eine Methode des Basis-Objektes erzeugt)
- Objekt-Sperrung und Objekt-Freigabe (das Objekt wird zur Synchronisation verwendet)

Während ein Event durch das Meta-Objekt bearbeitet wird, stehen die verschieden Mechanismen der strukturellen Reflexion zur Verfügung. Es ist also in diesem Stadium unter anderem möglich, den Zustand andere Objekte und ihre Beziehungen zueinander zu analysieren.

2.3.3 Grund der Verwendung

Der Grund für die Verwendung von metaXa besteht darin, daß es sich mit seinen zwei Ebenen (Meta- und Basisebene) hervorragend dazu eignet, den zur Replikation nötigen Code klar von der Anwendung zu trennen. Die Anwendung wird wie gehabt entwickelt und danach werden an die Objekte, die repliziert werden sollen, Meta-Objekte, die das Verhalten der Objekte entsprechend ändern, gebunden. Die Meta-Objekte werden von RepliXa zur Verfügung gestellt, oder können, falls spezielle Replikationsstrategien erforderlich sind, parallel zur Anwendung entwickelt werden. Durch diese Vorgehensweise kann die geforderte Einfachheit, Transparenz und Anpassbarkeit von RepliXa erreicht werden. Eine genauere Analyse und Bewertung dieses Sachverhalts erfolgt jedoch erst in Kapitel 6.

2.4 System- und Fehlermodell

Die mit Hilfe von RepliXa kooperierenden Teile einer Anwendung laufen in verschiedenen Prozessen ab und können auf mehrere Rechner verteilt sein. Es wird angenommen, daß die einzelnen Prozesse ein Fail-Stop Verhalten aufweisen, d.h. daß sie im Fehlerfall anhalten und keinen Einfluß mehr auf die restliche Anwendung haben.

Die einzelnen Prozesse sind über ein Kommunikationsnetz miteinander verbunden. Abgesehen von einer zeitweiligen Partitionierung des Kommunikationsnetzes mit der Folge, daß manche Prozesse nicht erreichbar sind, wird das Kommunikationsnetz als zuverlässig angenommen, d.h. alle Nachrichten treffen ohne Fehler in der gesendeten Reihenfolge beim Empfänger ein.

Inwieweit die einzelnen Teile von RepliXa mit einer Partitionierung des Netzwerkes zurecht kommen, wird in den entsprechenden Kapiteln näher erläutert.

Kapitel 3

Das Grundsystem

Die Aufgaben des Grundsystems sind sehr vielfältig. Zum einen sollte es dem Programmierer eine einheitliche Schnittstelle zur Nutzung der Dienste von RepliXa, d.h. des RMI-Mechanismus und der verschiedenen Replikationsstrategien, zur Verfügung stellen. Zum anderen sollte es aber auch gewisse Basisabstraktionen wie Objektübertragung oder RMI liefern, um die Implementierung der zu RepliXa gehörenden Replikationsstrategien zu erleichtern. Diese Abstraktionen kommen natürlich auch dem Programmierer zugute, falls er sich entschließt, eine eigene, seinen Bedürfnissen angepaßte Replikationsstrategie zu entwickeln. Die Integration neuer Strategien muß natürlich auch durch das Grundsystem ermöglicht werden.

Bei der Konzeption des Grundsystems traten verschiedene Probleme auf. Im nächsten Abschnitt erfolgt eine genauere Beschreibung dieser Probleme und ihrer Lösungen. Danach wird das Grundsystem von RepliXa näher erläutert, und zum Schluß erfolgt noch eine Beschreibung möglicher Erweiterungen. Ein Beispielprogramm zur Verwendung der Mechanismen des Grundsystems ist in Anhang A zu finden.

3.1 Aufgetretene Probleme und ihre Lösungen

Die in diesem Abschnitt behandelten Probleme sind sehr unterschiedlicher Natur. Zum Teil handelt es sich um Probleme, die bereits bei der Konzeption des Grundsystems auftraten, wie das selektive Binden, den RMI-Mechanismus, die Objektübertragung und den Zugriff auf neue Objekte, teilweise aber auch um Probleme, die erst während der späteren Realisierung zu lösen waren, wie die Synchronisation und die System-Objekte.

3.1.1 Selektives Binden

Bei den Überlegungen, wie dem Programmierer die Dienste am besten zur Verfügung gestellt werden könnten, wurde klar, daß es für den Programmierer von Vorteil wäre, wenn er seine Anwendung in zwei Phasen erstellen könnte. Zuerst entwickelt der Programmierer die Anwendung in einem nicht verteilten System, was den Vorteil hat, daß er sich auf die Funktionalität der Anwendung konzentrieren kann. Danach passt der Programmierer die Anwendung der verteilten Ausführungsumgebung an, indem er das Verhalten verschiedener Objekte ändert. In dieser Phase muß er die Funktionalität der Anwendung nicht mehr beachten, da es für die Interaktion der Objekte im Allgemeinen ohne Bedeutung ist, wo sie sich befinden, bzw. ob sie repliziert

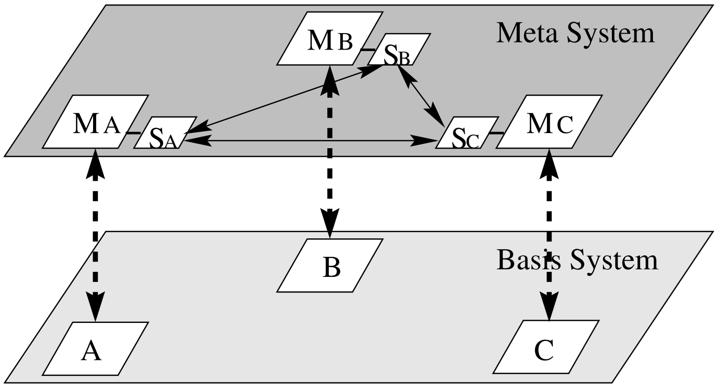


Abbildung 3.1: Zusammenhänge zwischen Objekten, Meta-Objekten und Strategie-Objekten (Die Strategie-Objekte werden in allen weiteren Abbildungen nicht mehr explizit dargestellt, da sie als ein fester Bestandteil der Meta-Objekte angesehen werden können.)

sind. Er kann also beispielsweise kritische Komponenten replizieren, um ihre Zuverlässigkeit zu erhöhen, ohne den vorher implementierten Anwendungscode ändern zu müssen. Natürlich können diese beiden Schritte auch gleichzeitig durchgeführt werden, aber der Programmcode für die Anwendung und zur Anpassung an die Ausführungsumgebung entstehen separat.

Das Problem war nun, eine Möglichkeit zu finden, dem Programmierer diese Vorgehensweise zu ermöglichen. Die hier gewählte Variante beruht auf der Verwendung der Metaebene des metaXa-Interpreters, d.h. an jedes Objekt, dessen Verhalten geändert werden soll, wird ein Meta-Objekt, das das Gewünschte leistet, gebunden. Es sind jedoch auch andere Ansätze denkbar, wie eine direkte Unterstützung des Betriebssystems oder externe Tools, die bei der Übersetzung der Anwendung verwendet werden. Siehe hierzu auch [Gol97] und [KG96].

Da das Binden der Meta-Objekte erst dynamisch zur Laufzeit erfolgt, hat es den Nachteil, daß der Programmcode, der für das Binden nötig ist, doch dem Anwendungscode hinzugefügt werden muß. Der Nachteil ist jedoch nicht besonders gravierend, da es sich im Allgemeinen nur um wenige Zeilen handelt, welche sich schnell wieder entfernen oder neuen Umgebungen anpassen lassen.

Das gebundene Meta-Objekt enthält ein Strategie-Objekt, welches für die eigentliche Verhaltensänderung verantwortlich ist. Zuerst teilt es dem Laufzeitsystem mit, über welche Ereignisse es informiert werden möchte. Bei Auftreten eines solchen Ereignisses kooperiert es dann eventuell mit Strategie-Objekten anderer Objekte, um das Objektverhalten in der gewünschten Weise zu ändern. Die Möglichkeit, diese Strategie-Objekte zu ändern, was bedeuten würde, das Verhalten eines Objektes während einer Ausführung mehrfach zu ändern, wird nicht unterstützt. Der Grund liegt darin, daß Untersuchungen, die in Zusammenhang mit dem Garf-Projekt [GGM96] gemacht wurden, dies als, für heutige Anwendungen, unnötig erachtet haben. Der Zusammenhang zwischen Objekten, Meta-Objekten und Strategie-Objekten ist in Abbildung 3.1 dargestellt.

3.1.2 Remote Method Invocation

Ein RMI-Mechanismus wird nun mit Hilfe des bereits beschriebenen Mechanismus zur Verfügung gestellt. Im Adreßraum des Prozesses, von dem aus der Zugriff auf das Remote-Objekt

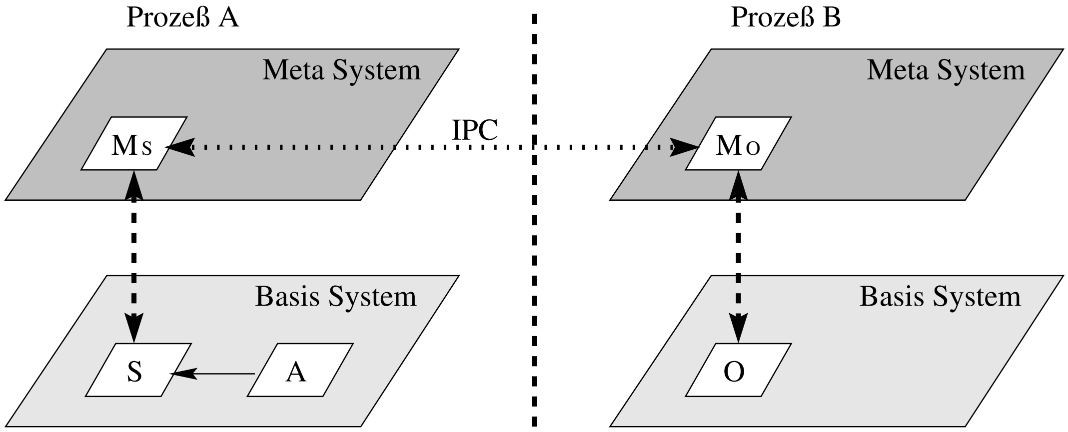


Abbildung 3.2: Ablauf einer *Remote Method Invocation*: Objekt A möchte eine Methode des Objektes O aufrufen. Objekt A ruft die Methode an dem Stub-Objekt S auf. Das Meta-Objekt Ms fängt den Methodenaufruf ab und leitet ihn über IPC-Mechanismen an das Meta-Objekt Mo weiter. Mo ruft daraufhin die entsprechende Methode von O auf. Das Ergebnis des Methodenaufrufs wird auf dem gleichen Weg zurück gegeben.

erfolgen soll, befindet sich ein Objekt gleichen Typs, an das alle Anfragen gerichtet werden. Dieses Objekt bearbeitet die Anfragen jedoch nicht selbst, sondern fungiert lediglich als Stub bzw. Proxy, indem ein daran gebundenes Meta-Objekt alle Anfragen abfängt. Dieses Meta-Objekt kommuniziert dann über IPC-Mechanismen (*inter process communication*) mit dem Meta-Objekt des Remote-Objekts. Das Meta-Objekt des Remote-Objekts führt dann die eigentliche Anfrage an dem Remote-Objekt durch. Das Ergebnis der Anfrage wird auf dem gleichen Weg zurückgegeben. Der Ablauf ist in Abbildung 3.2 dargestellt.

3.1.3 Synchronisation

In Java kann jedes Objekt zur Synchronisation verwendet werden, d.h. es kann von einem Aktivitätsträger exklusiv belegt werden, d.h. für den Zugriff anderer Aktivitätsträger gesperrt werden. Zu diesem Zweck schickt der Aktivitätsträger an das Objekt eine Sperr-Anforderung, um das Objekt zu sperren, und eine Freigabe-Anforderung, um das Objekt wieder frei zu geben. Falls es sich um ein Remote-Objekt handelt, sollte natürlich nicht der Stub gesperrt werden, sondern das Remote-Objekt selbst. Hierzu kann das gleiche Verfahren wie in Abschnitt 3.1.2 verwendet werden.

Ein Aktivitätsträger wird in Java durch einen Thread realisiert. Beim Übergang eines Aktivitätsträger von einem Prozeß in einen anderen (durch eine RMI) muß im neuen Prozeß für diesen Aktivitätsträger ein neuer Thread geschaffen werden.

Alle Threads innerhalb eines Prozesses haben eine eindeutige Identifikation. Diese Identifikation wird von dem Java eigenen Sperr-Mechanismus verwendet um festzustellen, welcher Thread ein Objekt gesperrt hat. Falls ein Thread also versucht, ein bereits von ihm gesperrtes Objekt noch einmal zu sperren, kann dies festgestellt und zugelassen werden.

Bei der Erweiterung dieses Mechanismus, so daß er mit Remote-Objekten verwendet werden kann, ergab sich bei folgendem Ablauf ein Problem:

1. Der zum Aktivitätsträger A gehörende Thread T1 belegt von Prozeß P1 aus ein Remote-Objekt R.

2. A wechselt mit Hilfe einer RMI von P1 zu Prozeß P2, woraufhin in P2 ein neuer Thread T2 für A erzeugt wird.

3. Der nun neu zu A gehörende Thread T2 versucht von P2 aus R erneut zu belegen.

Das Problem, das sich hier nun ergibt, ist: Das erneute Belegen des Remote-Objektes sollte zugelassen werden, da es sich um den gleichen Aktivitätsträger handelt. Der Java eigene Sperr-Mechanismus kann aber nicht erkennen, daß es sich bei T1 und T2 um den gleichen Aktivitätsträger handelt, da sie verschiedene Identifikationen besitzen.

Die bei RepliXa gewählte Lösung dieses Problems besteht aus zwei Schritten. In einem ersten Schritt wird für jeden Aktivitätsträger eine eindeutige Identifikation generiert, die über mehrere Threads gleich bleibt. Im zweiten Schritt wird ein eigener Sperr-Mechanismus entworfen, der mit Aktivitätsträger-Identifikationen statt mit Thread-Identifikationen arbeitet.

3.1.4 Objektübertragung

Eine weiteres Problem ergab sich durch die Frage, wie Objekte übertragen werden sollten, falls diese bei einer RMI als Parameter oder Rückgabewert angegeben werden. Das Objekt kann

- als Kopie übertragen werden
- zu einem Remote-Objekt gemacht werden und als Stub übertragen werden
- als Replikat übertragen werden
- als System-Objekt übertragen werden (siehe nächster Abschnitt).

Um hier eine eindeutige und doch flexible Lösung zu erhalten, wurde folgende Vorgehensweise gewählt. Objekte, an die kein Meta-Objekt gebunden ist, werden als Kopien übertragen. Bei allen weiteren Objekten legt das Strategie-Objekt des Meta-Objektes die Art der Übertragung fest.

3.1.5 System-Objekte

Bei Objekten, die von prozeßspezifischen Gegebenheiten abhängig sind, wie zum Beispiel graphische Komponenten, Dateien, Eingabe- und Ausgabe-Mechanismen, ist es nicht immer sinnvoll, sie als Stub oder als Kopie zu übertragen. Es könnte vielmehr sinnvoll sein, auf der Enpfängerseite ein Objekt zu verwenden, das auf der Enpfängerseite die gleiche Aufgabe wie das ursprüngliche Objekt auf der Sendeseite hat. Zu diesem Zweck wurde das Konzept der System-Objekte eingeführt. Diese Objekte werden nicht einfach als Kopie übertragen, sondern existieren entweder auf beiden Seiten bereits und sind zuordenbar, oder werden auf der Empfängerseite neu angelegt.

Bei RepliXa werden System-Objekte bei der ersten Übertragung auf der Empfängerseite neu angelegt. Danach ist es möglich sie eindeutig zuzuordnen, weswegen dann nur noch ihre Kennung übertragen wird.

3.1.6 Zugriff auf neue Objekte

Weiterhin zu klären war die Frage, wie ein Prozeß Zugriff auf neue Objekte erhält, d.h. wie erhält er den Stub, um auf ein Remote-Objekt zuzugreifen, oder wie erhält er ein Replikat eines replizierten Objekts.

Hierzu wurde ein einfacher Nameservice eingeführt. Jeder Prozeß besitzt einen eigenen Nameservice, bei dem er Objekte unter einem symbolischen Name registrieren kann. Auf diesen Nameservice kann dann von jedem anderen Prozeß aus zugegriffen werden. Die anderen Prozesse fordern also bei dem Nameservice einfach das betreffende Objekt an. Da zur Realisierung des Nameservice der bereits bekannte Mechanismus zur Objektübertragung verwendet wird, legen die Meta-Objekte der im Nameservice registrierten Objekt fest, ob die Prozesse eine Kopie, einen Stub, ein System-Objekt oder ein Replikat erhalten.

3.2 Realisierung

In diesem Abschnitt wird die Implementierung des Grundsystems von RepliXa näher betrachtet. Zuerst wird beschrieben, wie das Verhalten von Objekten geändert werden kann. Danach wird der Mechanismus zum Zugriff auf Remote-Objekte näher erläutert, und zum Schluß werden einige Verhaltensstrategien für Objekte und der Nameservice beschrieben.

Die Klassendiagramme in diesem Abschnitt wurden mit Hilfe der *Unified Modelling Language* (UML) modelliert [BRJ99].

3.2.1 Änderung des Objektverhaltens

Wenn der Programmierer das Verhalten eines Objektes ändern möchte, muß er zuerst ein Strategie-Objekt auswählen, das das gewünschte neue Verhalten realisiert. Das Strategie-Objekt legt außerdem fest, wie das Objekt übertragen wird. Eine Beschreibung der einzelnen, von RepliXa zur Verfügung gestellten Strategie-Objekte erfolgt im Laufe dieses Kapitels. Danach muß der Programmierer eine Verbindung zwischen dem Objekt und dem Strategie-Objekt herstellen.

Das Auswählen eines Strategie-Objektes erfolgt dadurch, daß er eine neue Instanz des von ihm gewünschten Strategie-Objektes bildet. Der RepliXa interne Ablauf beschränkt sich darauf, daß die einzelnen Komponenten des Strategie-Objektes initialisiert werden. Damit die Strategie-Objekte von RepliXa unter polymorphen Gesichtspunkten gleich behandelt werden können, müssen sie alle Unterklassen der abstrakten Klasse ACCESSTRANSMITSTRATEGY sein.

Die Verbindung zwischen dem Objekt und dem Strategie-Objekt erfolgt dadurch, daß ein Meta-Objekt (eine Instanz der Klasse ACCESSCONTROLOBJECT) an das Objekt gebunden wird. Da das Binden während der Konstruktion des neuen Meta-Objektes erfolgt, muß bei der Konstruktion bereits das zu überwachende Objekt und das neue Strategie-Objekt angegeben werden. RepliXa intern geht bei der Konstruktion folgendes vor:

1. Das Meta-Objekt bindet sich an das Objekt.
2. Das Meta-Objekt frägt beim Strategie-Objekt nach, an welchen Events des Basis-Objektes es Interesse hat, und leitet es an das Laufzeitsystem des metaXa-Interpreters weiter.

