Technische Universität München
Fakultät für Informatik

Diplomarbeit in Informatik

Erweiterung des „Scalable Source Routing" Protokolls zur Adressierung und Migration von „Global Accessible Objects"

Extension of the „Scalable Source Routing" protocol for addressing and migrating „Global Accessible Objects"

Bearbeiter:	Mathias Kellerer
Aufgabensteller:	Prof. Dr. Thomas Fuhrmann
Betreuer:	Dipl.-Ing. Björn Saballus
Abgabedatum:	11.12.2008

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

11.12.2008

Datum Mathias Kellerer

Zusammenfassung

Die Verbreitung von eingebetteten Systemen und deren zunehmende Vernetzung erhöht die Komplexität und den Aufwand der Programmierung dieser Systeme. Eine Lösung der Problematik ergibt sich durch die Abstraktion des verteilten Charakters des Systems mit Hilfe einer virtuellen Maschine. Das AmbiComp Projekt entwickelt eine derartige virtuelle Maschine für Java, die direkt auf den eingebetteten Systemen aufsetzt.
Objekte, die für die Ausführung einer Java Anwendung notwendig sind, sollen zwischen den einzelnen Knoten des Systems ausgetauscht werden können. Dies erfordert eine Möglichkeit, lokale und entfernte Objekte zu adressieren, sowie zu migrieren. Im Verlauf dieser Arbeit wird die Adressierung und Migration von Objekten und die sich daraus ergebenden Probleme diskutiert. Zur Lösung der Probleme werden mehrere Verfahren, die auf der Verwendung des Scalable Source Routing Protokolls basieren, betrachtet und miteinander verglichen. Darüber hinaus wird ein ausgewähltes Verfahren näher analysiert.

Inhaltsverzeichnis

1 Einleitung

Computer sind heutzutage fast nicht mehr aus dem alltäglichen Leben wegzudenken, auch wenn ihre Präsenz nicht immer offensichtlich ist. Dies fängt beim „Standard PC" an und hört bei eingebetteten Systemen, die für die vielfältigsten Aufgaben entwickelt wurden, auf. Zum Beispiel existieren in einem Haushalt die verschiedensten eingebetteten Systeme. Dazu zählen eine Steuerung der Rollläden in Abhängigkeit der Wind- und Sonnenverhältnisse oder die Erfassung der Innen- und Aussentemperaturen eines Hauses. Bisher agierten diese Systeme meist autonom. Es liegt jedoch nahe, die einzelnen Systeme miteinander zu verbinden und somit die vorhandenen Daten und Ressourcen gemeinsam nutzen zu können. Letztlich wird dadurch ein verteiltes System geschaffen.

Der Begriff *Ambient Intelligence* (siehe [1] und [29]) steht für die Symbiose einer Vielzahl einzelner, eingebetteter Systeme. Ausgestattet mit Rechen- und Kommunikationsleistung, Sensoren und Aktoren ist es diesen Systemen möglich, durch Kooperation bestimmte Funktionen zu realisieren und Aufgaben auszuführen. Umgebungen, die mit einem „ambienten System" ausgestattet sind, können durch das Zusammenwirken als scheinbar intelligent erscheinen. Anwendungen, die auf einem ambienten System aufsetzen, können in vielen Bereichen eingesetzt werden. Dabei ist das Ziel den Alltag des Menschen zu erleichtern. Das „Intelligente Haus" (siehe [7]) ist nur eines von vielen denkbaren Anwendungsbeispielen (aus [32]).

Die Entwicklung ambienter Systeme erfordert viele Techniken, die kombiniert werden müssen. Dies resultiert in einer hohen Komplexität und einem großen Aufwand für die Entwicklung. Das *AmbiComp* Projekt (siehe [8]) entwickelt eine Plattform, die zur „einfacheren" Erstellung und Ausführung von Java Anwendungen auf eingebetteten Systemen verwendet werden kann. Ein Fokus des Projekts liegt auf der *AmbiComp Virtual Machine* (ACVM). Die ACVM ist eine speziell für eingebettete Systeme entwickelte *Java Virtual Machine* (JVM), die die Ausführung einer Anwendung nicht lokal auf einen einzelnen Knoten beschränkt. Das bedeutet, dass eine Anwendung verteilt auf einem Teil oder dem gesamten ambienten System ausgeführt werden kann. Für die verteilte Ausführung muss die Anwendung jedoch nicht angepasst werden. Die ACVM abstrahiert das verteilte System, auf dem sie aufsetzt, und bildet somit ein sogenanntes *Single System Image* (siehe [5]). Dies erfordert eine automatische Verteilung der Anwendung und der zugehörigen Daten im System, sowie die verteilte Ausführung der Anwendung. Die Abstraktion bzw. das Single System Image sorgt dafür, dass ein Programmierer zur Erstellung einer verteilten Anwendung sich nicht mit den Details eines verteilten Systems befassen muss. Somit kann sich der Programmierer auf die eigentlichen Aufgaben konzentrieren.

Das ambiente System besteht aus vielen Einzelkomponenten, die im weiteren Verlauf als Knoten bezeichnet werden. Die einzelnen Knoten werden mittels verschiedener Netzwerktechnologien, wie z.B. Bluetooth, Wireless LAN, Ethernet, usw. miteinander verbunden. Abhängig vom Anwendungsszenario ergeben sich meistens Netzwerktopologien, die nicht strukturiert sondern zufällig sind. Zur Adressierung der einzelnen Knoten und zum Routen von Nachrichten wird das *Scalable Source Routing* (SSR) Protokoll (siehe [9] und Kapitel 4) verwendet. SSR ist ein neuer Ansatz für das Routen in großen und unstrukturierten Netzwerken und kann für eingebettete Systeme verwendet werden, da es einen limitierten Ressourcenbedarf aufweist.