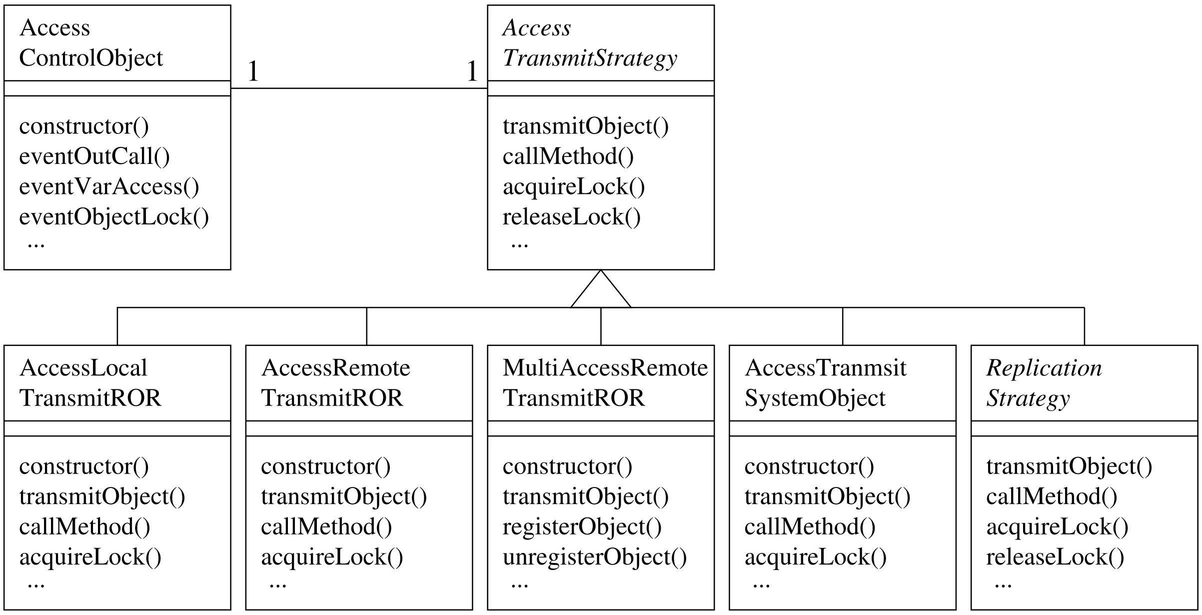


Abbildung 3.3: Zusammenhang zwischen dem Meta-Objekt ACCESSCONTROLOBJECT und den einzelnen Strategie-Objekten

3. Das Strategie-Objekt wird vom Meta-Objekt mit Informationen versorgt, die bei seiner Instantiierung nicht verfügbar waren.

Von nun an werden die am Basis-Objekt auftretenden Events vom Laufzeitsystem dem Meta-Objekt gemeldet. Dieses gibt sie an das Strategie-Objekt weiter. Das Strategie-Objekt reagiert darauf, um die gewünschte Verhaltensänderung zu erreichen.

Der Zusammenhang zwischen dem Meta-Objekt ACCESSCONTROLOBJECT und den einzelnen Strategien ist in Abbildung 3.3 dargestellt.

3.2.2 Zugriff auf Remote-Objekte

Der Zugriff auf Remote-Objekte wird im Wesentlichen durch die beiden Strategie-Objekte ACCESSLOCALTRANSMITROR und ACCESSREMOTETRANSMITROR realisiert, welche im Anschluß beschrieben werden. Danach wird der Ablauf eines Zugriffs dargestellt und die Mechanismen zur Objektübertragung und zum Sperren des Remote-Objektes näher erläutert.

3.2.2.1 Die Strategie AccessLocalTransmitROR

Das Strategie-Objekt ACCESSLOCALTRANSMITROR ist mit dem Remote-Objekt verbunden. Es ändert das Verhalten des Remote-Objektes in seinem eigenen Prozeß nicht. Es werden lediglich die Sperr- und Freigabe-Events abgefangen und angepasst, so daß sie mit der unter Kapitel 3.1.3 geforderten globalen Identifikation für Aktivitätsträger zurecht kommen. Der verwendete Sperr-Mechanismus wird in Kapitel 3.2.2.5 näher erläutert.

Falls das Basis-Objekt übertragen werden soll (siehe Kapitel 3.2.2.4), wird lediglich eine Instanz der Klasse REMOTEOBJECTREFERENCE übertragen. Mit deren Hilfe wird auf der Empfängerseite ein Stub gebildet, über den der Zugriff auf das Remote-Objekt erfolgt.

3.2.2.2 Die Strategie AccessRemoteTransmitROR

Das Strategie-Objekt ACCESSREMOTETRANSMITROR ist mit dem Stub verbunden, über den der Zugriff auf das Remote-Objekt erfolgen soll. Es fängt die Sperr- und Freigabe-Events und die Events, welche bei einem Methodenaufruf an Stub entstehen, ab und leitet sie an das Strategie-Objekt des Remote-Objektes weiter. Die genaue Beschreibung des Vorgangs erfolgt in Kapitel 3.2.2.3.

Das Basis-Objekt wird auf die gleiche Weise übertragen wie bei der vorhergehenden Strategie.

3.2.2.3 Ablauf eines Zugriffs

Ein Methodenaufruf an einem Remote-Objekt umfaßt folgende Schritte:

1. Das aufrufende Objekt ruft am Stub die Methode auf.

2. Das ACCESSREMOTETRANSMITROR Objekt des Stubs fängt den Methodenaufruf ab und ruft die Methode `synchronRMI()` der Klasse REMOTEACCESSCONTROL auf, um den Methodenaufruf weiterzuleiten.

3. Die Methode `synchronRMI()` öffnet eine TCP-Verbindung zum Prozeß, der das Remote-Objekt enthält, und überträgt den Methodenaufruf samt Parameter.

4. Das REPLICATIONENVIRONMENT (eine Klasse die von jedem Prozeß der Remote-Anfragen entgegen nehmen möchte instantiiert werden muß) des Prozesses empfängt den Methodenaufruf und die Parameter.

5. Das REPLICATIONENVIRONMENT stellt über eine Tabelle, in der alle Remote-Objekte eines Prozesses eingetragen sind, fest, an welchem Objekt der Methodenaufruf erfolgen soll.

6. Nachdem das Objekt festgestellt wurde, wird der Methodenaufruf an das Strategie-Objekt des Remote-Objektes weitergeleitet.

7. Das Strategie-Objekt ruft die Methode am Remote-Objekt auf.

8. Der Rückgabewert wird auf dem gleichen Weg wieder zurückgegeben.

Zum besseren Verständnis des Ablaufes kann noch einmal Abbildung 3.2 herangezogen werden.

Bei einem Sperr- oder Freigabe-Event werden die gleichen Schritte ausgeführt wie bei einem Methodenaufruf, außer daß es nicht bis zum Remote-Objekt weitergegeben, sondern bereits von seinem Strategie-Objekt bearbeitet wird. Vom Strategie-Objekt wird der gleiche Sperr-Mechanismus wie für lokale Anfragen verwendet.

3.2.2.4 Objektübertragung

Zur Objektübertragung wird von den Methoden der Klasse REMOTEACCESSCONTROL zunächst eine Instanz der Klasse OBJECTTRANSMISSIONCONTROL gebildet. An dieser Instanz wird dann für jedes zu übertragende Objekt die Methode `sendObject()` aufgerufen. Von dieser Methode wird dann das Objekt abhängig von seiner Art übertragen:

- Wenn es sich um ein Array (in Java werden Arrays wie Objekte verwaltet) handelt, werden die einzelnen Komponenten des Arrays übertragen (je nach Typ der Komponente unterschiedlich).
- Falls an das Objekt ein Meta-Objekt gebunden ist, wird die Methode `sendObject()` am Meta-Objekt aufgerufen. Das Meta-Objekt gibt daraufhin die Aufforderung zur Objektübertragung an sein Strategie-Objekt weiter. Dort wird eine der aktuellen Strategie entsprechende Methode der zu Anfang gebildeten Instanz der Klasse OBJECTTRANSMISSIONCONTROL aufgerufen. Eine Auflistung dieser Methoden und ihrer Arbeitsweisen erfolgt weiter unten.
- In allen anderen Fällen wird das Objekt, mit Hilfe der Methode `sendObjectCopy()`, als Kopie übertragen.

Während einer Objektübertragung kann es vorkommen, daß ein Objekt mehrfach übertragen werden müßte (durch Abhängigkeiten zwischen den Objekten). Um dies zu verhindern werden alle Objekte, die bereits übertragen wurden, registriert. Falls nun ein bestimmtes Objekt ein zweites mal übertragen werden soll, kann dies festgestellt werden. In diesem Fall wird nur eine eindeutige Objektidentifikation übertragen, so daß der Empfänger an dieser Stelle das bereits erhaltene Objekt verwenden kann.

Übertragung als Kopie
Bei Aufruf der Methode `sendObjectCopy()` wird das Objekt als Kopie übertragen. Hierzu wird die Struktur des zu übertragenden Objektes analysiert und jede Komponente einzeln übertragen. Die elementaren Datentypen werden direkt übertragen und für alle anderen wird die bereits bekannte Methode `sendObject()` verwendet.

Übertragung als Stub
Diese Übertragungsart wird von den Strategie-Objekten ACCESSLOCALTRANSMITROR und ACCESSREMOTETRANSMITROR verwendet und mit Hilfe der Methode `sendROR()` realisiert. Falls es sich bei dem zu übertragenden Objekt um ein Remote-Objekt handelt, wird die REMOTEOBJECTREFERENCE (eine eindeutige Identifikation des Objektes im ganzen System) des Objektes übertragen. Ansonsten handelt es sich um einen Stub eines Remote-Objektes. In diesem Fall wird nicht die REMOTEOBJECTREFERENCE des Stubs übertragen, sondern die des über den Stub erreichbaren Remote-Objektes. Auf der Empfängerseite wird mit Hilfe der REMOTEOBJECTREFERENCE ein Stub generiert, über den das Remote-Objekt angesprochen werden kann.

In jedem Prozeß wird ein Verzeichnis der eigenen Remote-Objekte und der bereits generierten Stubs unterhalten. Damit kann bei Empfang einer REMOTEOBJECTREFERENCE festgestellt werden, ob ein neuer Stub generiert werden muß oder ein bereits bestehender verwendet werden kann. Falls die REMOTEOBJECTREFERENCE auf ein eigenes Remote-Objekt verweist, wird natürlich kein Stub erzeugt, sondern dies direkt verwendet.

Übertragung als Multi-Stub, System-Objekt oder Replikat
Diese Übertragungsarten werden durch die Methoden `sendMultiROR()`, `sendSystemObject()` und `sendObjectReplica()` realisiert. Ihre Funktionsweise wird erst in Kapitel 3.2.3 erläutert, da sie ausschließlich von den dort neu eingeführten Strategien verwendet werden.

3.2.2.5 Sperr-Mechanismus

Als Grundlage für den Sperr-Mechanismus mußte eine Möglichkeit gefunden werden, den Aktivitätsträgern eine über mehrere Threads gleichbleibende, im ganzen System eindeutige Identifikation zuzuordnen. Dies erfolgt nach folgendem Schema:

Von der Klasse REPLICATIONENVIRONMENT wird in jedem Prozeß ein Tabelle unterhalten, die Threads eine eindeutige Aktivitätsträger-Identifikation zuordnet.

- Falls ein Thread nicht in der Tabelle eingetragen ist, wurde der zu ihm gehörende Aktivitätsträger in diesem Prozeß erzeugt, und ihm kann deshalb bei Bedarf eine Identifikation zugewiesen werden. Die neue Identifikation wird erzeugt aus der Identifikation des Prozesses und der Identifikation des Threads innerhalb des Prozesses.

- Wenn ein Aktivitätsträger den Prozeß wechselt (durch Zugriffe auf Remote-Objekte) wird dessen Identifikation an den neuen Prozeß übergeben. Dadurch kann dieser den für den Aktivitätsträger neu generierten Thread zusammen mit der Identifikation in die Tabelle eintragen.

Durch dieses Schema wird erreicht, daß von jedem Thread eindeutig festgestellt werden kann, zu welchem Aktivitätsträger er gehört.

Der Sperr-Mechanismus arbeitet nun wie folgt: Wenn ein Thread eine Sperre anfordert, so wird zuerst die Identifikation seines Aktivitätsträgers ermittelt, und danach sind drei Fälle zu unterscheiden:

Das Objekt ist nicht belegt:
Das Objekt wird als belegt markiert, und die Identifikation des belegenden Aktivitätsträgers wird notiert.

Das Objekt ist bereits von diesem Aktivitätsträger belegt:
Die Anforderung wird zugelassen und lediglich ein Zähler erhöht. Der Zähler hat die Aufgabe festzustellen, ob genausoviele Sperr- wie Freigabe-Anforderungen vorliegen, da erst dann das Objekt wieder freigegeben werden kann.

Das Objekt ist bereits von einem anderen Aktivitätsträger belegt:
Der Thread wird blockiert bis das Objekt nicht mehr belegt ist.

3.2.3 Weitere Strategien

In diesem Kapitel werden weitere durch das Grundsystem zu Verfügung gestellte Strategien vorgestellt.

3.2.3.1 Die Strategie MultiAccessRemoteTransmitROR

Diese Strategie realisiert im Gegensatz zu ACCESSREMOTETRANSMITROR nicht den Stub für ein Remote-Objekt, sondern für mehrere, also einen Multi-Stub. Alle Remote-Objekte müssen den gleichen Typ haben, da Methodenaufruf, bzw. Sperr- und Freigabe-Anforderung, einfach an alle angemeldeten Remote-Objekte weitergegeben werden. Als Aktivitätsträger-Identifikation wird allen die gleiche übergeben, da die Methodenaufrufe an den Remote-Objekten alle vom gleichen Aktivitätsträger aus erfolgen. Als Rückgabewert wird ein beliebiger ausgewählt.

Die An- bzw. Abmeldung von Remote-Objekten erfolgt über die Methoden `registerRemoteObject()` und `unregisterRemoteObject()`.

Zur Objektübertragung wird die Methode `sendMultiROR()` der Klasse OBJECTTRANSMISSIONCONTROL verwendet. Durch diese Methode wird auf der Empfängerseite zuerst ein neuer Multi-Stub erzeugt und danach die Remote-Objekte, welche bereits beim zu übertragenden Multi-Stub angemeldet waren, registriert.

3.2.3.2 Die Strategie AccessTransmitSystemObject

Objekte, an die das Strategie-Objekt ACCESSTRANSMITSYSTEMOBJECT gebunden ist, werden als System-Objekte behandelt. Der Zugriff auf System-Objekte erfolgt wie bei lokalen Objekten.

Die Übertragung von System-Objekten wird durch die Methode `sendSystemObject()` der Klasse OBJECTTRANSMISSIONCONTROL realisiert und läuft folgendermaßen ab:

1. Der Typ und die Identifikation des System-Objektes werden übertragen.

2. Auf der Empfängerseite wird mit Hilfe der Identifikation festgestellt, ob das System-Objekt bereits vorher schon einmal empfangen wurde. Wenn dies der Fall ist wird das vorher schon generierte Objekt verwendet.

3. Wenn das Objekt vorher noch nicht empfangen wurde, wird versucht, ein neues zu erzeugen. Dazu wird ein passender Konstruktor des Objektes gesucht und aufgerufen.

3.2.3.3 Die Strategie ReplicationStrategy

Die Klasse REPLICATIONSTRATEGY stellt eine abstrakte Oberklasse für alle Replikationsstrategien dar. Sie stellt keinerlei Funktionalität zur Verfügung und dient nur der besseren Strukturierung.

Die von RepliXa zur Verfügung gestellten Replikationsstrategien werden in den Kapiteln 4 und 5 vorgestellt.

Zur Objektübertragung wird von den RepliXa eigenen Replikationsstrategien die Methode `sendObjectReplica()` der Klasse OBJECTTRANSMISSIONCONTROL verwendet. Dieser

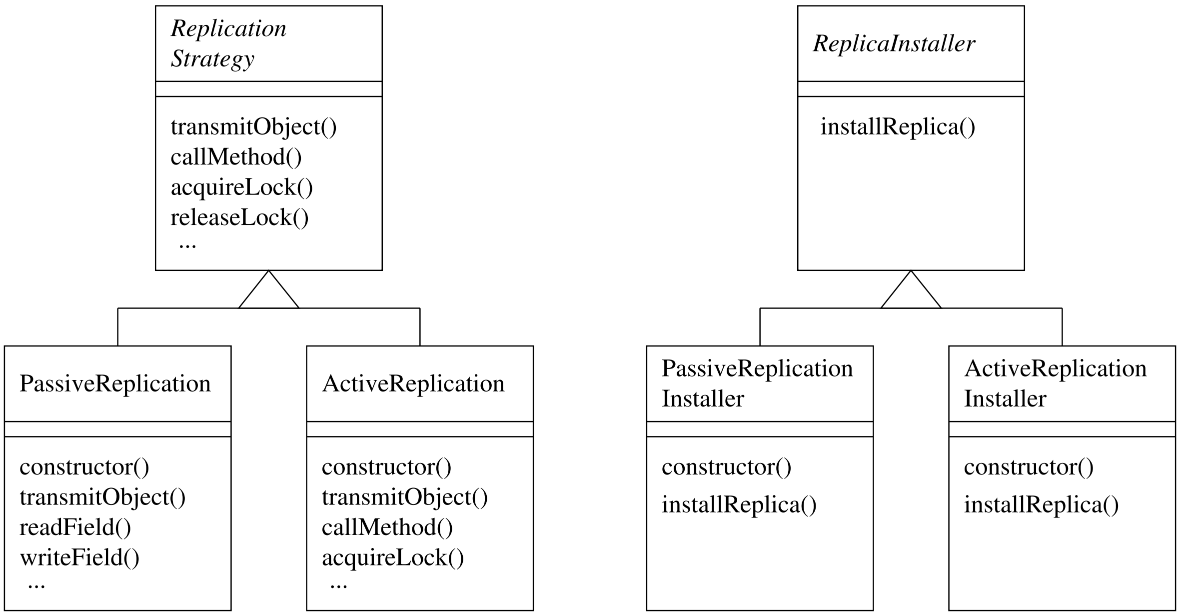


Abbildung 3.4: Darstellung der Zusammenhänge zwischen den einzelnen von RepliXa zur Verfügung gestellten Replikationsstrategien und der Zusammenhänge zwischen ihren Installationsobjekten.

Methode muß ein Installationsobjekt (eine Instanz einer Unterklasse der abstrakten Klasse REPLICAINSTALLER) übergeben werden. Dieses Installationsobjekt wird dann übertragen. Der empfangende Prozeß ruft danach an diesem Installationsobjekt die Methode `installReplica()` auf. Von dieser Methode werden dann alle zur Installation des neuen Replikates nötigen Schritte ausgeführt. Die Zusammenhänge zwischen den Replikationsstrategien und die ihrer Installationsobjekte sind in Abbildung 3.4 dargestellt.

Durch diese flexible Art der Objektübertragung ist es problemlos möglich, neue Replikationsstrategien in RepliXa zu integrieren. Folgende zwei Schritte sind dazu nötig:

- Implementierung der gewünschte Replikationssemantik in einer Unterklasse der abstrakten Klasse REPLICATIONSTRATEGY.

- Implementierung eines dazu passenden Installationsobjektes, um ein Replikat in einem neuen Prozeß zu installieren.

Mit diesem Ansatz sollte es sogar möglich sein, nicht nur replizierte Objekte, sondern auch migrierende Objekte oder ähnliches zu realisieren.

3.2.4 NameService

Ein Nameservice wird durch ein Exemplar der Klasse NAMESERVICE realisiert. In jedem Prozeß sollte genau ein Nameservice, also ein Exemplar dieser Klasse, vorhanden sein und ein globaler Zugriffspunkt darauf vorhanden sein. Die Klasse NAMESERVICE wurde nach dem objektbasierten Entwurfsmuster Singleton konzipiert, da dies die obigen Forderungen erfüllt. Informationen zu diesem Entwurfsmuster sind zu finden in [GHJ96] und [Lar98].