Die Verteilung von Anwendungen im System geschieht mittels dem Konzept der *Global Accessible Objects* (GAOs) (siehe [28]). Diese Objekte repräsentieren die einzelnen Bestandteile einer Anwendung. Dazu zählen Threads, ausführbarer Code, Java Objekte und Java Arrays. Die GAOs werden auf Knoten des Netzwerks verteilt und dort vorgehalten. Während der Ausführung der ACVM wird mittels Nachrichten auf die im Netzwerk verteilten GAOs zugegriffen. Des weiteren besteht die Anforderung, dass GAOs zwischen den Knoten migrieren können. Dies ermöglicht zur Laufzeit einer Anwendung, deren Threads und Daten im System zu verteilen. Daraus resultiert die parallele Ausführung von mehreren Threads und eine bessere Lastverteilung. Zusätzlich können Beziehungen zwischen den GAOs zur Steigerung der Lokalität ausgenutzt werden.

Im Rahmen dieser Diplomarbeit wird das bereits bestehende SSR Protokoll erweitert. Das Ziel der Erweiterung ist die Adressierbarkeit von GAOs. Das bedeutet, dass ein GAO immer für diejenigen Objekte erreichbar ist, die eine gültige Referenz auf das betreffende GAO besitzen. Dabei ist die Adressierbarkeit der GAOs unabhängig von den Knoten, die die GAOs speichern. Darüber hinaus wird ein Mechanismus geschaffen, der es erlaubt, GAOs beliebig zwischen den Knoten migrieren zu können. Es muss jedoch garantiert sein, dass alle referenzierten GAOs zu jedem Zeitpunkt erreichbar sind.

Im Folgenden wird zunächst auf den aktuellen Forschungsstand bezüglich verteilten Java Virtual Machines und der Adressierbarkeit von verteilten Objekten eingegangen. Anschließend gibt Kapitel 3 einen Überblick zur ACVM und dem Konzept der GAOs. Eine detailliertere Darstellung des SSR Protokolls erfolgt in Kapitel 4. Die daran anschließenden Kapitel 5 und 6 erläutern die Realisierung der Adressierung und Migration von GAOs, die im Verlauf dieser Diplomarbeit implementiert wurde. Kapitel 7 gibt einen Überblick zur Implementierung. Die Evaluation des implementierten Adressierungs- und Migrationsverfahrens findet im Kapitel 8 statt. Abschließend gibt Kapitel 9 eine Zusammenfassung und einen Ausblick über noch ausstehende und zukünftige Erweiterungen.

2 Aktueller Forschungsstand

Verteilte *Java Virtual Machines* (JVMs) wurden bereits mehrfach in der Forschung diskutiert. An dieser Stelle soll ein Überblick über die existierenden verteilten JVMs gegeben werden. Diese lassen sich in zwei Kategorien einteilen. In die erste Kategorie fallen diejenigen JVMs, die die Java Sprache erweitern oder eine Programmierschnittstelle für den Zugriff auf entfernt vorgehaltene Objekte zur Verfügung stellen. Dies erfordert jedoch eine Anpassung der Java Anwendungen. Die zweite Kategorie wird von den JVMs gebildet, die eine Java Anwendung unverändert auf dem verteilten System ausführen können. JVMs der zweiten Kategorie bieten dem Programmierer die Sichtweise eines *Single System Images* (SSI).

Im Folgenden wird nun auf Vertreter der beiden Kategorien eingegangen. Der Fokus der Betrachtung liegt dabei auf dem jeweils für die Adressierung und Migration von verteilten Objekten gewählten Ansatz.

cJVM

Die *Cluster Java Virtual Machine* (cJVM) (siehe [2]) ist eine verteilte JVM, die auf einem Cluster ausgeführt wird. Der Cluster besteht aus einer homogenen Menge von unabhängigen Maschinen, die mit einem schnellen Kommunikationsmedium verbunden sind. Die Homogenität der Maschinen bezieht sich dabei auf die Hardwarearchitektur und das Betriebssystem. Das Ziel von cJVM ist die Skalierbarkeit einer bestimmten Klasse von Java Anwendungen, die als *Java Server Applications* (siehe [2]) bezeichnet werden. Dazu werden die Threads der Anwendung auf mehrere Clusterknoten verteilt und ausgeführt. Die cJVM bietet jeder Anwendung die Sichtweise eines SSI.

Im Gegensatz zu Threads, die zwischen den Clusterknoten migrieren können, werden Java Objekte als stationär betrachtet. Das bedeutet, dass jedes Objekt auf genau einem Knoten des Clusters existiert. Dies sind die *Master Objekte*. Zugriffe von Threads auf Master Objekte, die nicht lokal gespeichert sind, werden mit Hilfe sogenannter *Proxies* realisiert. Für ein Master Objekt existieren Proxies auf den Knoten, die Threads ausführen, die auf dieses Master Objekt zugreifen. Dazu leiten die Proxies die Zugriffe an denjenigen Knoten weiter, der das Master Objekt speichert. Zur Leistungssteigerung werden mit der cJVM sogenannte *Smart Proxies* eingeführt. Die Smart Proxies basieren auf der Funktionsweise der Proxies. Jedoch können zusätzlich verschiedene Zugriffsstrategien realisiert werden. Zum Beispiel kann ein Objekt

innerhalb eines Smart Proxies gepuffert werden. Durch die Pufferung muss die Weiterleitung des Zugriffs nicht mehr durchgeführt werden. Jedoch erfordert dies, dass Zugriffe auf das durch den Smart Proxy repräsentierte Master Objekt nur lesend erfolgen (aus [2], Kapitel 4.1).