Die Klasse NAMESERVICE besitzt ein Klassenmethode[1] `getLocalNS()` die bei ihrem Aufruf das einzige Exemplar der Klasse NAMESERVICE zurückgibt. Falls noch kein Exemplar vorhanden war, wird es zuerst generiert. Anschließend können an diesem Exemplar die Methoden `register()` und `unregister()` aufgerufen werden, um Objekte unter symbolischen Namen ein- und auszutragen. Die Methode `get()` liefert das zu einem symbolischen Namen registrierte Objekt.

Der zu einem Prozeß gehörende Nameservice stellt ein Remote-Objekt dar, und durch Aufruf der Klassenmethode `getRemoteNS()` der Klasse NAMESERVICE in einem anderen Prozeß wird ein Stub generiert, über den auf das Remote-Objekt zugegriffen werden kann. Bei der Methode `getRemoteNS()` muß die Portnummer und der Rechnername des zu kontaktierenden Nameservices angegeben werden.Falls kein Rechnername angegeben wird, geht die Methode davon aus, daß sich der Prozeß auf dem lokalen Rechner befindet.

3.3 Erweiterungen

In diesem Abschnitt sollen einige mögliche Erweiterungen, die zum Teil aus Zeitgründen nicht implementiert werden konnten, vorgestellt werden.

3.3.1 Pool mit Threads

Das REPLICATIONENVIRONMENT erzeugt in der aktuellen Version bei jedem Zugriff auf ein Remote-Objekt von einem anderen Prozeß aus einen eigenen Thread, der diesen bearbeitet. Danach wird dieser Thread wieder beendet. Um den Aufwand zur Erzeugung und Beendung dieser Threads zu reduzieren wäre es sinnvoll, einen Pool mit Threads zu unterhalten. Bei Bedarf wird dem Pool eine Thread entnommen, der dann den Zugriff bearbeitet und danach wird dieser Thread wieder in den Pool zurückgestellt. Die Anzahl der zur Verfügung stehenden Threads im Pool wäre natürlich je nach Bedarf dynamisch anzupassen.

3.3.2 Halten der TCP-Verbindung

Im Moment wird bei jedem Zugriff auf ein Remote-Objekt eine neue TCP-Verbindung zwischen den beiden beteiligten Prozessen aufgebaut. Eine andere Variante wäre es, nur einmal eine TCP-Verbindung aufzubauen und über diese dann alle Zugriffe von einem bestimmten Stub auf ein bestimmtes Remote-Objekt durchzuführen. Der Vorteil dieser Variante ist, daß der Aufwand zum Auf- und Abbau der Verbindung vermieden wird. Sie hat jedoch den Nachteil, daß für jede offene Verbindung auf der Empfängerseite ein Thread gebraucht wird, der auf neue zu bearbeitende Zugriffe wartet.

Da also beide Varianten ihre Vor- und Nachteile haben, sollte eine Kombination aus beiden gewählt werden. Eine Möglichkeit hierzu wäre, daß die Verbindung nach einem Zugriff noch eine gewisse Zeit offen gehalten wird und dann automatisch beendet wird. Dies hat den Effekt, daß, wenn in regelmäßigen Abständen Zugriffe erfolgen, wie dies beispielsweise bei der

[1] Eine Klassenmethode ist eine Methode die direkt über die Klasse aufgerufen wird, also kein Exemplare benötigt. Sie ist außerdem von überall aus zugreifbar.

Kommunikation zwischen Replikaten der Fall sein könnte, die Verbindung gehalten wird. Wenn jedoch längere Zeit kein Zugriff mehr erfolgt, wird sie automatisch beendet.

3.3.3 Fehlertoleranter Sperr-Mechanismus

Der von RepliXa verwendete Sperr-Mechanismus ist nicht fehlertolerant. Wenn ein Aktivitätsträger zuerst über einen Stub ein Remote-Objekt belegt und dann ausfällt, so wird dies vom Sperr-Mechanismus nicht erkannt. Dies hat zur Folge, daß das Objekt für immer belegt bleibt und deshalb von anderen Aktivitätsträgern nicht belegt werden kann.

Um Fehlertoleranz zu erreichen, müßte der Sperr-Mechanismus, solange eine Sperr-Anforderung eines anderen Aktivitätsträgers vorliegt, regelmäßig prüfen, ob die alte Belegung noch gültig ist. Sollte dies der Fall sein, müßte die alte Belegung aufgehoben werden.

Das Feststellen, ob eine Belegung noch gültig ist, könnte folgendermaßen erfolgen:

- Der Sperr-Mechanismus führt nicht nur einen Zähler darüber, wie oft ein Aktivitätsträger eine Sperre angefordert hat, sondern er unterhält auch eine Liste, die die Prozesse enthält, von denen aus dies geschah. Bei einer Freigabe-Anforderung wird der entsprechende Prozeß wieder aus der Liste ausgetragen und der Zähler entsprechend angepasst.

- Solange eine Anforderung eines anderen Aktivitätsträgers vorliegt, wird nachgeprüft, ob von den Prozessen in der Liste einer ausgefallen sind. Wenn dies der Fall ist, sind die Sperr-Anforderungen von diesem Prozeß nicht mehr gültig. Deshalb wird er dann aus der Liste ausgetragen und der Zähler entsprechend angepasst.

- Die Belegung ist dann ungültig, falls durch den obigen Ablauf die Liste mit Prozessen leer wird, da dann keine gültige Sperr-Anforderung mehr existiert.

Ein Problem, das sich bei dieser Vorgehensweise ergibt, ist, ob Prozesse, die durch eine zeitweiligen Partitionierung des Netzwerkes vorübergehend nicht erreichbar sind, als ausgefallen betrachtet werden sollten oder nicht.

Ein anderes Problem stellt die Frage dar, ob eine ungültige Belegung überhaupt aufgehoben werden kann bzw. wie dies zu realisieren ist. Falls eine Sperre beispielsweise dazu verwendet wird, ein Objekt von einem konsistenten Zustand in einen anderen zu überführen, ergeben sich Probleme, wenn sie bei Ausfall des Aktivitätsträgers einfach aufgehoben wird. Die Probleme entstehen dann, wenn der Aktivitätsträger in einer Phase ausfällt, in der sich das Objekt in einem inkonsistenten Zustand befindet. Wenn in diesem Fall nämlich die Sperre aufgehoben wird, ist der inkonsistente Zustand des Objektes für andere zugreifbar. Es müßten also, bevor die Sperre aufgehoben wird, die Änderungen durch den Aktivitätsträger entweder rückgängig gemacht oder vollendet werden.

Diese beiden Probleme müßten noch gelöst werden, bevor ein fehlertoleranter Sperr-Mechanismus implementiert werden kann.

3.3.4 Name-Service

Der Nameservice kann in zweierlei Hinsicht verbessert werden. Eine Verbesserung besteht darin, daß der Aufenthaltsort von Objekten verborgen wird (*Location Transparency*). Daß also nicht jeder Prozeß seinen eigenen Name-Service hat, sondern daß ein zentraler Name-Service existiert, der allen bekannt ist. Dann müßten die Prozesse, welche bei einem Name-Service ein Objekt anfordern wollen, keine Information wie Portnummer und Rechnername besitzen, wie das im Moment mit den prozeßeigenen Name-Services der Fall ist. Fehlertoleranz beim zentralen Name-Service kann beispielsweise dadurch erreicht werden, daß mehrere davon existieren und die Objekte in allen registriert werden.

Die zweite Verbesserung besteht darin, daß beispielsweise bei replizierten Objekten nicht nur ein Replikat registriert ist, sondern alle. Wenn nur ein Replikat registriert würde, so wäre das replizierte Objekt nach Ausfall dieses Replikates nicht mehr erreichbar. Ein ähnlicher Ansatz wurde auch beim Globe Projekt gewählt. Information über den Name- und Location-Service von Globe sind in [HST96] und [SHH98] zu finden.

3.3.5 Garbage Collection

Zuerst wird erläutert, wozu ein Garbage Collection Algorithmus gebraucht wird und wie er arbeitet. Danach wird dargelegt, warum das Grundsystem eine Garbage Collection benötigt, obwohl jeder Prozeß seine eigene Garbage Collection hat. Zum Schluß wird noch eine mögliche Realisierung vorgestellt.

3.3.5.1 Aufgaben und Arbeitsweisen

Aufgabe der Garbage Collection ist es, Objekte, die im System nicht mehr benötigt werden, automatisch zu löschen. Da es sich in unserem Fall um passive Objekte[2] handelt, sind alle Objekte, auf die keine Referenz[3] mehr besteht, nicht mehr verwendbar, und deshalb unbrauchbar. Die Grundidee der Garbage Collection Algorithmen ist nun festzustellen, ob auf ein Objekt noch Referenzen bestehen, und falls nicht zu löschen. Sie können, entsprechend ihrer Vorgehensweise dabei, in zwei Gruppen eingeteilt werden [JL96]: Die Reference Counting Algorithmen und die Tracing Algorithmen. Im Anschluß werden kurz die Funktionsweise und die Vor- und Nachteile der beiden Gruppen erläutert. Diese können ebenfalls in [JL96] nachgelesen werden.

Reference Counting Algorithmen
Die Grundidee dieser Algorithmen ist es, die Referenzen auf ein Objekt zu zählen. Wenn eine neue Referenz angelegt wird, wird ein Zähler erhöht, und wenn eine Referenz gelöscht wird, wieder erniedrigt. Falls der Zähler anzeigt, daß keine Referenz mehr auf das Objekt besteht, kann es gelöscht werden.

Tracing Algorithmen
Diese Algorithmen kommen erst zum Einsatz, wenn kein Speicher mehr verfügbar ist. Wenn sie aktiviert werden, unterbrechen sie zuerst alle Basisberechnungen, um zu verhindern, daß sich an den Referenzen etwas ändert. Der eigentliche Algorithmus läuft dann in drei Schritten ab:

[2] Passive Objekte sind Objekte in die erst ein Aktivitätsträger einwandern muß, bevor es auf das System Einfluß nehmen kann.

[3] Eine Referenz ist ein Verweis auf ein Objekt. Über diesen Verweis können beispielsweise Daten des Objektes abgerufen werden oder Methoden des Objektes aufgerufen werden.

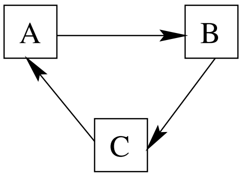


Abbildung 3.5: Ein Zyklus bei dem jeweils ein Objekt auf ein anderes verweist. Da auf jedes Objekt eine Referenz verweist, wird von Reference Countig Algorithmen keines der Objekte gelöscht, selbst wenn es nicht mehr benötigt wird. Für Tracing Algorithmen stellen Zyklen kein Problem dar, da bei ihnen der ganze Zyklus als nicht erreichbar erkannt wird.

1. Zuerst wird mit Hilfe des Laufzeitsystems eine Grundmenge von Objekten ermittelt, die nicht gelöscht werden dürfen. Hierzu gehören beispielsweise global erreichbare Objekte oder Objekte, in denen sich im Moment ein Aktivitätsträger befindet.

2. Danach werden alle Objekte, die von dieser Grundmenge aus über Referenzen erreichbar sind, ermittelt.

3. Alle Objekte, die nicht zur Grundmenge gehören oder im vorhergehenden Schritt ermittelt wurden, sind nicht mehr erreichbar und können deshalb gelöscht werden.

Die Tracing Algorithmen unterscheiden sich sehr darin, wie sie in Schritt zwei die noch benötigten Objekte ermitteln und in Schritt drei die nicht mehr benötigten löschen.

Reference Counting Algorithmen haben gegenüber Tracing Algorithmen eine Vorteil und einen Nachteil. Der Vorteil besteht darin, daß sie die Basisberechnung nicht unterbrechen müssen, der Nachteil darin, daß sie Zyklen, wie in Abbildung 3.5 dargestellt, nicht erkennen können. Dies hat zur Folge, daß eventuell nicht alle Objekte, die nicht mehr benötigt werden, gelöscht werden.

3.3.5.2 Verteilte Garbage Collection

Das Grundsystem sollte noch um einen verteilten Garbage Collection Algorithmus für die Remote-Objekte und ihre Stubs ergänzt werden, da Stubs von der lokalen Garbage Collection nicht erfasst werden und Remote-Objekt von ihr nicht gelöscht werden dürfen. Stubs können nicht erfasst werden, da in jedem Prozeß ein Verzeichnis der bereits generierten Stubs vorhanden ist und deshalb mindestens eine Referenz auf das Stub-Objekt verweist. Die Remote-Objekte dürfen nicht gelöscht werden, da die lokale Garbage Collection keine Informationen darüber besitzt, ob zu dem Remote-Objekt noch Stubs existieren oder nicht.

3.3.5.3 Mögliche Realisierung

Da bei Tracing Algorithmen die gesamten Basisberechnungen angehalten werden müßten, was in einem verteilten System allein schon unter dem Aspekt der Performance sehr problematisch ist, wird auf ihren Einsatz hier verzichtet. Bei der hier vorgeschlagenen Realisierung handelt es sich also um einen Reference Countig Algorithmus. Die Entstehung von Zyklen sollte nach Möglichkeit durch den Programmierer verhindert werden.

Die hier vorgeschlagene Realisierung ist im Wesentlichen entnommen aus [BEN93], worin auch eine Erweiterung zu einem fehlertoleranten Algorithmus zu finden ist. Die Grundidee ist folgende:

- Alle Remote-Objekte unterhalten eine Liste mit ihren Stubs und wenn diese leer ist, sind sie nur noch lokal ansprechbar und dürfen deshalb vom lokalen Garbage Collector gelöscht werden.
- Die Stubs werden nur über Weak-References[4] in dem Verzeichnis der bereits generierten Stubs eingetragen. Wenn nun der lokale Garbage Collector feststellt, daß ein Stub nicht mehr benötigt wird, so löscht er ihn. Bevor der Stub jedoch gelöscht wird, wird das Remote-Objekt darüber informiert, so daß es seine Liste mit Stubs anpassen kann.

[4]Weak-References werden vom lokalen Garbage Collector nicht berücksichtigt, das heißt ein Objekt, auf das nur Weak-References verweisen, wird von diesem gelöscht.

Kapitel 4

Aktive Replikation

Aufgabe der aktiven Objektreplikation ist es, wie bereits in der Einleitung erwähnt, eine Menge von Objekten in einem konsistenten Zustand zu halten. Dies bedeutet, daß sie alle den gleichen internen Zustand haben. Die Grundidee ist nun, daß, wenn alle Objekte in einem konsistenten Zustand sind und dann an allen die gleiche Methode aufgerufen wird, sie sich nach Beendigung des Aufrufs wieder in einem konsistenten Zustand befinden. Um die Objekte in einem konsistenten Zustand zu halten, müssen also die Methodenaufrufe, die nur an einem Replikat erfolgen, abgefangen werden und an alle Replikate weitergeleitet werden.

Diese an sich einfache Idee bringt jedoch in der Praxis viele Probleme mit sich. Diesen Problemen und ihren Lösungen widmet sich der nächste Abschnitt. Danach wird die von RepliXa bereitgestellte aktive Replikationsstrategie näher erläutert, und zum Schluß erfolgt noch eine Beschreibung möglicher Erweiterungen. Ein Beispielprogramm zur Verwendung der aktiven Replikation ist in Anhang B zu finden.

4.1 Aufgetretene Probleme und ihre Lösungen

Die in diesem Abschnitt vorgestellten Probleme sind zum größten Teil bekannt und wurden schon des öfteren in der einschlägigen Fachliteratur erläutert. Zu den Kapiteln 4.1.2, 4.1.3 und 4.1.5 kann [KG96] herangezogen werden.

4.1.1 Determinismus

Die Methoden, welche an den Replikaten aufgerufen werden, müssen deterministisch sein, da sonst nicht gewährleistet ist, daß sich die Replikate nach Bearbeitung des Aufrufs in einem konsistenten Zustand befinden. Desweiteren kann immer nur ein Aktivitätsträger innerhalb eines Replikates zugelassen werden, da durch unterschiedliches Timing zwischen den Aktivitätsträgern verschiedene Endzustände entstehen könnten.

4.1.2 Reihenfolge der Methodenaufrufe

Methodenaufrufe an den Replikaten dürfen zwar in einer beliebigen Reihenfolge erfolgen, müssen aber an den einzelnen Replikaten alle in der gleichen Reihenfolge erfolgen, damit sie jeweils den gleichen Ausgangszustand vorfinden. Dies bedeutet, daß die Methodenaufrufe nicht nur

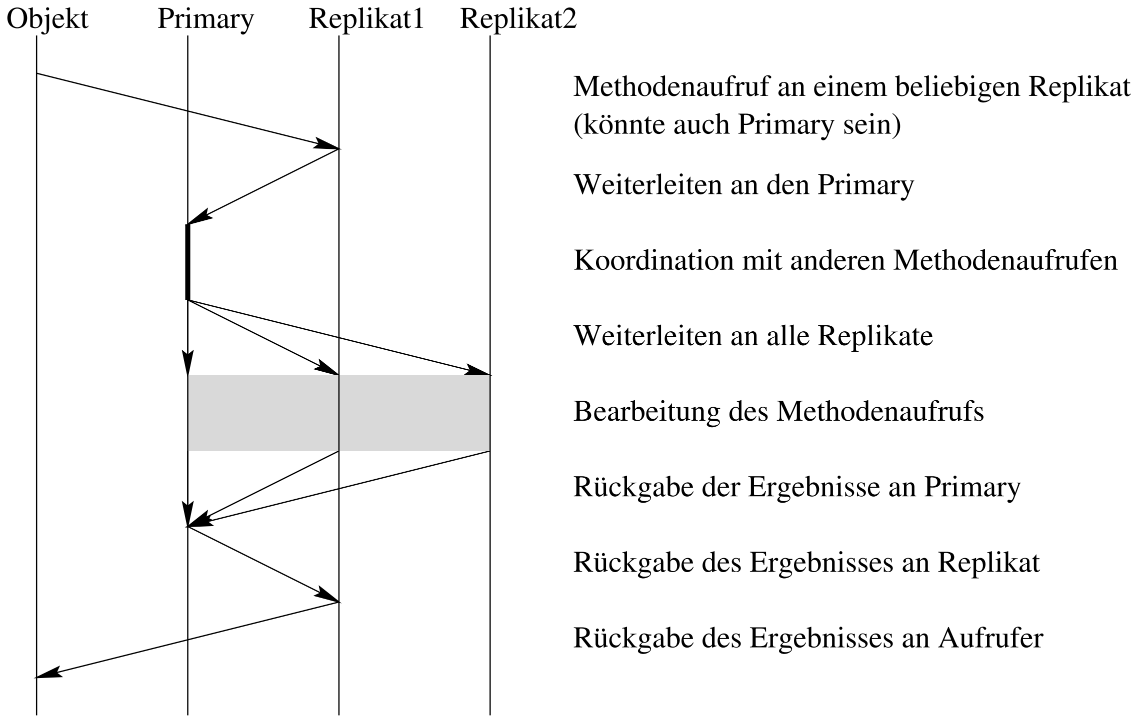


Abbildung 4.1: Ablauf eines replizierten Methodenaufrufs.

abgefangen und an alle weitergeleitet werden müssen, sondern daß auch eine totale Ordnung[1] gewährleistet werden muß.

Die bei RepliXa gewählte Lösung dieses Problems ist die Verwendung eines ausgezeichneten Replikates, dem Primary. Alle Methodenaufrufe werden zunächst nur an den Primary weitergeleitet. Der Primary legt dann für alle eingehenden Methodenaufrufe eine Reihenfolge fest und leitet dann die Methodenaufrufe in dieser Reihenfolge an alle Replikate weiter. Der Ablauf eines Methodenaufrufs ist in Abbildung 4.1 dargestellt.