Die cJVM verwendet einen verteilten Heap. Dieser setzt sich aus den lokalen Heaps (siehe [18]) der Clusterknoten zusammen. Somit ist es möglich, dass alle Proxies die zugehörigen Master Objekte lokalisieren und darauf verweisen können. Dazu wird bei der Übergabe von Referenzen zwischen Objekten folgendermaßen vorgegangen: Falls eine Referenz, die auf ein lokales Objekt A verweist, zum ersten Mal an ein Objekt B übergeben wird, so wird ein Proxy für A erzeugt. Der Proxy wird im lokalen Heap des Knotens, der B vorhält, gespeichert und verweist auf den Knoten, der Objekt A speichert. Das Objekte A wird zu einem Master Objekt und erhält eine global gültige Adresse (aus [2], Kapitel 4.3).

JESSICA2

JESSICA2 steht für *Java Enabled Single System Image Computing Architecture version 2* (siehe [36]) und stellt eine verteilte JVM dar. Eine Besonderheit der JVM ist die Verwendung eines *Just-In-Time* Compilers (siehe [35] und [36]) zur Erhöhung der Performanz im Vergleich zur reinen Interpretation des Bytecodes. Genauso wie die cJVM setzt JESSICA2 auf einem Cluster auf und unterstützt die Migration von Threads zwischen den einzelnen Knoten. Nachdem eine Anwendung auf einem Knoten, dem *Master* gestartet wurde, werden die Threads der Anwendung auf *Worker* Knoten verteilt. Die Verteilung der Threads wird durch den sogenannten *Load Monitor* koordiniert.

Nachdem Threads auf diverse Clusterknoten verteilt wurden und deren Ausführung begonnen hat, wird durch diese lesend und schreibend auf Objekte zugegriffen. Die Zugriffe auf Objekte und deren Synchronisierungen wird durch die *Global Object Space* (GOS) Schicht realisiert. Diese Schicht ist ein fester Bestandteil von JESSICA2 und erzeugt für Anwendungen die Sichtweise eines SSIs. Bezüglich des GOS wird jedes Objekt des verteilten Systems als *Master Kopie* bezeichnet. Um Zugriffe eines Threads auf entfernt gespeicherte Objekte zu beschleunigen, werden die Objekte lokal auf dem Knoten, der den Thread ausführt, zwischengespeichert. Diese werden als *Cached Kopien* bezeichnet. Solange keine Synchronisation der Objekte erforderlich ist, können die Zugriffe lokal erfolgen (aus [36], Kapitel 4.1).

Zur Synchronisation von Objektzugriffen werden Monitore (siehe [18]) verwendet. Immer wenn ein Monitor bezüglich einer synchronisierten Cached Kopie betreten wird, muss die Cached Kopie mit der Master Kopie überschrieben werden. Im Gegenzug muss beim Verlassen des Monitors die Master Kopie durch die geänderte Cached Kopie aktualisiert werden. Häufige Zugriffe auf synchronisierte Objekte, beeinflussen die Systemperformanz in negativer Weise. Um dies zu vermeiden, wird das

Konzept der *Adaptive Object Home Migration* verwendet. Dabei ist das Ziel die Master Kopien der Objekte zwischen den Knoten des Netzwerks auszutauschen, so dass die Anzahl lokaler Zugriffe erhöht wird. Dazu wird die Bindung der Master Kopie an einen bestimmten Knoten aufgehoben. Somit ist es möglich, dass die Master Kopie von einem Knoten zu einem anderen Knoten übertragen werden kann. Nachdem die Master Kopie zum Zielknoten transferiert wurde, wird auf dem ursprünglichen Knoten die Master Kopie zur Cached Kopie. Die Auswahl des Zielknotens ist davon abhängig, wie häufig Threads, die auf dem Zielknoten ausgeführt werden, auf das betreffende Objekt zugreifen (aus [36], Kapitel 4.2).

MagnetOS

Die Realisation eines Single System Images ist nicht auf Cluster beschränkt. Bezüglich eines SSIs für ad hoc Netzwerke wurde *MagnetOS* (siehe [19]) konzipiert und implementiert. MagnetOS stellt ein verteiltes Betriebssystem dar. Das gesamte Netzwerk wird durch die MagnetOS-Betriebssystemschicht abstrahiert, so dass das Netzwerk gegenüber Anwendungen als eine einzige, virtuelle JVM erscheint. Anwendungen werden automatisch und transparent in Komponenten aufgeteilt, die dynamisch auf Knoten des Netzwerks verteilt werden. Ziel der Verteilung ist die Reduktion des Energieverbrauchs und resultiert in einer Erhöhung der Lebensdauer des Systems.

Das durch MagnetOS erzeugte SSI basiert auf der Verwendung von *Events*. Darüber hinaus bestehen Anwendungen aus einer Menge von *Event Handlern*, *Dispatch Handles* und *Event Descriptions*. Ein Event Handler entspricht den Java Objekten der Anwendung und kann beliebig zwischen den Netzwerkknoten migriert werden. Referenzen auf entfernte Event Handler werden durch Dispatch Handles repräsentiert, die die aktuellen Positionen der Event Handler im Netzwerk speichern. Ein Event, das an einen Event Handler gerichtet ist, wird in Verbindung mit dem zugehörigen Dispatch Handle lokal ausgelöst. Die Laufzeitumgebung von MagnetOS bildet diesen Event auf einen *Remote Procedure Call* (siehe [21]) ab und leitet diesen an denjenigen Knoten weiter, der den zugehörigen Event Handler vorhält. Des weiteren werden die Signaturen der Events durch die Event Descriptions beschrieben.

Die Migration eines Objekts umfasst die Migration des repräsentierenden Event Handlers. Dabei werden die referenzierenden Dispatch Handles aus Gründen der Energieeinsparung nicht aktualisiert. Jedoch wird eine Referenz, die als Weiterleitung dient, auf dem ursprünglichen Knoten hinterlassen. Durch wiederholte Migrationen entstehen Referenzketten, die bei der Traversierung von MagnetOS zusammengefasst werden. Eine Traversierung geschieht einerseits durch Events und andererseits durch die Garbage Collection.

Das von MagnetOS verwendete Verfahren zur Migration ist ähnlich zu dem vorgeschlagenen Verfahren aus Kapitel 6.2.4. Dabei ist ein Nachteil von MagnetOS die

Tatsache, dass im Falle einer unterbrochenen Referenzkette der zugehörige Event Handler durch einen Broadcast ermittelt werden muss. Zu unterbrochenen Referenzketten kann es durch eine fehlende Verbindung, Überlastung und Interferenzen (bei drahtlosen Verbindungen) kommen. Die Broadcasts sind durch das von MagnetOS verwendete AODV Routing Protokoll (siehe [23]) bedingt. Dies ist bei dem in dieser Arbeit vorgestellten Mechanismus nicht der Fall, da bei unterbrochenen Referenzketten der SSR Routing Algorithmus (siehe Kapitel 4.2) verwendet wird. SSR benötigt diesbezüglich keine Broadcasts.

JavaParty

Im Gegensatz zu den oben angeführten JVMs erweitert *JavaParty* (siehe [24]) die Programmiersprache Java um das Schlüsselwort *remote*. Mit Hilfe dieses Schlüsselwortes werden Java Klassen modifiziert, so dass Instanzen dieser Klassen auf entfernten Knoten vorgehalten und zwischen den Knoten migriert werden können. Die Modifikation ist nicht nur auf Objekte beschränkt, sondern auch für Threads möglich. Bezüglich Threads bedarf es jedoch eines erhöhten Aufwands, der in [15] dargestellt wird. JavaParty realisiert für alle Instanzen von lokalen und remote Objekten einen gemeinsamen Adressraum und ermöglicht somit den transparenten Zugriff auf verteilte Objekte.

Zugriffe auf remote Objekte, genauer gesagt die Aufrufe von Methoden, werden auf Java's *Remote Method Invocation* (RMI) (siehe [20]) abgebildet. Dazu verfolgt JavaParty den Ansatz, dass der erweiterte Quellcode in einem Vorverarbeitungsschritt in puren Java Quellcode übersetzt wird. Während des Vorverarbeitungsschritts wird das Erzeugen von remote Objekten und die Zugriffe auf remote Objekte durch entsprechenden RMI Quellcode ersetzt. Der in diesem Schritt entstandene Java Quellcode wird mit einem Standard Java Compiler übersetzt und kann somit auf jeder Plattform, die über eine JVM verfügt, ausgeführt werden. RMI verwendet für den Zugriff auf ein Objekt die Adresse des Knotens, der das remote Objekt speichert. Im Falle einer Migration wird auf dem ursprünglichen Knoten ein sogenannter *Proxy* erzeugt, der den Zielknoten der Migration speichert. Erreicht ein Methodenaufruf einen Proxy, so wird an das aufrufende Objekte eine *MovedException* zurückgeschickt. Die MovedException enthält den Zielknoten der Migration und löst somit den erneuten Zugriff auf das remote Objekt aus.

Die vorhergehend beschriebenen Projekte verfolgen primär nicht das Ziel einer effektiven Realisierung der Adressierung und Migration von Objekten. Entweder wird die Migration, wie bei cJVM, komplett umgangen oder mit Hilfe von gespeicherten *Proxies*, die die Zugriffe auf migrierte Objekte an die jeweiligen Zielknoten weiterleiten, gelöst. Die Tatsache, dass dadurch zusätzlicher Speicher benötigt wird und ein entsprechender Mehraufwand bezüglich der Kommunikation entsteht, wird jedoch vernachlässigt. Diese Vernachlässigung ist auf Grund der Leistungsfähigkeit

der verwendeten Hardware- und Kommunikationsinfrastrukturen zulässig, solange der Overhead beschränkt ist. Hingegen relativiert die Zielplattform des AmbiComp Projekts diesen Sachverhalt. Die mit der Verwendung von eingebetteten System einhergehenden Ressourcenbeschränkungen erfordern ein an die Anforderungen angepasstes Verfahren.