Eine weitere Möglichkeit wird in Abschnitt 4.3 vorgestellt.

4.1.3 Mehrfache Aufrufe

Das Problem der mehrfachen Aufrufe entsteht, wenn von einem replizierten Objekt aus eine Methode eines anderen Objektes aufgerufen wird. In diesem Fall wird, da der Aktivitätsträger repliziert ist, die Methode des anderen Objektes nicht nur einmal, sondern mehrfach aufgerufen. Dies muß verhindert werden, da sich die Semantik eines replizierten Objektes nicht von der eines unreplizierten unterscheiden sollte. Der Sachverhalt ist in Abbildung 4.2 dargestellt.

Mehrfache Aufrufe können auf unterschiedliche Weise verhindert werden. Eine Möglichkeit besteht darin, daß sich die Replikate untereinander koordinieren, so daß bei einem verschachtelten Aufruf von allen Replikaten zusammen nur ein einziger Aufruf erfolgt. Die andere Möglichkeit ist, daß das Objekt, an dem die Aufrufe erfolgen, doppelte Aufrufe erkennt und ignoriert.

[1]Totale Ordnung bedeutet, daß alle Replikate die Methodenaufrufe in der gleichen Reihenfolge erhalten.

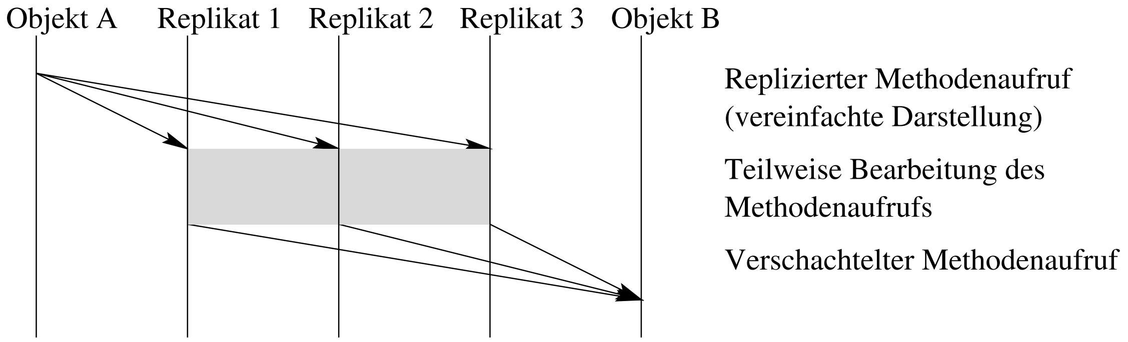


Abbildung 4.2: Darstellung des Problems der mehrfachen Aufrufe: Das Objekt A ruft an einem replizierten Objekt eine Methode auf. Dies hat eine Abarbeitung der Methode in den Replikaten 1-3 zur Folge. Wenn nun während dieser Abarbeitung in den Replikaten ein verschachtelter Methodenaufruf an Objekt B erfolgt, so wird die Methode am Objekt B mehrfach aufgerufen, was nicht sein darf.

Bei RepliXa werden die mehrfachen Aufrufe durch eine Koordination innerhalb der Replikate gelöst. Dies hat verschiedene Gründe: Zum einen hat eine Lösung auf der Seite des aufgerufenen Objektes zwei Nachteile. Der eine ist, daß Informationen übertragen und verwaltet werden müßten, mit deren Hilfe doppelte Aufrufe erkannt werden könnten, und der andere besteht darin, daß es sich bei den aufgerufenen Objekten nicht notwendigerweise um ebenfalls replizierte Objekte handelt. Es könnten also auch Remote-Objekte sein, die replizierte Aufrufe erkennen müßten, obwohl sie mit Replikation an sich nichts zu tun haben. Zum anderen existiert dadurch, daß zum Zweck der totalen Ordnung für Methodenaufrufe einen Primary eingeführt wurde, bereits ein idealer Koordinationspunkt innerhalb der Replikate.

Wenn also innerhalb eines Replikates ein verschachtelter Methodenaufruf erfolgt, so wird dieser abgefangen und an den Primary weitergeleitet. Der Primary wartet dann, bis alle Aufrufe eingegangen sind (es wäre auch denkbar, daß er, sobald ein Aufruf eingeht, diesen sofort weiterleitet) und ruft dann die Methode am Objekt auf. Der Ablauf ist in Abbildung 4.3 dargestellt.

4.1.4 Implizit replizierte Objekte

Ein weiteres Problem ergibt sich mit implizit replizierten Objekten. Dabei handelt es sich um Objekte, die nicht explizit, also durch Binden an eine Replikationsstrategie, repliziert werden, sondern dadurch, daß sie ein fester Teil eines replizierten Objektes sind. Das bedeutet also, daß jedes Replikat eine eigene Kopie dieses Objektes hat, auf die es verweist. Diese Kopien müssen auch in einem konsistenten Zustand gehalten werden, da sie Bestandteil des replizierten Objektes sind, wodurch dieses Teilobjekt also implizit repliziert wird. Die Replikate und die Kopien des implizit replizierten Objektes sind in Abbildung 4.4 a) dargestellt.

Implizit replizierte Objekte können auf drei verschiedene Arten entstehen:

- Innerhalb eines Methodenaufrufs, wenn ein neues Objekt angelegt wird, da dies dann in jedem Replikat geschieht, wodurch ein implizit repliziertes Objekt entsteht.

- Beim Binden einer Replikationsstrategie an ein Basis-Objekt, falls in diesem Basis-Objekt Verweise auf Objekte, die durch ihre Strategie-Objekte als Kopien übertragen werden, vorhanden sind. Dies ist der Fall, da beim Anlegen neuer Replikate diese Objekte

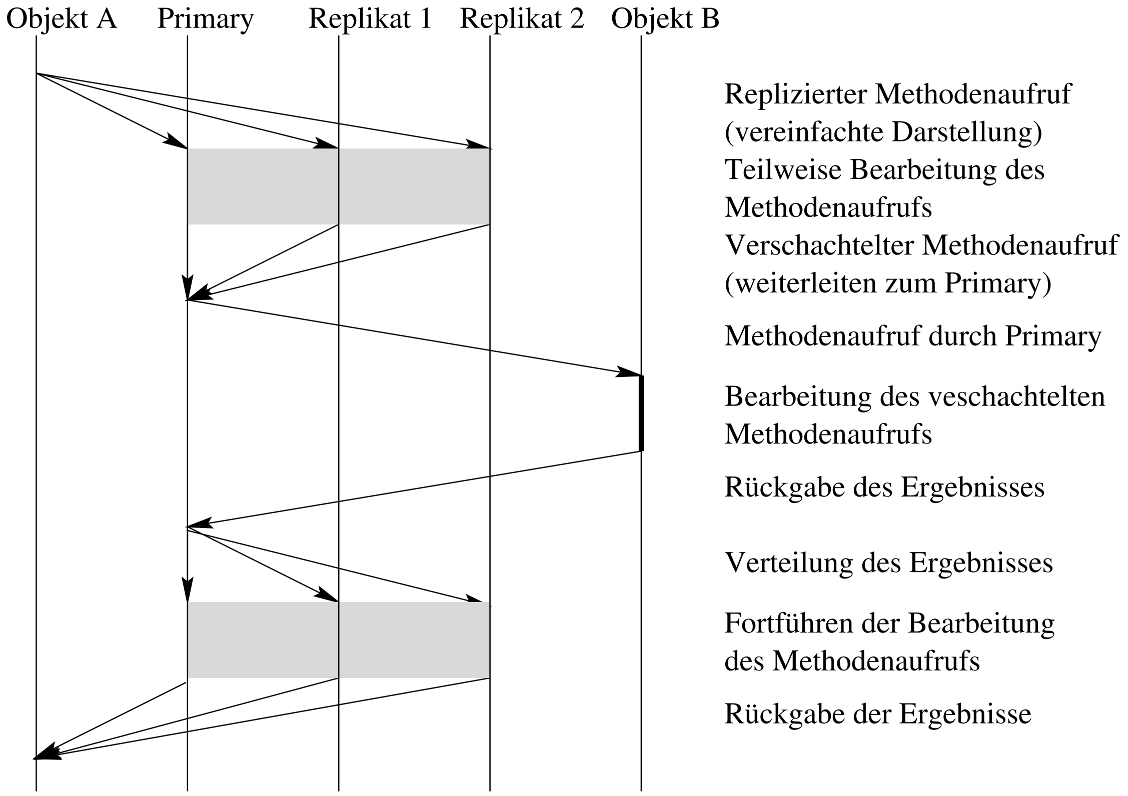


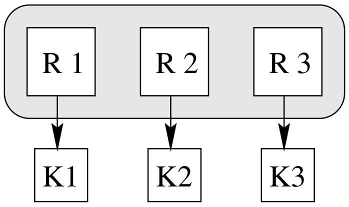
Abbildung 4.3: Ablauf eines verschachtelten Methodenaufrufs.

dann als Kopien übertragen werden, wodurch zusammen mit dem Original ein implizit repliziertes Objekt entsteht.

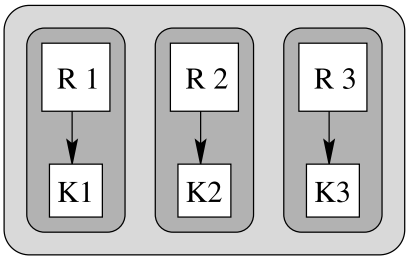
- Wenn als Parameter oder Rückgabewert eines Methodenaufrufs Objekte übergeben werden, die als Kopie übertragen werden, da in diesem Fall jedes Replikat seine eigene Kopie erhält.

Da es von diesen Teilobjekten stets genau so viele wie Replikate gibt, müssen verschachtelte Methodenaufrufe von den Replikaten an diesen Teilobjekten direkt ausgeführt werden, dürfen also nicht über den Primary zu einem zusammengefasst werden. Dadurch ergeben sich jedoch zwei Probleme: Das eine besteht darin festzustellen, ob der verschachtelte Methodenaufruf über den Primary erfolgen sollte oder nicht, also ob es sich bei dem Zielobjekt um ein Teilobjekt handelt oder nicht. Das andere ist folgendes: Nachdem durch einen verschachtelten Methodenaufruf die Aktivitätsträger von den Replikaten in die Teilobjekte übergegangen sind, können sie nicht mehr kontrolliert werden, da an die Teilobjekte keine Strategie-Objekte hierzu gebunden sind. Dies hat zur Folge, daß, falls aus den Teilobjekten erneute verschachtelte Methodenaufrufe an einem Nicht-Teilobjekt erfolgen, diese nicht zu einem zusammengefasst werden können, obwohl dies zwingend erforderlich wäre.

Diese Probleme können auf zwei Arten gelöst werden. Eine beruht darauf, daß an die Teilobjekte spezielle Strategie-Objekte gebunden werden, welche dann mit dem Strategie-Objekt des Replikates zusammenarbeiten. Dadurch kann um ein Replikat und seine Teilobjekte eine Hülle gebildet werden und diese somit zu einer Einheit verbunden werden, welche dann re-



a)



b)

Abbildung 4.4:
a) Darstellung eines replizierten Objektes mit seinen drei Replikaten R1-R3. Jedes Replikat hat einen Verweis auf ein Kopie K1-K3 eines nicht replizierten Objektes.
b) Darstellung der Replikate R1-R3 und der Kopien K1-K3, wobei jedoch jeweils um ein Replikat und eine Kopie eine Hülle gebildet wurde, so daß die Kopien ebenfalls mit in die Replikation aufgenommen wurden.

pliziert wird. Diese Vorgehensweise ist in Abbildung 4.4 b) dargestellt. Eine andere Lösung, welche auch bei RepliXa gewählt wurde, ist die Existenz solcher Teilobjekte zu vermeiden.

Da Teilobjekte an drei verschiedenen Stellen entstehen können, müssen sie auch an diesen drei Stellen vermieden werden. Dies geschieht wie folgt:

- Beim Anlegen eines neuen Objektes innerhalb der Replikate koordinieren sich diese, ähnlich wie bei einem verschachtelten Methodenaufruf, über den Primary. Der Primary erzeugt dann ein neues Objekt, bindet an dieses ein Strategie-Objekt, welches das neue Objekt repliziert, und gibt an jedes Replikat ein Replikat des neuen Objektes zurück.

- Falls beim Binden einer Replikationsstrategie an ein Basis-Objekt Verweise auf Objekte, die als Kopien übertragen werden, vorhanden sind, werden diese in Remote-Objekte umgewandelt. Beim Anlegen neuer Replikate werden sie dann nur als Stub übertragen.

- Wenn als Parameter oder Rückgabewert eines Methodenaufrufs Objekte übergeben werden, die als Kopie übertragen werden, werden diese ebenfalls in Remote-Objekte umgewandelt.

Durch diese Vorgehensweise wird gewährleistet, daß keine Teilobjekte entstehen und somit die damit verbundenen Probleme vermieden.

4.1.5 Semantische Informationen

Mit Hilfe semantischer Informationen über Methodenaufrufe ist es möglich festzustellen, welche Methodenaufrufe den Objektzustand nicht verändern. Diese Methoden können dann direkt in dem Replikat, an dem sie aufgerufen wurden, ausgeführt werden und müssen nicht an den Primary weitergeleitet werden. Das Problem ist nun, wie die Strategie-Objekte semantische Informationen über die Methoden der Basis-Objekte erhalten. Eine Möglichkeit wäre, daß die Funktionalität des Laufzeitsystems des metaXa-Interpreters dahingehend erweitert wird, daß es solche Informationen zur Verfügung stellen kann. Eine andere, auch von RepliXa gewählte Möglichkeit besteht darin, daß Methoden, deren Methodenname eine bestimmte Endung aufweist, als nicht verändernd angenommen werden.

4.2 Realisierung

In diesem Abschnitt wird die Implementierung der im Rahmen von RepliXa implementierten aktiven Replikationsstrategie näher betrachtet. Nachdem die Grundidee vorgestellt wurde, wird zunächst darauf eingegangen, wie aus einem Objekt ein repliziertes Objekt entsteht und wie die Installation neuer Replikate vor sich geht. Danach wird der Ablauf eines Methodenaufrufs an einem Replikat und der Sperr-Mechanismus erläutert. Zum Schluß wird die durch diese Replikationsstrategie erreichte Fehlertoleranz vorgestellt.

4.2.1 Grundidee

An jedes Replikat eines replizierten Objektes ist ein Strategie-Objekt gebunden. Bei dem Strategie-Objekt handelt es sich um Instanzen der Klasse ACTIVEREPLICATION. Einem dieser Strategie-Objekte fällt die Rolle des Primary zu, welcher, wie in Abschnitt 4.1 erläutert wurde, zur Koordinierung benötigt wird. Die Strategie-Objekte kommunizieren untereinander mit Hilfe des RMI-Mechanismus.

Die Strategie-Objekte werden vom Laufzeitsystem über folgende Ereignisse am Basis-Objekt informiert:

- Methodenaufrufe an Replikaten, um sie an den Primary weiterzuleiten (siehe Abschnitt 4.2.4).
- Verschachtelte Methodenaufrufe, da hier ein Koordinierung erforderlich ist (siehe Abschnitt 4.2.4.1).
- Erzeugen neuer Objekte, da hier ebenfalls ein Koordinierung erforderlich ist (siehe Abschnitt 4.2.4.2).
- Sperr- und Freigabe-Events, um sie an den Primary weiterzuleiten (siehe Abschnitt 4.2.5).

4.2.2 Initialisierung

Um ein Basis-Objekt zu replizieren, muß eine Instanz des Strategie-Objektes ACTIVEREPLICATION daran gebunden werden. Dieses Strategie-Objekt wird dadurch automatisch zum Primary und koordiniert von da an alle Aktivitäten der Replikate. Während der Bindephase werden alle Objekte, auf die das Basis-Objekt verweist und die als Kopie übertragen würden, in Remote-Objekten umgewandelt, indem eine Instanz der Klasse ACCESSLOCALTRANSMITROR daran gebunden wird.

4.2.3 Installation eines neuen Replikates

Die Notwendigkeit, in einem Prozeß ein neues Replikat anzulegen besteht dann, wenn der Prozeß es bei einem Nameservice anfordert, oder es im Rahmen einer RMI als Parameter oder Rückgabewert übergeben wird. Der Ablauf ist dann wie folgt:

1. Durch das Grundsystem wird die Methode `transmitObject()` an einem Strategie-Objekt eines Replikates aufgerufen.

2. In dieser Methode wird zunächst ein Installationsobjekt (eine Instanz der Klasse ACTIVEREPLICATIONINSTALLER) erzeugt und initialisiert. Danach wird dieses Installationsobjekt mit Hilfe der Methode `sendObjectReplica()` der Klasse OBJECTTRANSMISSIONCONTROL an den Zielprozeß übertragen.

3. Das REPLICATIONENVIRONMENT des Zielprozesses empfängt das Installationsobjekt und ruft daran die Methode `installReplica()` auf. Alle weiteren Schritte werden von dieser Methode veranlasst.

4. Nun wird überprüft, ob in diesem Prozeß bereits ein Replikat vorhanden ist. Sollte dies der Fall sein, wird dieses zurückgegeben und die Installation ist beendet.

5. Jetzt wird eine Instanz des Basis-Objektes gebildet und ein Strategie-Objekt, welches vorher ebenfalls instantiiert und initialisiert wurde, daran gebunden.

6. Zum Schluß wird vom Primary noch ein gültiger Objektzustand für das Basis-Objekt angefordert und in dieses kopiert.

4.2.4 Ablauf eines Aufrufs

Ein Methodenaufruf läuft in folgenden Schritten ab:

1. Ein Methodenaufruf an einem Replikat wird vom Laufzeitsystem abgefangen, welches dann die Methode `callMethod()` an dem zu diesem Replikat gehörenden Strategie-Objekt aufruft.

2. In dieser Methode wird zuerst geprüft, ob es sich um eine nur lesende Methode handelt. Wenn dies der Fall ist, wird sie direkt von diesem Replikat bearbeitet. Ansonsten wird der Aufruf an den Primary weitergeleitet, indem die Methode `distributeCall()` an diesem aufgerufen wird. Vorher jedoch werden alle Objekte, die im Rahmen des Methodenaufrufs als Kopien übertragen würden, zu Remote-Objekte transformiert.

3. Der Primary koordiniert den Methodenaufruf dann mit anderen Methodenaufrufen, um einerseits eine totale Ordnung der Methodenaufrufe zu gewährleisten und andererseits zu vermeiden, daß sich mehr als ein, den Objektzustand verändernder, Aktivitätsträger in einem Replikat befindet.

4. Um den Methodenaufruf an alle Replikate weiterzuleiten, erzeugt der Primary eine Instanz der Klasse ACTIVEREPLICATIONCALLMANAGER, an welcher er anschließend die Methode `callMethod()` aufruft.

5. Die Instanz leitet den Aufruf dann an alle Replikate weiter, indem sie an ihren Strategie-Objekten die Methode `doReplicaCall()` aufruft.

6. Die Strategie-Objekte rufen dann die Methode an den Replikaten auf.

7. Das Ergebnis des Aufrufs wird, nachdem alle Replikate die Ausführung beendet haben, auf dem gleichen Weg wieder zurückgegeben. Falls im Rückgabewert Objekte, die als Kopie übertragen würden, enthalten sind, werden diese in Remote-Objekte umgewandelt.

Zur Verdeutlichung des Ablaufs kann noch einmal Abbildung 4.1 herangezogen werden.

4.2.4.1 Verschachtelter Methodenaufruf

Ein verschachtelter Methodenaufruf kann während eines Methodenaufrufs nur an Position 6. erfolgen. Bei Auftreten eines solchen wird wie folgt vorgegangen:

6.1. Der verschachtelte Methodenaufruf wird vom Laufzeitsystem abgefangen, und an das Strategie-Objekt des Replikates, in dem er auftrat, weitergeleitet. Dies geschieht, indem die Methode `outCall()` des Strategie-Objektes aufgerufen wird.

6.2. In dieser Methode wird überprüft, ob es sich bei dem Objekt, an dem der verschachtelte Methodenaufruf erfolgen soll, um ein System-Objekt handelt. In diesem Fall wird er direkt am System-Objekt aufgerufen und nicht an den Primary weitergeleitet, da es sich bei System-Objekten um prozeßspezifische Komponenten handelt.

6.3. Im anderen Fall wird der Aufruf an den Primary weitergeleitet, indem an ihm die Methode `outCallFromReplica()` aufgerufen wird.

6.4. Erst nachdem alle Replikate die Aufrufe gemeldet haben, vereinigt der Primary diese zu einem Aufruf und führt diesen aus. Da durch die Meldungen der Replikate über den verschachtelten Methodenaufruf alle Aktivitätsträger die Replikate verlassen haben, kann während der Ausführung eines verschachtelten Methodenaufrufs ein neuer Methodenaufruf an dem replizierten Objekt zugelassen werden.

6.5. Das Ergebnis des verschachtelten Methodenaufrufs wird auf dem gleichen Weg wieder zurückgegeben. Falls im Rückgabewert Objekte, die als Kopie übertragen würden, enthalten sind, werden diese in Remote-Objekte umgewandelt.

4.2.4.2 Erzeugen neuer Objekte

Die Erzeugung neuer Objekte kann innerhalb eines Replikates wie der verschachtelter Methodenaufruf nur an Position 6 eines Methodenaufrufs erfolgen. Der Ablauf hierzu ist dann wie folgt:

6.1. Die Erzeugung eines neuen Objektes wird vom Laufzeitsystem abgefangen, und an das Strategie-Objekt des Replikates, in dem er auftrat, weitergeleitet. Dies geschieht, indem die Methode `newObject()` des Strategie-Objektes aufgerufen wird.

6.2. Das Strategie-Objekt benachrichtigt den Primary, daß ein neues Objekt erzeugt werden soll, indem es an ihm die Methode `generateNewObject()` aufruft.

6.3. Erst nachdem alle Replikate die Erzeugung eines neuen Objektes gemeldet haben, erzeugt der Primary ein neues repliziertes Objekt. Dies geschieht, indem er zuerst ein Basis-Objekt instantiiert und danach eine Strategie-Objekt zur Replikation daran bindet.

6.4. Nachdem der Primary das neue Objekt erzeugt hat, gibt er an jedes Replikat ein Replikat des neuen Objektes zurück.

6.5. Die einzelnen Replikate verwenden das empfangene neue Replikat als neues Objekt.

4.2.5 Sperr-Mechanismus

Als Sperr-Mechanismus kann der unter Abschnitt 3.2.2.5 für Remote-Objekte vorgestellte verwendet werden. Hierbei ist der Primary als Remote-Objekt und alle anderen Replikate als Stubs zu sehen. Die unter Abschnitt 3.3.3 vorgestellte fehlertolerante Erweiterung kann natürlich auch verwendet werden. Allerdings muß hier darauf geachtet werden, daß sie mit der durch die Replikationsstrategie verwirklichten, in Abschnitt 4.2.6 vorgestellten, Fehlertoleranz zusammenpasst.

4.2.6 Fehlertoleranz

Die von RepliXa implementierte aktive Replikationsstrategie ist fehlertolerant gegenüber Netzwerkpartitionen und Ausfällen einzelner Prozesse (außer dem Primary). Der Primary und die anderen Replikate erreichen dies durch folgendes Verhalten:

Primary
Falls während der Kommunikation mit einem Replikat beim Primary ein Fehler auftritt, so entfernt er dies aus seiner Liste mit gültigen Replikaten. Dies hat zur Folge, daß der Primary nicht mehr versucht, zu diesem Replikat Kontakt herzustellen. Falls der Prozeß, in dem das Replikat läuft, nicht ausgefallen ist, sondern nur wegen einer Netzwerkpartition nicht erreichbar ist, so meldet sich das Replikat, nachdem die Netzwerkpartition nicht mehr besteht, irgendwann beim Primary und wird dadurch wieder in die Liste aufgenommen. Ein neu angelegtes Replikat wird ebenfalls in die Liste aufgenommen.

Andere Replikate
Den Objektzustand nicht verändernde Methoden werden grundsätzlich immer ohne Kontakt zum Primary zugelassen. Falls ein Replikat während eines verändernden Methodenaufrufs feststellt, daß der Primary nicht erreichbar ist, markiert es den Zustand des Basis-Objektes als ungültig und gibt einen Fehler an das Laufzeitsystem zurück. Von da an wird bei jedem verändernden Methodenaufruf versucht, Kontakt zum Primary herzustellen. Falls dies gelingt, wird ein gültiger Zustand für das Basis-Objekt angefordert und die Methode wie gewohnt an den Primary weitergeleitet. Falls nicht, wird ein Fehler zurück gegeben.

Ein Nachteil dieser Replikationsstrategie ist, daß der Ausfall des Primary den Ausfall des gesamten replizierten Objektes zur Folge hat, also ein *Single Point of Failure* vorliegt. Eine Möglichkeit, die Existenz eines Primary zu vermeiden, wird in Abschnitt 4.3.2 vorgestellt.

4.3 Erweiterungen

In diesem Abschnitt sollen einige mögliche Erweiterungen, die zum Teil aus Zeitgründen nicht implementiert werden konnten, vorgestellt werden.

4.3.1 Garbage Collection

Die Replikationsstrategie sollte noch um einen Garbage Collection Algorithmus ergänzt werden. Hierzu kann der unter Abschnitt 3.3.5 vorgestellte Algorithmus verwendet werden, wenn der Primary als Remote-Objekt und die anderen Replikate als Stubs gesehen werden.

4.3.2 Vermeidung eines Primary

Wie unter Abschnitt 4.2.6 erläutert wurde, stellt der Primary einen *Single Point of Failure* dar. Eine Möglichkeit diesen *Single Point of Failure* zu vermeiden ist, daß auf die Existenz eines Primary verzichtet wird. Dies wiederum bedeutet, daß eine Möglichkeit gefunden werden muß, die totale Ordnung der Methodenaufrufe an den Replikaten auch ohne Primary zu gewährleisten.

Eine Möglichkeit hierzu ist, einen Methodenaufruf, der an einem Replikat stattfindet, über einen Broadcast-Mechanismus direkt an alle weiterzuleiten. Der Broadcast-Mechanismus muß in diesem Fall verläßlich sein und die totale Ordnung gewährleisten. Es gibt zwei Möglichkeiten eine solchen Broadcast-Mechanismus zu erhalten. Die eine wäre, ihn selbst zu implementieren, wie es beispielsweise in [Sch90] beschrieben ist. Die andere wäre, Systeme wie Isis, Horus oder Transis, welche solche Broadcast-Mechanismen zur Verfügung stellen, zu verwenden. Dieser Ansatz wird beispielsweise von [GGM95] verwendet.

Kapitel 5

Passive Replikation

Das Ziel der passiven Objektreplikation ist, wie auch bei der aktiven Objektreplikation, eine Menge von Objekten in einem konsistenten Zustand zu halten. Bei der passiven Objektreplikation wird dies jedoch auf eine völlig andere Weise erreicht: Methodenaufrufe können an jedem Replikat direkt ausgeführt werden, und die dadurch hervorgerufenen Veränderungen werden an alle anderen Replikate weitergegeben. Aufgabe der Replikationsstrategie ist es darauf zu achten, daß die vorübergehend entstehenden Inkonsistenzen keinerlei Einfluß auf das Gesamtsystem haben.

Diese Vorgehensweise ist vergleichbar mit einem DSM-System, bei dem mehrere Prozesse auf eine gemeinsam genutzte Speicherseite zugreifen (der Speicherseite entspricht der interne Zustand des Objektes), weswegen im Rahmen von DSM-Systemen entwickelte Algorithmen verwendet werden können.

Im nächsten Abschnitt wird zunächst ein Algorithmus aus dem Bereich der DSM-System vorgestellt, da er die Grundlage für die von RepliXa bereitgestellte Replikationsstrategie bildet. Anschließend werden die Probleme erläutert, die bei der Anpassung des Algorithmus, so daß er zur Objektreplikation verwendet werden kann, entstanden sind. Danach wird die von RepliXa bereitgestellte passive Replikationsstrategie näher erläutert, und zum Schluß erfolgt noch eine Beschreibung möglicher Erweiterungen. Ein Beispielprogramm zur Verwendung der passiven Replikation ist in Anhang C zu finden.

5.1 Der Algorithmus

Da sich die passiven Objektreplikation von RepliXa auf einen aus dem DSM-Bereich bekannten Algorithmus stützt, wird dieser hier kurz erläutert. Eine genauere Beschreibung und ein Beweis für die korrekte Funktionsweise des Algorithmus sind in [BW94] zu finden.

Da es sich bei dem Algorithmus um eine fehlertolerante Erweiterung des *Dynamic Distributed Manager DSM* Algorithmus handelt, wird zuerst die Funktionsweise dieses Algorithmus und anschließend die fehlertolerante Erweiterung beschrieben.

5.1.1 Dynamic Distributed Manager DSM

Der Dynamic Distributed Manager Algorithmus verwendet ein *Write-Invalidate* Protokoll[1], um strenge Konsistenz[2] innerhalb eines gemeinsam genutzten verteilten Speicher zu erreichen. Es wird vorausgesetzt, daß die Rechner, welche die Seiten des gemeinsam genutzten verteilten Speichers enthalten, eine identische Hardware-Architektur aufweisen und über zuverlässige Kommunikationskanäle logisch vollständig miteinander verbunden sind.

Die Grundidee des Algorithmus ist nun folgende: Jede Seite des Speichers hat einen Eigentümer (ein beliebiger Rechner), im Folgenden bezeichnet als Primary, der sich dynamisch ändern kann, und für jede Seite existieren zwei Zugriffsmöglichkeiten: Entweder mehrere Rechner haben ein Lese-Recht oder der Primary hat ein Schreib-Lese-Recht. Falls ein Rechner auf eine Seite zugreifen möchte, muß er die entsprechenden Zugriffsrechte anfordern, falls er sie nicht bereits besitzt. Beim Anfordern der Zugriffsrechte wird wie folgt vorgegangen:

Anfordern des Lese-Rechtes
Wenn ein Rechner lesend auf eine Speicherseite zugreifen möchte, so fordert er eine gültige Kopie dieser Seite beim Primary an. Der Primary verliert dadurch, falls nicht schon vorher geschehen, sein Schreib-Recht auf die Seite und hat folglich nur noch ein Lese-Recht.

Anfordern des Schreib-Lese-Rechtes
Wenn es sich um den Primary handelt, der das Lese-Schreib-Recht anfordert, so erklärt er alle herausgegebenen Kopien der Seite als ungültig. Dies geschieht, indem er an alle Rechner, welche eine Kopie der Seite empfangen haben, eine entsprechende Nachricht sendet. Da der Primary jetzt der einzige Rechner ist, der ein gültige Kopie der Seite besitzt, kann Schreib-Lese-Zugriff zugelassen werden. Falls es sich jedoch nicht um den Primary handelt, der das Schreib-Lese-Recht anfordert, so veranlaßt der anfordernde Rechner den Primary, ihn zum neuen Primary zu ernennen. Nachdem der anfordernde Rechner der neue Primary geworden ist, verfährt er wie gewohnt, um das Schreib-Lese-Recht zu erhalten.

Wie aus den obigen Abläufen leicht ersichtlich, muß zum Anfordern der entsprechenden Zugriffsrechte immer der Primary kontaktiert werden. Da dieser sich jedoch dynamisch ändert, müßten nach jeder Änderung alle Rechner informiert werden, welcher der neue Primary ist. Da dies aber aus Gründen der Effizienz nicht vertretbar ist, wird folgender Ansatz gewählt:

- Der Verweis auf den Primary einer Seite, der ja mit jeder Seite gespeichert wird, verweist nur wahrscheinlich auf den Primary, d.h. der Rechner, auf den er verweist, ist entweder der Primary oder kann die Nachricht in Richtung des Primary weiterleiten (über seinen Verweis auf den nächsten wahrscheinlichen Primary).
- Bei einem Wechsel des Primary ändert der alte Primary den Verweis, der vorher auf ihn selbst verwies, so ab, daß er auf den neuen Primary verweist. Dadurch kann er alle Anfragen, die fälschlicherweise an ihn gesendet werden, in Richtung des neuen Primary weiterleiten.

[1] Bei einem *Write-Invalidate* Protokoll werden alle Kopien außer einer eines gemeinsam genutzten Datenbereichs als ungültig markiert, bevor auf diese eine geschrieben werden kann. Siehe hierzu auch [SS94].

[2] Strenge Konsistenz bedeutet, daß jeder Lesevorgang den Wert zurück gibt, der beim letzten Schreibvorgang gespeichert wurde. Nähere Erläuterungen sind zu finden in [Tan95].

- Bei allen Nachrichten die der Primary versendet, sendet er seine Identität mit, so daß die empfangenden Rechner den Verweis auf den wahrscheinlichen Primary aktualisieren können. Dies ist nötig, um die Ketten der wahrscheinlichen Primarys, die zum Auffinden des tatsächlichen Primary durchlaufen werden müssen, möglichst kurz zu halten.

5.1.2 Integrated Snooper Fault-Tolerant DSM

Das Problem des *Dynamic Distributed Manager* System ist, daß es bei Ausfall eines Rechners, der sich in einer Kette mit wahrscheinlichen Primarys befindet, teilweise und bei Ausfall des Primary komplett ausfällt. Deshalb wurde von Brown und Wu (siehe [BW94]) eine fehlertolerante Erweiterung entworfen, die trotz Ausfall eines beliebigen Rechners folgendes gewährleistet:

- Es existiert zu jedem Zeitpunkt eine vollständige und gültige Kopie jeder Seite im System.
- Der Primary ist zu jedem Zeitpunkt für alle Rechner erreichbar.

Im Rahmen dieser Erweiterung wurde ein sog. Snooper (ein beliebiger Rechner, jedoch nicht der Primary) eingeführt, dessen Aufgabe es ist, alle Aktionen des Primary zu protokollieren. Zu diesem Zweck wird er vom Primary informiert, wenn dieser den Inhalt der Seite ändert, eine Kopie der Seite herausgibt oder einen neuen Rechner zum Primary ernennt. Dadurch wird erreicht, daß der Snooper den Inhalt der Seite vollständig kennt und über alle existierenden Kopien der Seite Bescheid weiß. Da der Snooper also über die gleichen Informationen wie der Primary verfügt, kann er bei einem Ausfall des Primary dessen Aufgaben übernehmen.

Durch das Protokollieren des Snoopers existieren im System zwei vollständige gültige Kopien einer Seite. Dies ist ausreichend, um die Forderung, daß zu jedem Zeitpunkt eine vollständige und gültige Kopie einer Seite im System existiert, zu erfüllen, da die Rechner als fehlerunabhängig angesehen werden, und deshalb die Wahrscheinlichkeit, daß beide Kopien der Seite verloren gehen sehr gering ist. Jedoch muß darauf geachtet werden, daß zu keinem Zeitpunkt ein Rechner Primary und gleichzeitig Snooper der selben Seite ist, da die beiden Kopien dann nicht mehr fehlerunabhängig wären. Dies hat zur Folge, daß sich der Snooper auch dynamisch ändern muß, wenn der Rechner der gerade Snooper ist, Primary werden muß, weil er auf die Seite Schreib-Lese-Zugriff benötigt.

Damit der Snooper einer Seite von jedem Rechner aus erreichbar ist, speichert jeder Rechner eine Verweis auf den Snooper jeder Seite. Da sich der Snooper dynamisch ändert, wird das gleiche Verfahren wie beim Primary angewandt. Die Rechner speichern also nicht den tatsächlichen, sondern nur den wahrscheinlichen Snooper.

Falls ein Rechner eine eigene Anfrage oder ein Anfrage, die er weiterleiten soll, nicht an den wahrscheinlichen Primary weiterleiten kann, so leitet er sie an den wahrscheinlichen Snooper weiter. Es sind zwei Gründe denkbar, weswegen in manchen Fällen eine Anfrage nicht an den wahrscheinlichen Primary weitergeleitet werden kann:

- Der eine ist, daß durch den Ausfall eines Rechners eine Lücke in der Kette der wahrscheinlichen Primarys entstanden ist. Diese Lücke wird dann durch das Weiterleiten entlang der Kette der wahrscheinlichen Snooper umgangen, so daß der Primary trotz der Lücke erreicht wird. Beweis siehe [BW94]. (Falls in der Kette der wahrscheinlichen Snooper eine Lücke auftritt, wird wieder die Kette der wahrscheinlichen Primarys verwendet.)

- Der andere ist, daß der Primary selbst ausgefallen ist. In diesem Fall gelangt die Anfrage über die Kette der wahrscheinlichen Snooper zum tatsächlichen Snooper, welcher den Ausfall des Primary feststellt und daraufhin dessen Aufgaben übernimmt. Da der Snooper dadurch zum neuen Primary geworden ist, hat die Anfrage schließlich doch ihr Ziel erreicht.

Durch diese Vorgehensweise wird also erreicht, daß selbst nach Ausfall eines beliebigen Rechners der Primary, bzw. bei einem Ausfall des Primary sein Vertreter, der Snooper, noch erreichbar sind. Dadurch wird die Forderung, daß der Primary zu jedem Zeitpunkt für alle Rechner erreichbar ist, erfüllt.

Nachdem ein neuer Snooper bestimmt und initialisiert wurde und die Lücken in den Ketten der wahrscheinlichen Primarys und Snooper geschlossen wurden, verkraftet das System den Ausfall eines weiteren Rechners.

5.2 Aufgetretene Probleme und ihre Lösungen

Bei der Anpassung des Algorithmus, so daß er zur Objektreplikation verwendet werden kann, ergaben sich zwei Probleme. Das eine Problem in Zusammenhang mit der Frage, wann der Primary den Snooper über Veränderungen an der Speicherseite informiert, und das andere in Zusammenhang mit der Behandlung von Teilobjekten.

5.2.1 Informieren des Snoopers

Bei dem Algorithmus für DSM-Systeme wird dabei folgendermaßen vorgegangen: Um eine gute Performance des Systems zu erreichen, wird der Snooper nur dann informiert, wenn dies zwingend erforderlich ist. So lange also die Änderungen, die der Primary an einer Seite durchführt, nicht für andere Rechner sichtbar werden, muß der Snooper nicht über den neuen Inhalt der Seite informiert werden. Wenn also der Primary die Seite verändert und danach ausfällt, so sind die Änderungen zwar verloren, aber für das System sieht es so aus, als ob der Primary schon vor der Modifikation ausgefallen wäre.

Die Änderungen an der Seite können auf zwei verschiedene Arten für die anderen Rechner sichtbar werden: Die eine ist, daß eine Kopie der Seite an einen anderen Rechner gesandt wird, oder ein anderer Rechner zum Primary ernannt wird. Die andere ist, daß vom neuen Zustand der Seite abhängige Informationen an andere Rechner übertragen werden. Es kann also zusammenfassend gesagt werden, daß, sobald ein Rechner Informationen an andere Rechner weiter gibt, der neue Zustand aller geänderter Seiten an die jeweiligen Snooper übertragen werden muß.