3 ACVM und GAOs

Mit der Idee der Vernetzung von kleinen, eingebetteten Systemen, deren Präsenz im Alltag immer mehr zunimmt, entsteht der Bedarf für eine verteilte Laufzeitumgebung. Diese ermöglicht es, das durch die Vernetzung entstandene verteilte System zu programmieren und Anwendungen darauf auszuführen. Der Fokus liegt hierbei auf der kollektiven Nutzung von Daten und Ressourcen, die jede einzelne Komponente des Systems zur Verfügung stellen kann. Somit ist es für alle Knoten möglich, kooperativ Problemstellungen zu bearbeiten und Aufgaben auszuführen.

Ein großer Nachteil von eingebetteten Systemen ist jedoch deren Programmierung in einer hardwarenahen Programmiersprache, wie zum Beispiel C oder Assembler. Zur Programmierung der vorliegenden Hardware ist ein fundiertes Wissen erforderlich. Der dabei entstehende Quellcode ist sehr systemspezifisch und nur schwer oder überhaupt nicht portierbar. Wird hingegen eine Hochsprache, wie zum Beispiel die Programmiersprache Java, zur Abstraktion der zugrundeliegenden Hardware verwendet, so bedarf dies eines zusätzlichen Aufwands. Die eingeführte Abstraktionsebene muss durch eine virtuelle Maschine realisiert werden, die den vom Compiler erzeugten Bytecode ausführen kann (aus [27], Abstract).

Um für Sensorknoten und andere eingebettete Systeme den obig genannten Nachteil zu umgehen, wird im Zuge des Forschungsprojekts *AmbiComp* die *AmbiComp Virtual Machine* (ACVM) entwickelt. Die ACVM ist eine *Java Virtual Machine* (JVM) (siehe [18]), die für die Ausführung auf Mikrocontrollern optimiert ist. Zur Gewährleistung kleiner Binärdateien, werden die durch den *javac* Compiler erzeugten *.class*-Files mittels eines vorgelagerten Pre-Processing Schritts (Transkodierung) in ausführbaren Code übersetzt. Durch den Übersetzungsvorgang entsteht ein sogenanntes *Binary Large Object* (BLOB), das anschließend auf die Zielknoten transferiert wird. Nach Abschluss des Vorgangs startet die ACVM mit der Ausführung des BLOBs. Jeder Knoten des vernetzten Systems muss dazu eine Instanz der ACVM bereits im Vorfeld gestartet haben. Die Ausführung kann entweder nur lokal oder auch verteilt im System geschehen.

Die ACVM selbst realisiert einen Großteil der Funktionen, die normalerweise von einem Betriebssystem zur Verfügung gestellt werden. Dazu zählen das Speichermanagement und eine Realisierung des Speicherschutzes durch die *Memory Management Unit* (MMU), sowie alle Arten von IO-Funktionalität. Für die Unterstützung von Multithreading ist ein Scheduler für die Threads, die innerhalb der VM existieren, notwendig. Mit Bezug auf eine verteilte Ausführung von Anwendungen muss das Speichermanagement einen Mechanismus zur verteilten *(distributed) Garbage Collection* (DGC) (siehe [25]) realisieren. Eine verteilte Garbage Collection hat die

Aufgabe, nicht mehr referenzierte Objekte, die global auf die Knoten des Systems verteilt sind, zu entfernen. Die ACVM ist somit ohne Betriebssystem direkt auf den Mikrocontrollern oder anderen Zielplattformen lauffähig (aus [27], Kapitel 3).

Eine verteilte Ausführung von Anwendungen durch eine Vielzahl miteinander verbundener ACVMs erfordert nicht nur den reinen Austausch von Daten zwischen Threads, die auf unterschiedlichen Knoten ausgeführt werden. Es wird ein Mechanismus benötigt, der es zusätzlich ermöglicht, ausführbaren Programmcode, Threads und deren zugehörige Zustandsinformationen (Programmzähler, Register, Stack) zwischen den einzelnen ACVM Instanzen auszutauschen bzw. von einer ACVM Instanz zu einer anderen Instanz zu transferieren. Die Daten sind im Kontext von Java Programmen durch Java Objekte und Java Arrays repräsentiert. Bei Betrachtung von Programmcode, Threads, Java Objekten und Java Arrays auf einer niedrigeren Ebene zeigt sich, dass dies Speicherbereiche sind, die die zugehörigen Befehle, Funktionsparameter, Daten, usw. enthalten. Aus diesem Grund wird die Verwendung des Begriffs *Objekt* auf alle im vorhergehenden aufgeführten Speicherbereiche erweitert (aus [28], Kapitel 1).

Jedes Objekt des verteilten Systems ist in einem bestimmten Rahmen abhängig von anderen Objekten. Die Objekte referenzieren sich gegenseitig und greifen lesend oder schreibend aufeinander zu. Dabei wird zwischen lokalen und globalen Objekten unterschieden. Lokale Objekte sind an einen Knoten gebunden und werden nur von lokalen Objekten referenziert. Globale Objekte hingegen besitzen globale Gültigkeit und können sowohl von globalen, als auch von lokalen Objekten referenziert werden. Diese Eigenschaft ist maßgeblich für die Bezeichnung der globalen Objekte als *Global Accessible Objects* (GAOs). Jedes GAO besitzt ein eindeutiges Identifizierungsmerkmal, das für alle ACVM Instanzen des verteilten Systems gültig ist. Wie bereits in der Einführung erwähnt, werden alle Knoten mittels des SSR Protokolls adressiert. Jedem GAO wird als Identifizierungsmerkmal eine eindeutige Adresse aus dem SSR Adressraum zugewiesen. Somit ist es prinzipiell möglich, GAOs in Verbindung mit dem SSR Protokoll zu adressieren. Des weiteren müssen Referenzen auf GAOs global eindeutig und immer gültig sein. Globale Referenzen sind vor allem für die Adressierbarkeit der GAOs von Bedeutung und werden in Kapitel 5 detailliert erläutert. Im Folgenden werden Referenzen, die im Zusammenhang mit GAOs verwendet werden, immer als globale Referenzen betrachtet und es wird auf eine explizite Hervorhebung verzichtet.