Bei einem DSM-System ist dies sehr einfach zu realisieren, da die einzige Möglichkeit (die bei der Konzeption des Algorithmus in Betracht gezogen wurde), Informationen an andere Rechner weiterzugeben, das DSM-System selbst ist. Das DSM-System kann also selbst darauf achten und falls dies nötig ist, die Snooper der Seiten informieren.

Bei dem Versuch, den Algorithmus zur passiven Objektreplikation (hier entsprechen den Rechnern Prozesse und den Speicherseiten Objektzustände) zu verwenden, tritt das Problem auf, daß bei RepliXa noch andere Möglichkeiten bestehen, Informationen an andere Prozesse weiterzugeben. Ein Beispiel hierfür ist der RMI-Mechanismus. Es ist der passiven Replikationsstrategie ohne Einflußnahme auf diese anderen Mechanismen also nicht möglich, die Snooper wie oben beschrieben über Änderungen zu informieren.

Da eine derartige Einflußnahme nicht gewollt und auch nicht immer möglich ist, wurde bei der passiven Replikationsstrategie ein anderer Weg eingeschlagen. Es wird dem Programmierer des Systems überlassen, eine für ihn passende Snooper-Informationsstrategie zu wählen. Der Programmierer kann bei der Konstruktion des replizierten Objektes aus folgenden zwei Informationsstrategien wählen:

- Der Snooper wird nach jeder Änderung informiert. Diese Strategie hat den Vorteil, daß sie das gleiche Verhalten wie bei DSM-Systemen nachbildet, jedoch eine schlechtere Performance mit sich bringt.

- Der Snooper wird nur informiert, wenn der Primary den neuen Inhalt der Seite bewußt weitergibt. Diese Strategie erzielt eine wesentlich bessere Performance, hat jedoch den Nachteil, daß Inkonsistenzen zwischen den einzelnen Objekten (nicht innerhalb der Replikate eines Objektes, sondern zwischen Objekten) des Systems entstehen können.

Der Programmierer kann also die Strategie mit der besseren Performance wählen, muß jedoch in seinem Programm darauf achten, daß eventuell auftretende Inkonsistenzen keinen Einfluß haben.

5.2.2 Teilobjekte

Wenn in dem replizierten Objekt Verweise auf andere Objekte gespeichert sind, so werden diese anderen Objekte als Teilobjekte (oder auch implizit replizierte Objekte, wie sie in Abschnitt 4.1.4 eingeführt wurden) bezeichnet, da sie als Teil des replizierten Objektes gesehen werden. Diese Teilobjekte müssen natürlich auch in einem konsistenten Zustand gehalten werden.

Da bei DSM-Systemen als Teilobjekte nur Objekte in Betracht gezogen werden, die selbst auf Seiten des DSM-Systems gespeichert sind, tritt das Problem der Konsistenzhaltung hier nicht auf, da die Teilobjekte selbst replizierte Objekte sind.

Bei RepliXa jedoch könnte es sich bei den Teilobjekten um beliebige Objekte handeln, weswegen der Zugriff auf die Teilobjekte, ähnlich wie bei der aktiven Replikation in Abschnitt 4.1.4 geschildert, nicht immer überwacht werden kann. Zur Lösung dieses Problems gibt es zwei Möglichkeiten: Die eine ist, um das Replikat und seine Teilobjekte eine Hülle zu bilden, wie dies in Abschnitt 4.1.4 beschrieben ist, und komplett zu replizieren. Die andere, von RepliXa gewählte Möglichkeit ist, als Teilobjekte nur Remote-Objekte und replizierte Objekte zuzulassen, da diese selbst für ihre Konsistenz sorgen. Wie dies erreicht werden kann, ist ebenfalls in Abschnitt 4.1.4 beschreiben.

5.3 Realisierung

In diesem Abschnitt wird die Implementierung der im Rahmen von RepliXa verwirklichten aktiven Replikationsstrategie näher betrachtet. Nachdem die Grundidee vorgestellt wurde, wird zunächst darauf eingegangen, wie aus einem Objekt ein repliziertes Objekt entsteht, und wie die

Installation neuer Replikate vor sich geht. Danach wird der Mechanismus zum Weiterleiten von Nachrichten zum Primary erläutert. Der Ablauf bei der Anforderung von Lese- und Schreib-Lese-Zugriff wird im Anschluß daran erläutert. Zum Schluß werden die Maßnahmen bei einem Rechnerausfall und der Sperr-Mechanismus näher betrachtet.

5.3.1 Grundidee

An jedes Replikat eines replizierten Objektes ist ein Strategie-Objekt gebunden. Bei dem Strategie-Objekt handelt es sich um Instanzen der Klasse PASSIVEREPLICATION. Einem dieser Strategie-Objekte fällt die Rolle des Primary zu, einem anderen die des Snoopers. Die Strategie-Objekte kommunizieren untereinander mit Hilfe des RMI-Mechanismus.

Die Strategie-Objekte werden vom Laufzeitsystem über folgende Ereignisse am Basis-Objekt informiert:

- Zugriff auf den internen Objektzustand, um gegebenenfalls die entsprechenden Zugriffsrechte anzufordern (siehe Abschnitte 5.3.5 und 5.3.6).
- Sperr- und Freigabe-Events, um sie zu bearbeiten (siehe Abschnitt 5.3.8).

Der Zustand eines replizierten Objektes ist entweder Lese-Zugriff für alle, wobei in diesem Fall automatisch alle Replikate eine gültige Kopie besitzen, oder Lese-Schreib-Zugriff für den Primary. Die Vorgehensweise, daß bei Lese-Zugriff automatisch alle Replikate eine gültige Kopie erhalten, ist eine Abweichung vom eigentlichen Algorithmus für DSM-Systeme, auf die in Abschnitt 5.4.1 noch einmal eingegangen wird.

5.3.2 Initialisierung

Um ein Basis-Objekt zu replizieren muß zuerst eine Instanz des Strategie-Objektes PASSIVEREPLICATION erzeugt werden, wobei angegeben werden muß, welche Snooper-Informationsstrategie verwendet werden soll. Dieses Strategie-Objekt muß dann an das Basis-Objekt gebunden werden. Es wird dadurch automatisch zum Primary, und der Prozeß hat Schreib-Lese-Zugriff auf das Objekt. Während der Bindephase werden alle Objekte, auf die das Basis-Objekt verweist und die als Kopie übertragen würden zu Remote-Objekten transformiert, indem eine Instanz der Klasse ACCESSLOCALTRANSMITROR daran gebunden wird.

5.3.3 Installation eines neuen Replikates

Die Notwendigkeit, in einem Prozeß ein neues Replikat anzulegen besteht dann, wenn der Prozeß es bei einem Nameservice anfordert oder es im Rahmen einer RMI als Parameter oder Rückgabewert übergeben wird. Der Ablauf ist dann wie folgt:

1. Durch das Grundsystem wird die Methode `transmitObject()` an einem Strategie-Objekt eines Replikates aufgerufen.
2. In dieser Methode wird zunächst ein Installationsobjekt (eine Instanz der Klasse ACTIVEREPLICATIONINSTALLER) erzeugt und initialisiert. Danach wird dieses Installationsobjekt mit Hilfe der Methode `sendObjectReplica()` der Klasse OBJECTTRANSMISSIONCONTROL an den Zielprozeß übertragen.

3. Das REPLICATIONENVIRONMENT des Zielprozesses empfängt das Installationsobjekt und ruft daran die Methode `installReplica()` auf. Alle weiteren Schritte werden von dieser Methode veranlasst.

4. Nun wird überprüft, ob in diesem Prozeß bereits ein Replikat vorhanden ist. Ist dies der Fall, so wird das Replikat zurückgegeben und die Installation ist beendet.

5. Jetzt wird eine Instanz des Basis-Objektes gebildet und ein Strategie-Objekt, welches vorher ebenfalls instantiiert und initialisiert wurde, daran gebunden.

6. Zum Schluß registriert sich das Replikat noch beim Primary, indem es diesen über die Methode `forwardToPrimary()` (welche in Abschnitt 5.3.4 näher erläutert wird) kontaktiert. Der Primary trägt das neue Replikat in eine Liste ein, in der alle Replikate verzeichnet sind.

5.3.4 Weiterleiten zum Primary

Nachrichten zwischen den einzelnen Strategie-Objekten werden dadurch realisiert, daß das sendende Strategie-Objekt an dem empfangenden eine Methode aufruft. Da verschiedene Nachrichten, wie zum Beispiel Registrieren eines neuen Replikates oder Anfordern des Lese-Zugriffs, über wahrscheinliche Primarys bzw. Snooper zum tatsächlichen Primary bzw. Snooper weitergeleitet werden müssen, wurde hierfür ein einheitlicher Mechanismus geschaffen. Der Ablauf zum Weiterleiten an den Primary funktioniert also wie folgt:

1. Wenn ein Strategie-Objekt eine Nachricht an den Primary schicken will, so ruft es nicht die entsprechende Methode direkt auf, sondern die Methode `forwardToPrimary()`, der sie als Parameter eine Kennung der tatsächlich gewünschten Methode übergibt.

2. Die Methode `forwardToPrimary()` prüft, ob es sich bei dem aktuellen Strategie-Objekt um den Primary handelt. Sollte dies der Fall sein, ruft sie die gewünschte Methode auf, und das Weiterleiten ist beendet.

   - Die Methode `registerNewReplica()` bei registrieren eines neuen Replikates.
   - Die Methode `getReadAccess()` bei anfordern des Lese-Zugriffs (siehe Abschnitt 5.3.5).
   - Und die Methode `getReadWriteAccess()` bei anfordern des Schreib-Lese-Zugriffs (siehe Abschnitt 5.3.6).

3. Ansonsten prüft die Methode, ob es sich um den Snooper handelt. Ist dies der Fall, wird abhängig davon, ob der Primary noch funktionsfähig ist, entweder die Nachricht an diesen weitergeleitet oder der Snooper wird, wie in Abschnitt 5.3.7 beschrieben, zum neuen Primary und verfährt dann wie unter 2. beschrieben.

4. Nun, da es sich bei dem Strategie-Objekt weder um den Primary noch um den Snooper handelt, wird versucht die Nachricht in Richtung des wahrscheinlichen Primary weiter zu geben. Wenn dies nicht möglich ist, wird versucht, sie in Richtung des wahrscheinlichen Snoopers weiterzugeben.

5. Wenn die Nachricht nicht weitergegeben werden konnte, so sind mehr als ein Prozeß ausgefallen und es wird ein Fehler zurückgegeben.

5.3.5 Anfordern von Lese-Zugriff

Lese-Zugriff wird angefordert, wenn ein lesender Zugriff auf den internen Zustand des Replikates abgefangen wurde und der Prozeß keine gültige Kopie des replizierten Objektes besitzt. Der Ablauf der Anforderung ist wie folgt:

1. Ein lesender Zugriff an einem Replikat wird vom Laufzeitsystem abgefangen, welches dann die Methode `readField()` an dem zu diesem Replikat gehörenden Strategie-Objekt aufruft.

2. In dieser Methode wird zuerst geprüft, ob bereits eine gültige Kopie vorhanden ist, und nur falls nicht, wird mit Hilfe der Methode `forwardToPrimary()` beim Primary Lese-Zugriff angefordert.

3. Nachdem die Anforderung beim Primary eingegangen ist, prüft dieser, ob bereits Lese-Zugriff für alle besteht. Dies könnte der Fall sein, da bereits eine andere Anforderung dies bewirkt haben könnte. Wenn bereits Lese-Zugriff für alle besteht, so kann die Anforderung ignoriert werden, da in diesem Fall das anfordernde Replikat bereits eine gültige Kopie erhalten hat.

4. Falls jedoch noch Schreib-Lese-Zugriff besteht, so wechselt der Primary die Zugriffsart von Schreib-Lese-Zugriff zu Lese-Zugriff für alle. Dies geschieht, indem er zuerst den Snooper über den bevorstehenden Wechsel informiert, dann allen Replikaten eine gültige Kopie sendet und zum Schluß die Zugriffsart in Lese-Zugriff für alle ändert. Das Informieren des Snoopers, das Senden einer gültigen Kopie an alle Replikate und das Ändern der Zugriffsart, erfolgt, indem die Methode `setNewState()` am Snooper, den Replikaten und am Primary aufgerufen wird.

5.3.6 Anfordern von Schreib-Lese-Zugriff

Schreib-Lese-Zugriff wird angefordert, wenn ein schreibender Zugriff auf den internen Zustand des Replikates abgefangen wurde und der Prozeß nicht der Primary ist, oder zwar der Primary ist, aber kein Schreib-Lese-Recht hat. Der Ablauf der Anforderung ist wie folgt:

1. Ein Schreib-Zugriff an einem Replikat wird vom Laufzeitsystem abgefangen, welches dann die Methode `writeField()` am zu diesem Replikat gehörenden Strategie-Objekt aufruft.

2. In dieser Methode wird die Anfrage an den Primary weitergeleitet.

3. Der Primary prüft zuerst, ob bisher Schreib-Zugriff für alle bestand. Ist dies der Fall, so wechselt er die Zugriffsart von Lese-Zugriff zu Schreib-Lese-Zugriff. Dies geschieht, indem er zuerst den Snooper über den bevorstehenden Wechsel informiert, dann allen Kopien als ungültig erklärt und zum Schluß die Zugriffsart in Schreib-Lese-Zugriff ändert.

4. Wenn der Primary nicht den Schreib-Lese-Zugriff angefordert hat, so muß noch ein Wechsel des Primary erfolgen. Dies geschieht, indem der alte Primary zuerst den Snooper über den bevorstehenden Wechsel informiert, dann den neuen Primary benachrichtigt und zum Schluß seinen eigenen Zustand anpasst.

5.3.7 Ausfälle und Recovery

Abhängig von der Art des ausgefallenen Prozesses werden unterschiedliche Recovery-Maßnahmen eingeleitet. Es wird unterschieden zwischen einem Ausfall des Primary, des Snoopers und eines gewöhnlichen Replikates.

Der Ausfall des Snoopers oder eines gewöhnlichen Replikates wird vom Primary erkannt. Bei Ausfall des Snoopers wird vom Primary ein neuer bestimmt und initialisiert. Bei Ausfall eines gewöhnlichen Replikates wird dies einfach aus der Liste der Replikate ausgetragen.

Falls der Primary ausfällt, so erkennt der Snooper dies und leitet eine von der letzten Aktivität des Primary abhängige Recovery Maßnahme ein:

- Falls der Primary im Begriff war, Lese-Zugriff für alle einzurichten, so wir dies vollendet, indem der Snooper sich selbst zum Primary ernennt, einen neuen Snooper bestimmt und allen Replikaten eine gültige Kopie sendet.
- Wenn der Primary im Begriff war, Schreib-Lese-Zugriff einzurichten, so wird dies rückgängig gemacht, da der Primary ausgefallen ist und deshalb den Schreib-Lese-Zugriff nicht mehr benötigt. Dies geschieht, indem sich der Snooper selbst zum Primary ernennt, einen neuen Snooper bestimmt und alle Kopien für gültig erklärt.
- Falls sich gerade ein Wechsel des Primary vollzog, so könnte es sein, daß der alte Primary ausgefallen ist, bevor der neue Primary darüber benachrichtigt wurde, daß er der neue Primary ist. Der Snooper informiert deshalb den neuen Primary noch einmal darüber.
- Befand sich der Primary in keiner der obigen kritischen Phasen, so kann der Snooper einfach seine Rolle übernehmen, indem er sich selbst zum Primary ernennt und einen neuen Snooper bestimmt.

Eventuell entstandene Lücken in den Ketten der wahrscheinlichen Primarys bzw. Snooper schließen sich automatisch, wenn die darauf verweisenden Rechner aktuellere Informationen erhalten.

Durch diese Recovery-Maßnahmen wird erreicht, daß das System den Ausfall eines Prozesses toleriert. Nach einer gewissen Regenerationszeit, ist das System dann im Stande den nächsten Ausfall zu tolerieren.

5.3.8 Sperr-Mechanismus

Der Sperr-Mechanismus der passiven Replikation basiert auf einer einfachen Grundidee: Es können niemals zwei Prozesse gleichzeitig Primary sein. Das Sperren eines passiv replizierten Objektes erfolgt deshalb in zwei Schritten: Im ersten Schritt wird der Prozeß, von dem aus die Sperre angefordert wird, zum Primary, um den gegenseitigen Ausschluß zwischen Prozessen zu gewährleisten. Im zweiten Schritt wird der Java eigene Sperr-Mechanismus verwendet, um den gegenseitigen Ausschluß zwischen den Aktivitätsträgern innerhalb eines Prozesses zu realisieren.

Dieser Sperr-Mechanismus hat den Nachteil, daß ein Aktivitätsträger, der das Objekt von einem Prozeß aus belegt, dann mit Hilfe des RMI-Mechanismus in einen anderen Prozeß einwandert und hier versucht, das Objekt erneut zu belegen, dies nicht kann, obwohl es möglich sein sollte. Der Grund hierfür liegt darin, daß es nicht möglich ist, daß zwei Prozesse gleichzeitig Primary sind.

Er hat jedoch den großen Vorteil, daß er fehlertolerant ist. Wenn der Aktivitätsträger, der das Objekt belegt, also ausgefallen ist, so wird dies automatisch freigegeben. Dies ist der Fall, da die einzige Möglichkeit des Aktivitätsträgers auszufallen darin besteht, daß der Prozeß, in dem er sich befindet, ausfällt. Es sind nun zwei Fälle zu unterscheiden:

1. Der Aktivitätsträger befindet sich nicht mehr in dem Prozeß, von dem aus er das Objekt belegt hat. In diesem Fall hat ein Ausfall des Aktivitätsträgers nur die Rückgabe eines Fehlers durch den RMI-Mechanismus zur Folge. Dies wiederum bedeutet, daß der Aktivitätsträger in dem Prozeß, von dem aus die RMI erfolgte, weiterläuft.

2. Der Aktivitätsträger befindet sich in dem Prozeß, von dem aus er das Objekt belegt hat. Wie bereits festgestellt besteht die einzige Möglichkeit des Aktivitätsträgers auszufallen darin, daß der ganze Prozeß ausfällt, wodurch alle Informationen, die die Sperre betreffen, verloren gehen, wodurch das Objekt automatisch frei gegeben wird.

Im Zusammenhang mit dem Sperr-Mechanismus muß die in Abschnitt 5.2.1 diskutierte Frage, wann der Snooper über Änderungen an der Seite vom Primary informiert werden sollte, noch einmal aufgegriffen werden. Da eine Sperre, wie bereits in Abschnitt 3.3.3 erläutert, unter anderem dazu verwendet wird, ein Objekt von einem konsistenten Zustand in einen anderen zu überführen, muß daß Objekt in seinen ursprünglichen Zustand zurückgeführt werden, falls die Sperre in Folge eines Fehlers aufgehoben wird. Dieses Verhalten wird dadurch erreicht, daß der Snooper vom Primary nicht informiert wird, solange die Sperre besteht. Hierdurch können jedoch die in Abschnitt 5.2.1 diskutierten Probleme, im Zusammenhang mit der Tatsache, daß zeitweise Änderungen für andere Prozesse sichtbar werden könnten, auftreten.

5.4 Erweiterungen

In diesem Abschnitt sollen einige mögliche Erweiterungen, die zum Teil aus Zeitgründen nicht implementiert werden konnten, vorgestellt werden.