Die Speicherung eines GAOs auf einem Knoten des Netzwerks erzeugt eine Bindung zwischen der Adresse des GAOs und der Adresse des zugehörigen Knotens. Diese Bindung ist für die Referenzierung des GAOs von Bedeutung. Auf welchem Knoten das GAO gespeichert wird ist jedoch unabhängig von der Wahl der Adressen des GAOs und des Knotens. Falls ein GAO entlang einer Referenz auf ein anderes GAO zugreift und die Bindung des referenzierten GAOs zu dessen vorhaltenden Knoten im Vorfeld bekannt ist, kann mittels des SSR Protokolls der vorhaltende Knoten adressiert und somit auf das referenzierte GAO zugegriffen werden. Jedoch erlaubt

diese Zuweisung nicht das Migrieren des GAOs von einem Knoten zu einem anderen Knoten, da nach Abschluss einer Migration die ursprüngliche Bindung des migrierten GAOs nicht mehr gültig wäre. An dieser Stelle wird auf Kapitel 6, das sich mit dieser Problematik in ausführlicher Weise auseinandersetzt, verwiesen.

Ein GAO kann auf drei verschiedene Arten Referenzen erhalten. Zum ersten kann mit Hilfe einer global verfügbaren Datenbasis die Referenz auf ein GAO ermittelt werden. Diese Vorgehensweise ist jedoch nur für Referenzen auf statische Objekte sinnvoll. In [28] wird dafür die Verwendung eines *Orakels* vorgeschlagen, das als *Distributed Hash Table* (DHT) (siehe [3]) in Verbindung mit dem SSR Protokoll realisiert wird. Zum zweiten können GAOs globale Referenzen von anderen GAOs oder lokalen Objekten durch Funktionsparameter, Rückgabewerte von Funktionen oder über Variablen erhalten. Beispielsweise kann ein lokales Objekt eine Funktion eines GAOs aufrufen und dabei die Referenz auf ein Objekt übergeben. Dazu ist es aber notwendig, dass die Referenz dem lokalen Objekt im Vorfeld bekannt ist. Hierbei kann der Fall eintreten, dass ein GAO eine Referenz auf ein lokales Objekt erhält und dieses zu einem GAO *promoted* werden muss (aus [28], Kapitel 3). Als dritte Möglichkeit kann ein GAO ein neues GAO erzeugen und somit dessen Referenz bekommen. Für eine detaillierte Diskussion des *Global Object Space*, die die Themen GAOs, Promotion von lokalen Objekten, Verwaltung von GAOs innerhalb der ACVM, usw. zusammenfasst, wird auf die Diplomarbeit *Accessing remote objects in a distributed embedded Java VM* (siehe [17], Kapitel 4) von Alexander Kiening verwiesen.

Zum gegenwärtigen Zeitpunkt der Diplomarbeit ist die ACVM nur lokal lauffähig. Alle Komponenten, die für eine verteilte Ausführung der VM notwendig sind, fehlen bisher noch komplett. Dazu zählt der in dieser Diplomarbeit bearbeitete Aspekt der Adressierung und Migration von GAOs mit Hilfe des SSR Protokolls. Ohne einen entsprechenden Mechanismus ist es schwierig, weitere Komponenten, die darauf aufsetzen, zu realisieren und zu testen. Zum Beispiel ist die im vorhergehenden angesprochene verteilte Garbage Collection auf die Adressierbarkeit von GAOs angewiesen. Andere Komponenten sind wiederum relativ eng mit der Adressierung und Migration von GAOs verzahnt und müssen berücksichtigt werden. Ein verteilter *Software Transactional Memory* (STM) (siehe [30]), der die Synchronisation und Koordination nebenläufig ausgeführter Berechnungen garantiert, ist ein Beispiel dafür. Dies macht es erforderlich, dass für die Evaluation (siehe Kapitel 8) des Adressierungs- und Migrationsverfahrens die eigentliche ACVM und deren Funktionsweise abstrahiert wird.