5.4.1 Lese-Zugriff

In der aktuell implementierten passiven Replikationsstrategie sendet der Primary bei einem Wechsel der Zugriffsart von Schreib-Lese-Zugriff zu Lese-Zugriff für alle eine gültige Kopie der Seite an alle Replikate. Diese Strategie ist vorteilhaft, falls damit zu rechnen ist, daß wesentlich mehr Lese-Zugriffe als Schreib-Zugriffe auf die Seite erfolgen, da dann nicht jedes Replikat einzeln eine gültige Kopie der Seite anfordern muß.

Ist dies jedoch nicht der Fall, so wäre es günstiger, wenn die wenigen Replikate, über die lesend auf die Seite zugegriffen wird, selbst eine Kopie anfordern würden, da dadurch die Last des Gesamtsystems reduziert würde. Da dies eine Entscheidung ist, die am besten vom Programmierer getroffen werden kann, sollte diese zweite Strategie hinzugefügt werden und dem Programmierer die Möglichkeit geboten werden, die passende zu wählen.

5.4.2 Konsistenzmodelle

Ein weiteres Problem ist die schlechte Performance des Systems, die im Wesentlichen auf die Forderung nach strenger Konsistenz zurückzuführen ist. Es sollte geprüft werden, ob ein anderes Konsistenzmodell und ein dazu passendes Programmiermodell besser geeignet wären. Eine Auflistung geläufiger Konsistenzmodelle und dazu passender Programmiermodelle ist in [Tan95] zu finden.

5.4.3 Garbage Collection

Natürlich sollte die passive Replikationsstrategie noch um einen Garbage Collection Algorithmus ergänzt werden. Hierzu kann ein ähnlicher Algorithmus wie für die aktive Replikation verwendet werden, jedoch muß berücksichtigt werden, daß der Primary ständig wechselt. Außerdem ist in diesem Zusammenhang die Frage zu klären, ob ein Replikat, das in einem Prozeß nicht mehr erreichbar ist, wirklich nicht mehr gebraucht wird. Es könnte sich beispielsweise um den Snooper handeln, welcher durchaus wichtige Aufgaben erfüllt.

Kapitel 6

Schlußbetrachtungen

In diesem letzten Kapitel folgen einige Schlußbetrachtungen. Zuerst werden anhand der von RepliXa verfolgten Ziele die wesentlichen Eigenschaften des Systems noch einmal herausgestellt und kritisch bewertet. Danach erfolgt eine Auflistung möglicher Erweiterungen des Systems.

6.1 Überblick und Bewertung

Die von RepliXa verfolgten Ziele werden in diesem Abschnitt noch einmal aufgegriffen. Dies geschieht aus zwei Gründen. Zum einen soll noch einmal ein kurzer Überblick über das System gegeben werden und zum anderen soll kritisch geprüft werden, inwieweit die Ziele erreicht wurden.

6.1.1 Einfachheit

Die Idee der Einfachheit war, daß dem Programmierer eine möglichst einheitliche und kleine Schnittstelle zur Nutzung der Dienste von RepliXa zur Verfügung steht.

Dieses Ziel kann als erreicht betrachtet werden, da eine bewußte Interaktion des Programmierers mit RepliXa nur an drei Stellen erforderlich ist und an diesen drei Stellen, wie im Folgenden noch erläutert wird, eine einheitliche und kleine Schnittstelle vorliegt. Eine Interaktion ist erforderlich zum

- Starten der Replikationsumgebung. Hier muß lediglich eine Instanz der Klasse REPLICATIONENVIRONMENT erzeugt werden.
- Ändern des Verhaltens von Objekten, d.h. die Umwandlung gewöhnlicher Objekte in Remote-Objekte bzw. replizierte Objekte. Hier wird ein für alle Verhaltensänderungen einheitlicher Mechanismus verwendet, der lediglich aus den Schritten Erzeugen eines Strategie-Objektes und Binden an das Basis-Objekt besteht.
- Registrieren und Anfordern von Objekten beim Nameservice. Hier wird ebenfalls ein für alle Objekte einheitlicher Mechanismus verwendet, der nur zwei Schritte umfaßt, nämlich das Einrichten des Zugriffs auf den Nameservice und das Registrieren bzw. Anfordern des Objektes.

Auszuführende Aktion	Sun JDK 1.1.7	RepliXa
Anfordern eines Stubs	340 ms	260 ms
Methodenaufruf	2.8 ms	70 ms

Tabelle 6.1: Vergleich der Performance des Java-RMIs und des RepliXa-RMIs. Die Messungen wurden auf einem Intel Pentium II Processor mit 350 MHz unter dem Betriebssystem Linux mit Hilfe des Shell-Kommandos `time` durchgeführt. Der das Remote-Objekt exportierende Prozeß und der anfordernde bzw. aufrufende Prozeß befanden sich auf dem gleichen Rechner. Als Remote-Objekt wurde das in Anhang A.1 beschriebene DATABASE Objekt verwendet.

6.1.2 Transparenz

Das Ziel der Transparenz konnte nicht vollständig erreicht werden, da diese im Moment des Bindens verloren geht. Dies ist der Fall, da, um ein Strategie-Objekt an ein Basis-Objekt zu binden, der Programmcode der Anwendung um Programmcode zur Interaktion mit RepliXa ergänzt werden muß. Dies wiederrum hat zur Folge, daß die Anwendung nicht vollständig ohne das Wissen über die Replikation erstellt werden kann.

Es handelt sich jedoch nur um eine leichte Verletzung der Transparenz, da das Meta-System nur für das Binden von Strategie-Objekten an Basis-Objekte sichtbar ist. Danach ist das Meta-System vollkommen transparent für das Basis-System.

Für das Starten der Replikationsumgebung und das Registrieren und Anfordern von Objekten beim Nameservice trifft das Gleiche zu.

6.1.3 Anpassbarkeit

Die geforderte leichte Anpassbarkeit an neue Ausführungsumgebungen wurde erreicht, da durch die einfache Schnittstelle von RepliXa nur wenig Änderungen erforderlich sind, um eine Anwendung einer neuen Umgebung anzupassen.

6.1.4 Offenheit

Bei RepliXa handelt es sich um ein offenes System, da problemlos neue Replikationsstrategien integriert werden können. Es müssen lediglich Unterklassen der abstrakten Klassen REPLICATIONSTRATEGY und REPLICAINSTALLER mit der entsprechenden neuen Funktionalität implementiert werden.

6.1.5 RMI-Mechanismus

Ein weiteres Ziel von RepliXa war es, einen RMI-Mechanismus zur Verfügung zu stellen. Die Analyse und Beurteilung dieses Mechanismus erfolgt im Rahmen eines Vergleichs mit dem ab Java 1.1 von Sun zur Verfügung gestellten.

Performance
Wie aus Tabelle 6.1 leicht zu ersehen ist, ist die Zeitdauer, die die beiden Mechanismen benötigen um einen Stub zur Verfügung zu stellen, in etwa gleich. Ein Methodenaufruf erfolgt jedoch

bei dem Java-RMI ca. um Faktor 25 schneller als bei dem RepliXa-RMI. Die Gründe hierfür sind im Wesentlichen:

- Das *Marshalling* der Parameter erfolgt auf unterschiedliche Weise. Bei dem RepliXa-RMI muß zur Laufzeit die Struktur der zu übertragenden Parameter analysiert werden. Bei dem Java-RMI erfolgt dies jedoch durch ein spezielles Tool zur Compilezeit, wodurch zur Laufzeit eine bessere Performance erzielt wird.

- Wie in Abschnitt 3.3.1 beschrieben, wird bei RepliXa jeder Entgegennahme eines RMIs ein neuer Thread erzeugt. Da dies einen großen Aufwand zur Laufzeit darstellt, sollte es in Zukunft wie in Abschnitt 3.3.1 vermieden werden.

- Das Auf- und Abbauen der TCP-Verbindung, wie es in Abschnitt 3.3.2 erläutert ist, stellt ebenfalls einen unnötigen Aufwand dar und sollte deshalb durch die in Abschnitt 3.3.2 beschriebenen Maßnahmen verhindert werden. Es wäre auch eine Realisierung mit Hilfe des UDP-Mechanismus denkbar.

- Der von RepliXa verwendete Java-Interpreter metaXa erzielt eine etwas schlechtere Performance als der Java-Interpreter von Sun.

Ein weiterer Grund für diese schlechte Performance des RepliXa-RMIs gegenüber dem Java-RMI dürfte an der Art der Messung liegen. Wenn der das Remote-Objekt exportierende Prozeß und der anfordernde bzw. aufrufende Prozeß auf verschiedenen Rechnern ablaufen würden, so würde durch die Kommunikation der beiden ein für beide Mechanismen gleicher Aufwand hinzukommen. Es ist anzunehmen, daß sich dadurch der Faktor von ca. 25 stark verringern würde.

Sperr-Mechanismus
Im Rahmen des RepliXa-RMIs ist es, im Gegensatz zum Java-RMI, möglich, Remote-Objekte zu sperren.

Objektübertragung
Bei dem Java-RMI gibt es, falls als Parameter oder Rückgabewert ein Objekt übertragen werden soll, nur die Möglichkeit, das Objekt entweder als Remote-Objekt oder komplett als Kopie zu übertragen. Bei dem RepliXa-RMI jedoch kann mit Hilfe von Strategie-Objekten für jedes Teilobjekt eine eigene Übertragungsstrategie festgelegt werden, wodurch eine wesentlich höhere Flexibilität im Bereich der Objektübertragung erreicht wird.

Abschließend kann festgestellt werden, daß der RepliXa-RMI durchaus Vorteile gegenüber dem Java-RMI besitzt. Die Performance jedoch sollte in Zukunft durch die oben beschriebenen Maßnahmen erheblich verbessert werden.

6.1.6 Replikationsstrategien

Das Bereitstellen von vielseitig verwendbaren Replikationsstrategien war ein weiteres Ziel von RepliXa. Es werden eine aktive und eine passive Replikationsstrategie zur Verfügung gestellt.

Aktive Replikation
Die aktive Replikationsstrategie eignet sich hervorragend zur Replikation von Daten, die teilweise graphisch dargestellt werden, da der replizierte Aktivitätsträger dazu verwendet werden kann, die graphische Komponente über Änderungen an den Daten zu informieren.

Dadurch, daß nur lesende Anfragen von den einzelnen Replikaten lokal bearbeitet werden, eignet sich diese Strategie auch besonders für Daten, die häufig gelesen und nur selten verändert werden. Da in diesem Fall der Aufwand zur Replikation gering bleibt und dadurch, daß eine Kopie der Daten in jedem Prozeß verfügbar ist, eine wesentlich bessere Performance als bei einem Remote-Objekt erzielt wird.

Passive Replikation
Die passive Replikationsstrategie eignet sich durch ihre Fehlertoleranz gegenüber dem Ausfall beliebiger Prozesse hervorragend zur Replikation kritischer Objekte. Außerdem muß eine passive Replikationsstrategie verwendet werden, falls die Methoden des Objektes nicht deterministisch sind.

Dem Anwendungsprogrammierer stehen also bereits eine Reihe von Strategien für unterschiedliche Anwendungen zur Verfügung. Es wär jedoch sinnvoll, diese noch um weitere zu ergänzen, wie beispielsweise eine passive Replikationsstrategie, die durch ein schwächeres Konsistenzmodell eine bessere Performance erzielt.

6.2 Erweiterungen

Da in den Abschnitten 3.3, 4.3 und 5.4 bereits viel Vorschläge zur Erweiterung des Systems gemacht wurden, werden in diesem Abschnitt nur noch ein paar wichtige Aspekte herausgegriffen.

6.2.1 Kombination der Mechanismen

Derzeit ist es in RepliXa nicht möglich, den RMI-Mechanismus mit einem replizierten Objekt zu kombinieren, ein Objekt ist also entweder ein Remote-Objekt oder ein repliziertes Objekt. Es könnte jedoch sinnvoll sein, ein repliziertes Objekt nicht vollständig zu replizieren. Nur ein paar Prozesse würden also ein Replikat erhalten und die anderen Prozesse würden auf diese Replikate über einen Stub, ähnlich dem Stub für Remote-Objekte, zugreifen.

Untersuchungen und Heuristiken, ob ein Objekt repliziert werden sollte und wo die Replikate zu plazieren sind, sind in [MC92] und [ML92] zu finden. Solche Untersuchungen sind nötig, da diese Entscheidungen bei größeren Systemen nicht mehr dem Programmierer überlassen werden sollten.

6.2.2 Migrierende Objekte

Da die Konzepte zur Objektmigration sehr eng mit den Konzepten für Remote-Objekte und replizierte Objekte verwandt sind, sollten diese in RepliXa integriert werden.

6.3 Schlußwort

Mit der Entwicklung von RepliXa wurde ein System geschaffen, das es ermöglicht, flexibel und transparent Objekte zu replizieren. Es ist jedoch noch nicht für den praktischen Einsatz geeignet, da aus Zeitgründen viele nötige Erweiterungen nur beschrieben und nicht implementiert werden konnten. Es kann jedoch dadurch als Grundlage für weitere Arbeiten in diesem Bereich dienen.

Anhang A

Beispielprogramm zum Grundsystem

Durch das hier vorgestellte Beispielprogramm soll die Verwendung der vom Grundsystem zur Verfügung gestellten Mechanismen demonstriert werden.

A.1 Die Klasse Database

```
package test.remote;
import java.util.*;

public class Database {
  private Hashtable entries;

  public Database() {
    entries = new Hashtable(20);
  }

  public synchronized void put(String key, String data) {
    System.out.println("Database(put): "+key+" "+data);

    entries.put(key, data);
  }

  public synchronized String retrieve(String key)
    throws UnknownKeyException {

    System.out.println("Database(retrieve): "+key);

    if (entries.containsKey(key))
      return (String) entries.get(key);

    throw new UnknownKeyException(key);
  }
}
```

Das Remote-Objekt, auf das durch die anderen Klassen zugegriffen wird, ist eine Instanz der Klasse DATABASE. Sie besitzt zwei Methoden. Eine Methode `put()`, mit der ein Datum unter einem bestimmten Schlüssel gespeichert werden kann, und eine Methode `retrieve()`, mit der das zu einem bestimmten Schlüssel gehörende Datum abgefragt werden kann.

A.2 Die Klasse UnknownKeyException

```
package test.remote;

public class UnknownKeyException extends Exception
{
  public UnknownKeyException() {
    super();
  }

  public UnknownKeyException(String s) {
    super(s);
  }
}
```

Falls durch die Methode `retrieve()` bei der Klasse DATABASE die Daten zu einem Schlüssel abgefragt werden, der nicht existiert, wird von dieser Methode eine Exception der Klasse UNKNOWNKEYEXCEPTION zurück gegeben.

A.3 Die Klasse Server

```
package test.remote;
import replica.*;

public class Server {
  public static void main(String argv[]) {

    // Bestimmen des Ports über den der Prozeß und
    // somit sein Nameservice später erreichbar sind
    int callInPort = 12345;
    if (argv.length >= 1) {
      try {
        callInPort = Integer.parseInt(argv[0]);
      } catch (NumberFormatException e) {
        callInPort = 12345;
      }
    }

    // Starten der Replikationsumgebung
    new ReplicationEnvironment(callInPort);
```

```
    // Erzeugen einer Datenbank als Remote-Objekt
    Database base = new Database();
    new AccessControlObject(base,new AccessLocalTransmitROR());

    // Registrieren beim Nameservice
    NameService.getLocalNS().register("Database", base);
  }
}
```

Beim Starten der Klasse SERVER wird eine Instanz der Klasse DATABASE als Remote-Objekt erzeugt und beim Nameservice registriert. Dadurch ist sie, über den Namesevice, für andere Prozesse erreichbar.

A.4 Die Klasse Update

```
package test.remote;
import replica.*;

public class Update {
  public static void main(String argv[]) {

    // Feststellen ob genügend Parameter angegeben wurden
    if (argv.length <2) {
      System.out.println("Usage: Update <key> <value> [port]");
      System.exit(1);
    }

    // Bestimmen des Ports über den der Nameservice
    // des Server-Prozesses erreichbar ist
    int contactPort = 12345;
    if (argv.length >2) {
      try {
        contactPort = Integer.parseInt(argv[2]);
      } catch (NumberFormatException e) {
        contactPort = 12345;
      }
    }

    // Starten der Replikationsumgebung
    new ReplicationEnvironment();

    try {
      // Zugriff auf Nameservice des Servers ermöglichen
      NameService remNS = NameService.getRemoteNS(contactPort);
```

```
      // Anfrage beim Nameservice des Servers
      Database base = (Database)remNS.get("Database");

      // Eintrag eines neuen Datums in der Datenbank
      base.put(argv[0], argv[1]);

    } catch(Exception ex) {

      // Abfangen und Ausgeben von Exceptions
      System.out.println("Exception(Update): "+
                         ex.getClass().getName()+
                         " "+ex.getMessage());
    }

    // Beenden des Programmes
    System.exit(0);
  }
}
```

Beim Starten der Klasse UPDATE wird zuerst eine Verbindung zur Datenbank hergestellt und dann ein neues Datum in der Datenbank gespeichert.

A.5 Die Klasse Query

```
package test.remote;
import replica.*;

public class Query {
  public static void main(String argv[]) {

    // Feststellen ob genügend Parameter angegeben wurden
    if (argv.length <1) {
      System.out.println("Usage: Query <key> [port]");
      System.exit(1);
    }

    // Bestimmen des Ports über den der Nameservice
    // des Server-Prozesses erreichbar ist
    int contactPort = 12345;
    if (argv.length >1) {
      try {
        contactPort = Integer.parseInt(argv[1]);
      } catch (NumberFormatException e) {
        contactPort = 12345;
      }
    }
```

```
    // Starten der Replikationsumgebung
    new ReplicationEnvironment();

    try {
      // Zugriff auf Nameservice des Servers ermöglichen
      NameService remNS = NameService.getRemoteNS(contactPort);

      // Anfrage beim Nameservice des Servers
      Database base = (Database)remNS.get("Database");

      // Abfragen eines Datums bei der Datenbank
      System.out.println("The data is: "+
                         base.retrieve(argv[0]));

    } catch(Exception ex) {

        // Abfangen und Ausgeben von Exceptions
        System.out.println("Exception(Query): "+
                           ex.getClass().getName()+
                           " "+ex.getMessage());
    }

    // Beenden des Programmes
    System.exit(0);
  }
}
```

Beim Starten der Klasse QUERY wird zuerst eine Verbindung zur Datenbank hergestellt und dann ein zu einem bestimmten Schlüssel gehörendes Datum bei der Datenbank abgefragt.

Anhang B

Beispielprogramm zur aktiven Replikation

Durch das hier vorgestellte Beispielprogramm soll die Verwendung der aktiven Replikationsstrategie demonstriert werden.

B.1 Die Klasse Diary

```
package test.active;

public class Diary {

  // Termineinträge
  String entries[][];

  // System-Objekt zur graphische Darstellung
  DiaryDisplay display;

  // System-Objekt zur Ausgabe von Debug-Meldungen
  DiarySystemOutput out;

  public Diary() {

    entries = new String[7][5];
    display = new DiaryDisplay(this);
    out = new DiarySystemOutput();
  }

  public String getEntryR(int day, int time) {

    try {
      out.println("Diary: getEntryR("+day+","+time+
                  ") called ("+entries[day][time]+")");
      return entries[day][time];
    } catch(IndexOutOfBoundsException ex) {
```

```
      return null;
    }
  }

  public void setEntry(int day, int time, String newEnt) {

    out.println("Diary: setEntry("+day+","+time+
                  ","+newEnt+") called");
    try {
      entries[day][time] = newEnt;
    } catch(IndexOutOfBoundsException ex) {}

    display.notifyDisplay(day, time, newEnt);
  }
}
```

Das Objekt, welches aktiv repliziert wird, ist eine Instanz der Klasse DIARY. Diese realisiert einen einfachen Terminkalender, der für jeden Tag der Woche fünf Termine verwalten kann. Sie besitzt eine Instanz der Klasse DIARYDISPLAY, welche für die graphische Darstellung zuständig ist, und ein Instanz der Klasse DIARYSYSTEMOUTPUT, welche für Debug-Ausgaben verwendet wird. Bei den Instanzen der Klassen DIARYDISPLAY und DIARYSYSTEMOUTPUT handelt es sich um System-Objekt, da replizierte Aufrufe an ihnen nicht über den Primary zu einem zusammengefasst werden sollen.