Im Gesamten ergibt sich die in Abbildung 3.1 dargestellte Systemstruktur für einen Knoten des AmbiComp Projektes. Der Knoten ist mit Hilfe einer Kommunikationstechnologie an das AmbiComp Netzwerk angebunden. Ein „Programm“, bestehend aus der ACVM und dem SSR Protokoll, wird auf der Zielplattform ausgeführt. Das SSR Protokoll wurde im Vorfeld um den zur Adressierung und Migration von GAOs

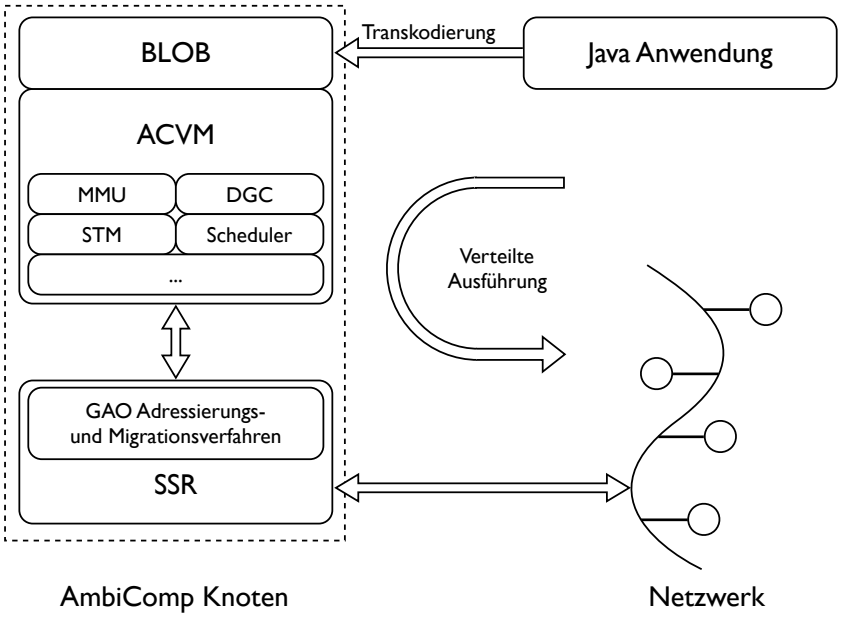


Abbildung 3.1: Generelle Systemstruktur

verwendeten Mechanismus erweitert. Die ACVM enthält die im vorhergehenden angesprochenen Komponenten MMU, DGC, STM, einen Scheduler, sowie nicht näher erläuterte Komponenten, die diverse Aufgaben übernehmen. Auf jeden Knoten mit einer ACVM Instanz werden die BLOBs transferiert und anschließend durch die ACVM ausgeführt. Ein BLOB repräsentiert ein Java Programm, das in einem vorhergehenden Übersetzungsschritt erzeugt wurde. Während der Ausführung des Java Programms, genauer gesagt des BLOBs, finden entsprechend der gespeicherten Referenzen Zugriffe auf die lokalen und globalen Objekte statt. Die Zugriffe auf GAOs werden durch das Adressierungs- und Migrationsverfahren auf das Netzwerk abgebildet.

Im weiteren Verlauf wird zunächst auf das Scalable Source Routing Protokoll eingegangen. Anschließend wird die Adressierbarkeit von GAOs, sowie die Probleme und möglichen Realisierungen bezüglich der Migration von GAOs erläutert. Ein ausgewähltes Adressierungs- und Migrationsverfahren wurde während dieser Diplomarbeit implementiert und evaluiert. Die Implementierung und die Ergebnisse der Evaluation werden in den Kapiteln 7 und 8 zusammengefasst.

4 Scalable Source Routing

Die Verwirklichung der Vision der *Ambient Intelligence* basiert auf der Kombination und Vernetzung zahlreicher eingebetteter Systeme, die als Knoten bezeichnet werden. Die Vernetzung geschieht dabei meist ad hoc und resultiert in großen und unstrukturierten Netzwerken. *Scalable Source Routing* (SSR) (siehe [12]) ist ein neuer Ansatz für das Routen in derartigen Netzwerken. Es ist speziell für gewachsene Netzwerke, die aus vielen mobilen Geräten bestehen, ausgelegt. Meistens stehen durch die Art der Knoten nur begrenzte Ressourcen zur Verfügung. SSR ist ein vollständiges Routing Protokoll, das direkt die Semantik eines strukturierten Peer-to-Peer Overlays bietet. Infolgedessen kann es als effiziente Basis für vollständig dezentralisierte Anwendungen dienen.

SSR kombiniert Source Routing im physikalischen Netzwerk mit einem an Chord angelehnten Routingverfahren. Das Source Routing ermöglicht dem Sender einen vollständigen Pfad, der vom Sender zum Empfänger führt, zu konstruieren. An Hand dieser Pfade leiten Knoten Nachrichten entsprechend im Netzwerk weiter. Chord ist ein Peer-to-Peer Netzwerk, das Nachrichten innerhalb eines virtuellen Rings weiterleitet. Für Details zu Chord wird an dieser Stelle auf [31] verwiesen. Der virtuelle Ring des SSR Netzwerks wird von allen Knoten gebildet. Dazu ist jedem Knoten ein virtueller Vorgänger- und Nachfolgerknoten zugeordnet. In Verbindung mit den virtuellen Nachbarn werden die zugehörigen Source Routen gespeichert. Die Distanz von Nachrichten zu ihrem Ziel auf dem virtuellen Ring nimmt durch einfaches Weiterleiten ab. Damit Nachrichten schnell ihr Ziel erreichen, werden physikalische Abkürzungen auf ihrem Weg bevorzugt (aus [12], Abstract).

Im Unterschied zu Chord ist SSR ein Protokoll der Vermittlungsschicht. Dies hat zur Folge, dass es kein darunter liegendes Verfahren zum Routen benötigt (siehe Abbildung 4.1). Dieser *Cross Layer* Ansatz reduziert, im Vergleich zu Overlay Routing Protokollen der Anwendungsschicht, sowohl die Komplexität der Implementation, als auch die Anforderungen bezüglich der Ressourcen (aus [10], Kapitel 3.1).