Des Weiteren besitzt die Klasse zwei Methoden. Die eine ist die Methode `getEntryR()`, über die ein Termineintrag abgefragt werden kann, und die andere ist die Methode `setEntry()`, über die ein Termineintrag gesetzt werden kann. Daß es sich bei der Methode `getEntryR()` um eine nur lesende Methode handelt, wird von der Replikationsstrategie dadurch erkannt, daß der Name der Methode mit `R` endet. Diese Methoden werden von der Klasse DIARYDISPLAY verwedet, die Methode `getEntryR()` um den internen Zustand des Displays zu aktualisieren, und die Methode `setEntry()`, um einen im Display neu eingegebenen Termin in den Terminkalender aufzunehmen.

B.2 Die Klasse DiarySystemOutput

```
package test.active;
import replica.*;

class DiarySystemOutput {

  DiarySystemOutput() {

    new AccessControlObject(this,
        new AccessTransmitSystemObject());
  }
```

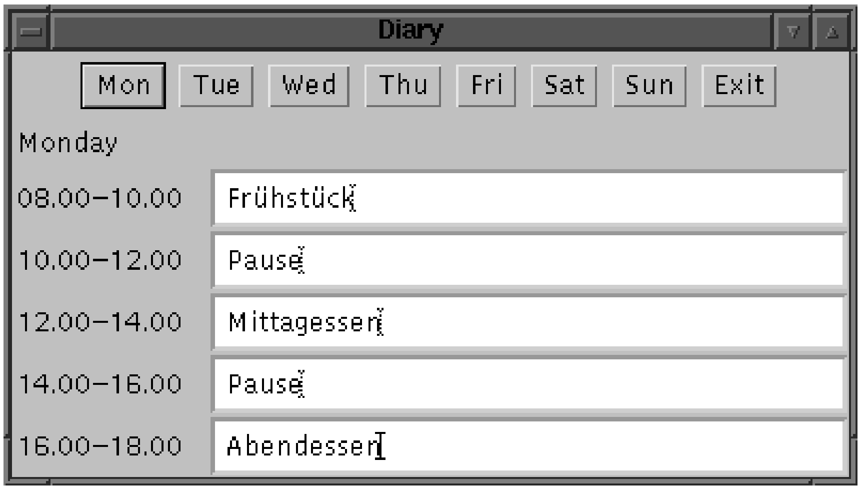


Abbildung B.1: Graphische Ausgabe der Klasse DiaryDisplay

```
  void println(String s) {

    System.out.println(s);
  }
}
```

Die Klasse DiarySystemOutput wird zur Ausgabe von Debug-Meldungen verwendet. Instanzen dieser Klasse sind automatisch System-Objekt.

B.3 Die Klasse DiaryDisplay

Bei der Klasse DiaryDisplay handelt es sich um eine graphische Komponente, welche zur Anzeige des Terminkalenders und zur Benutzerinteraktion verwendet wird. Sie ist in Abbildung B.1 dargestellt. Instanzen dieser Klasse sind automatisch System-Objekt.

B.4 Die Klasse Server

```
package test.active;
import replica.*;

public class Server {
  public static void main(String argv[]) {

    // Bestimmen des Ports über den der Prozeß und
    // somit sein Nameservice später erreichbar sind
    int callInPort = 12345;
    if (argv.length >= 1) {
      try {
        callInPort = Integer.parseInt(argv[0]);
```

```
      } catch (NumberFormatException e) {
        callInPort = 12345;
      }
    }

    // Starten der Replikationsumgebung
    new ReplicationEnvironment(callInPort);

    // Erzeugen eines Terminkalenders als repliziertes Objekt
    Diary diary = new Diary();
    new AccessControlObject(diary, new ActiveReplication());

    // Registrieren beim Nameservice
    NameService.getLocalNS().register("Diary", diary);
  }
}
```

Beim Starten der Klasse SERVER wird eine Instanz der Klasse DIARY als repliziertes Objekt erzeugt und beim Nameservice registriert.

B.5 Die Klasse Client

```
package test.active;
import replica.*;

public class Client {
  public static void main(String argv[]) {

    // Bestimmen des Ports über den der Nameservice
    // des Server-Prozesses erreichbar ist
    int contactPort = 12345;
    if (argv.length >=1) {
      try {
        contactPort = Integer.parseInt(argv[0]);
      } catch (NumberFormatException e) {
        contactPort = 12345;
      }
    }

    // Starten der Replikationsumgebung
    new ReplicationEnvironment();

    // Zugriff auf Nameservice des Servers ermöglichen
    NameService remNS = NameService.getRemoteNS(contactPort);

    // Anfordern eines Replikates beim Nameservice des Servers
```

```
    try {
      Diary diary = (Diary)remNS.get("Diary");
    } catch(Throwable th) {
      // Fehler ausgeben und beenden
      System.out.println("Exception(Client): "+th.getClass()
                         .getName()+" "+th.getMessage());
      System.exit(1);
    }
  }
}
```

Beim Starten der Klasse CLIENT wird ein Replikat des Terminkalenders angefordert.

Anhang C

Beispielprogramm zur passiven Replikation

Durch das hier vorgestellte Beispielprogramm soll die Verwendung der passiven Replikationsstrategie demonstriert werden.

C.1 Die Klasse OrderObject

```
package test.passive;

public class OrderObject {

  private int value;

  public OrderObject() {
    value = 0;
  }

  public int getNumber() {
    return value+1;
  }

  public synchronized int incNumber() {
    value=value+1;
    return value;
  }
}
```

Die Klasse ORDEROBJECT verfügt über zwei Methoden, mit deren Hilfe es möglich ist, eine monoton steigende Folge von Nummern zu erhalten. Die Methode `incNumber()` liefert die nächste Nummer und mit Hilfe der Methode `getNumber()` kann festgestellt werden, welche Nummer als nächstes geliefert würde.

Wenn eine Instanz dieser Klasse repliziert wird, so kann sie in einem verteilten System dazu genutzt werde, die zeitliche Abfolge von Ereignissen festzustellen. Wenn also für jedes

Ereignis eine Nummer angefordert wird, so geschahen Ereignisse mit eine kleineren Nummer vor Ereignissen mit einer größeren Nummer.

Diese Klasse dient jedoch in unserem Fall nur der Demonstration der passiven Replikation, da das zeitliche Ordnen von Ereignissen durch andere Vorgehensweisen mit einer wesentlich besseren Performance zu erreichen ist. Das Objekt welches im folgenden Beispiel passiv repliziert wird, ist also eine Instanz der Klasse ORDEROBJECT.

C.2 Die Klasse Writer

```
package test.passive;

import replica.*;
import java.util.*;

public class Writer {
  public static void main(String argv[]) {

    // Falls als Parameter eine Portnummer übergeben wurde,
    // so wird versucht, über den Nameservice der über diesen
    // Port erreichbar ist, ein Replikat eines bereits
    // existierenden replizierten OrderObjects zu erhalten.
    // Falls nicht, wird ein neues repliziertes OrderObject
    // angelegt.
    int conPort = 0;
    if (argv.length >=1) {
      try {
        conPort = Integer.parseInt(argv[0]);
      } catch (NumberFormatException e) {
        conPort = 0;
      }
    }

    // Starten der Replikationsumgebung
    new ReplicationEnvironment();

    // Verweis auf das Replikat des OrderObjects
    OrderObject order=null;

    // Keine Portnummer angegeben --> neues OrderObjekt anlegen
    if (conPort == 0) {

      // Neues repliziertes OrderObject erzeugen
      order = new OrderObject();
      new AccessControlObject(order, new PassiveReplication());
```

```
// Portnummer angegeben --> Replikat anfordern
} else {
  try {
    // Zugriff auf entfernten Nameservice ermöglichen
    NameService remNS = NameService.getRemoteNS(conPort);

    // Anfordern eines Replikates
    order = (OrderObject)remNS.get("OrderObject");

  } catch(Exception ex) {
    // Fehler ausgeben und beenden
    System.out.println("Exception(Writer): "+ex.getClass()
                       .getName()+" "+ex.getMessage());
    System.exit(1);
  }
}

// Registrieren beim eigenen Nameservice
NameService.getLocalNS().register("OrderObject", order);

// Bestimmen einer Verzögerungszeit zwischen einer und
// drei Sekunden
Random ran = new Random();
int r = ran.nextInt();
if (r<0) r=-r;
int t=r/2000;
r=r-(t*2000)+1000;

// In einer Endlosschleife wird zuerst eine neue Nummer
// angefordert und ausgegeben und danach eine bestimmte
// Verzögerungszeit gewartet.
while (true) {

  // Anfordern und Ausgeben einer neuen Nummer
  // (Zusätzlich wird die Portnummer des lokalen
  // Nameservices ausgegeben, so daß sie als Parameter
  // bei anderen Prozessen angegeben werden kann.)
  try {
    System.out.println("New Number "+order.incNumber()+
                       " ("+RemoteObjectControl.ownPort+
                       ")");
  } catch (Exception ex) {
    // Fehler ausgeben und beenden
    System.out.println("Exception(Writer): "+ex.getClass()
                       .getName()+" "+ex.getMessage());
    System.exit(1);
```

```
      }

      // Warten
      try {
        Thread.sleep(r);
      } catch (InterruptedException ex) {}
    }
  }
}
```

Beim Starten der Klasse WRITER wird zuerst festgestellt, ob eine Portnummer als Parameter übergeben wurde. Ist dies der Fall, so wird davon ausgegangen, daß bereits ein repliziertes ORDEROBJECT vorliegt und versucht, über den Nameservice, der über die Portnummer erreichbar ist, ein Replikat dieses Objektes zu erhalten. Ist dies nicht der Fall, wird eine neues repliziertes ORDEROBJECT erzeugt. Danach wird das Replikat bzw. das neue ORDEROBJECT beim lokalen Nameservice registriert, so daß es für andere Prozesse erreichbar ist. Nachdem dies geschehen ist, wird eine Endlosschleife ausgeführt, in der zuerst eine neue Nummer angefordert und ausgegeben wird und anschließend eine Zeit zwischen einer und drei Sekunden gewartet wird.

Die Portnummer des lokalen Nameservices wird in der Endlosschleife ebenfalls mit ausgegeben, so daß diese beim Start anderer Prozesse als Parameter mit angegeben werden kann. Den anderen Prozessen wird dadurch ermöglicht, ein Replikat des ORDEROBJECTs anzufordern.

C.3 Die Klasse Reader

```
package test.passive;

import replica.*;
import java.util.*;

public class Reader {
  public static void main(String argv[]) {

    // Falls als Parameter eine Portnummer übergeben wurde,
    // so wird versucht, über den Nameservice der über diesen
    // Port erreichbar ist, ein Replikat eines bereits
    // existierenden replizierten OrderObjects zu erhalten.
    // Falls nicht, wird ein neues repliziertes OrderObject
    // angelegt.
    int conPort = 0;
    if (argv.length >=1) {
      try {
        conPort = Integer.parseInt(argv[0]);
      } catch (NumberFormatException e) {
        conPort = 0;
```

```
    }
  }

  // Starten der Replikationsumgebung
  new ReplicationEnvironment();

  // Verweis auf das Replikat des OrderObjects
  OrderObject order=null;

  // Keine Portnummer angegeben --> neues OrderObjekt anlegen
  if (conPort == 0) {

    // Neues repliziertes OrderObject erzeugen
    order = new OrderObject();
    new AccessControlObject(order, new PassiveReplication());

  // Portnummer angegeben --> Replikat anfordern
  } else {
    try {
      // Zugriff auf entfernten Nameservice ermöglichen
      NameService remNS = NameService.getRemoteNS(conPort);

      // Anfordern eines Replikates
      order = (OrderObject)remNS.get("OrderObject");

    } catch(Exception ex) {
      // Fehler ausgeben und beenden
      System.out.println("Exception(Reader): "+ex.getClass()
                          .getName()+" "+ex.getMessage());
      System.exit(1);
    }
  }

  // Registrieren beim eigenen Nameservice
  NameService.getLocalNS().register("OrderObject", order);

  // Bestimmen einer Verzögerungszeit zwischen einer und
  // drei Sekunden
  Random ran = new Random();
  int r = ran.nextInt();
  if (r<0) r=-r;
  int t=r/2000;
  r=r-(t*2000)+1000;

  // In einer Endlosschleife wird zuerst die nächste Nummer
  // erfragt und ausgegeben und danach eine bestimmte
```

```
    // Verzögerungszeit gewartet.
    while (true) {

      // Erfragen und Ausgeben der nächsten Nummer
      // (Zusätzlich wird die Portnummer des lokalen
      // Nameservices ausgegeben, so daß sie als Parameter
      // bei anderen Prozessen angegeben werden kann.)
      try {
        System.out.println("The next number will be "+
                           order.getNumber()+" ("+
                           RemoteObjectControl.ownPort+")");
      } catch (ReplicaException ex) {
        // Fehler ausgeben und beenden
        System.out.println("Exception(Reader): "+ex.getClass()
                           .getName()+" "+ex.getMessage());
        System.exit(1);
      }

      // Warten
      try {
        Thread.sleep(r);
      } catch (InterruptedException ex) {}
    }
  }
}
```

Die Klasse READER unterscheidet sich von der Klasse WRITER lediglich darin, daß in der Endlosschleife nicht eine neuen Nummer angefordert wird, sonder die nächste Nummer erfragt wird.

Literaturverzeichnis

[BEN93] Andrew Birrell, David Evers, Greg Nelson, Susan Owicki, Edward Wobber, Distributed Garbage Collection for Network Objects, Digital, System Research Center, Research report 116, December 1993

[Bir96] Kenneth P. Birman, Building Secure and Reliable Network Applications, Manning Publications Co., Greenwich, 1996

[Boo93] Grady Booch, Object-Oriented Analysis and Design with Applications, 2nd ed., Redwood City, California, Addison-Wesley Publishing Company, 1993

[BRJ99] Grady Booch, James Rumbaugh, Ivar Jacobson, The Unified Modelling Language User Guide, Addison Wesley Longman, Inc., Massachusetts, 1999

[BW94] Larry Brown, Jie Wu, Dynamic Snooping in a Fault-Tolerant Distributed Shared Memory, 14th ICDCS, 1994, Seite 218-226

[Dud93] Hermann Engesser, Herausgeber, Duden Informatik, BI-Wissenschaftsverlag, Mannheim, 1993

[Fla96] David Flanagan, Java in a Nutshell, O'Reiley & Associates, Inc., Cambridge, 1996

[Gar98] Benoît Garbinato, Protocol Objects and Patterns for Structuring Reliable Distributed Systems, Département d´Informatique, École Polytechnique Fédérale de Lausanne, Lausanne, 1998

[GGM95] Rachid Guerraoui, Benoît Garbinato, Karim Mazouni, Implementation of the GARF Replicated Objects Platform, Distributed Systems Engineering Journal, March 1995

[GGM96] Rachid Guerraoui, Benoît Garbinato, Karim Mazouni, Lessons from Designing and Implementing GARF, erschienen in: Object Oriented Parallel and Distributed Computing, Springer Verlag, LNCS, 1996

[GHJ96] Erich Gamma, Richard Helm, Ralph Johnson, Jon Vlissides, Entwurfsmuster, Addison-Wesley, Bonn 1996

[GK97a] Michael Golm, Jürgen Kleinöder, MetaJava-A Platform for Adaptable Operating-System Mechanisms, Presented at the ECOOP 97 Workshop on Object-Orientation and Operating Systems, Jyväskylä, Finland, LNCS 1357, June 10, 1997

[GK97b] Michael Golm, Jürgen Kleinöder, MetaJava, Presented at STJA '97, Erfurt, September 10, 1997

[GK98] Michael Golm, Jürgen Kleinöder, metaXa and the Future of Reflection, Presented at the OOPSLA Workshop on Reflective Programming in C++ and Java, Vancouver, British Columbia, October 18, 1998

[Gol97] Michael Golm, Design and Implementation of a Meta Architecture for Java, Diplomarbeit DA-14-02-97, Universität Erlangen-Nürnberg, IMMD IV, Erlangen, 1997

[HST96] Franz J. Hauck, Maarten van Steen, Andrew S. Tanenbaum, A Location Service for Worldwide Distributed Objects, Position Paper for ECOOP'96 Workshop on Mobility and Replication, April 10, 1996

[JL96] Richard Jones, Rafael Lins, Garbage Collection, John Wiley & Sons Ltd., Chichester, 1996

[KG96] Jürgen Kleinöder, Michael Golm, Transparent and Adaptable Object Relication Using a Reflective Java, Technical Report TR-14-96-07, Universität Erlangen-Nürnberg, IMMD IV, Erlangen, 1996

[Kle98] Jürgen Kleinöder, Skript zur Vorlesung Object-Oriented Concepts in Distributed Systems, Universität Erlangen-Nürnberg, IMMD IV, Erlangen, 1998

[KS92] James J. Kirstler, M. Satyanarayanan, Disconnected Operation in the Coda File System, ACM Transactions on Computer Systems, Vol. 10, No. 1, Februar 1992, Seite 3-25

[Lar98] Craig Larman, Applying UML and Patterns, Prentice Hall, Inc., New Jersey, 1998

[LGJ91] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, Michael Williams, Replication in the Harp file system, Proceedings of 13th ACM Symposium on Operating Systems Principles, Association for Computing Machinery SIGOPS, Oktober 1991, Seite 226-38

[MC92] Silvano Maffeis, Clemens H. Cap, Replication Heuristics and Polling Algorithm for Object Replication and a Replicating File Transfer Protocol, IFI TR 92.06, Zürich, 1992

[ML92] Daniel McCue, Mark Little, Computing Replica Placement in Distributed Systems, Appeared in the Proceedings of the IEEE Second Workshop on Replicated Data, Monterey, November 1992, pp 58-61

[Mos93] D. Mosberger, Memory Concistency Models, Technical Report (TR 93/11), University of Arizona, 1993

[Per93] Radia Perlman, Interconnections, Addison-Wesley, Reading, Mass., 1993

[Sch90] Fred B. Schneider, Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial, ACM Computing Surveys, Vol. 22, N0.4, December 1990

[SG98] Abraham Silberschatz, Peter Bear Galvin, Operating System Concepts, Addison Wesley Longman, Inc., 1998

[SHH98] Marteen van Steen, Franz J. Hauck, Philip Homburg, Andrew S. Tanenbaum, Locating Objects in Wide-Area Systems, IEEE Communications Magazine, January 1998, pp. 104-109

[SS94] Mukesh Singhal, Niranjan G. Shivatatri, Advanced Concepts in Operating Systems, McGraw-Hill, Inc., 1994

[Tan92] Andrew S. Tanenbaum, Modern Operating Systems, Prentice Hall, Inc., New Jersey, 1992

[Tan95] Andrew S. Tanenbaum, Verteilte Betriebssysteme, Prentice Hall Verlag, München, 1995

[Tan96] Andrew S. Tanenbaum, Computer Networks, Prentice Hall, Inc., New Jersey, 1996