Im Gegensatz zu anderen Ansätzen wird die Skalierbarkeit nicht durch das Einführen künstlicher Hierarchien oder Zuweisung ortsbezogener Adressen erreicht. SSR ermöglicht die Kommunikation zwischen beliebigen Knoten innerhalb eines flachen Adressraums. Es ist jedoch nicht notwendig die Routen, die zu allen potentiellen Kommunikationspartnern führen, zu speichern. Jeder Knoten muss nur seine Nachbarn im virtuellen Ring und zusätzlich eine beschränkte Menge an Pfaden speichern (aus [12], Abstract).

Im Folgenden wird das Protokoll und die Arbeitsweise von SSR detailliert erläutert.

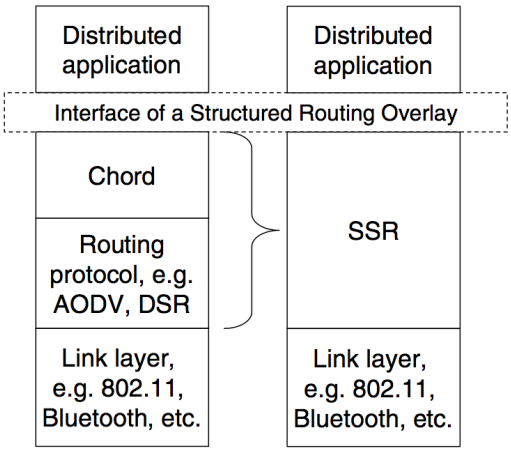


Abbildung 4.1: SSR *Cross Layer* Ansatz
Aus [10], Abbildung 1

4.1 Physikalische und virtuelle Struktur

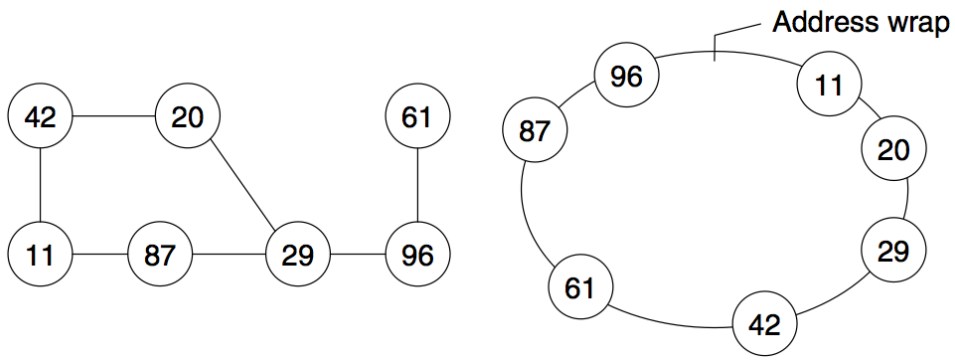
Die Schlüsselidee von SSR ist die zusätzliche Erweiterung der physikalischen Netzwerkstruktur mit einer virtuellen Ringstruktur. Dabei wird die virtuelle Struktur von den ortsunabhängigen Adressen der Knoten abgeleitet. Folglich ergeben sich bei Änderungen der physikalischen Netzwerktopologie keine Änderungen der virtuellen Struktur.

Abbildung 4.2 stellt die Kombination von physikalischer und virtueller Struktur dar. Jeder Knoten trägt eine ortsunabhängige Adresse und besitzt eine bestimmte Menge physikalischer Nachbarn (Abbildung 4.2(a)). Bezüglich der virtuellen Struktur können die Knotenadressen so betrachtet werden, dass diese einen virtuellen Ring bilden (Abbildung 4.2(b)). Zwei Knoten A, B des virtuellen Rings werden als direkte virtuelle Nachbarn bezeichnet, falls für A und B folgende Bedingung gilt:

Es existiert kein Knoten C, so dass $addr_v(A) < addr_v(C) < addr_v(B)$ gilt.

B wird dann als Nachfolger von A und A als Vorgänger von B bezeichnet. Darüber hinaus liefert $addr_v(A)$ die virtuelle Adresse des Knotens A. Bei dieser Betrachtung ist jedoch zu beachten, dass der Adressraum einen geschlossenen Ring bildet (aus [10], Kapitel 3.2).

Die Knotenadressen können einerseits selbst zugewiesene Adressen sein. Beim Zuweisen muss aber darauf geachtet werden, dass jede Adresse nur einmal vorkommt und eine Gleichverteilung gewährleistet ist. Andererseits können die Adressen auch von MAC Adressen, kryptographischen Schlüsseln oder globalen, einzigartigen Merkmalen, wie zum Beispiel dem Hash einer Anwendung, abgeleitet werden (aus [11], Kapitel 2).



(a) Physikalische Struktur (b) Virtuelle Struktur

Abbildung 4.2: Beispiel einer Netzwerktopologie
Aus [10], Abbildung 2

4.2 Weiterleitung von Nachrichten

SSR verwendet zwei Metriken für den Abstand zwischen zwei Knoten. Darauf basierend werden die Entscheidungen für das Weiterleiten von Nachrichten getroffen. Die physikalische Distanz d_p zwischen zwei Knoten A und B wird in Hops gemessen und ist gleich der Länge der kürzesten Source Route von A nach B. Hingegen entspricht die virtuelle Distanz d_v zwischen A und B dem absoluten Wert der numerischen Differenz der beiden Knotenadressen. Somit gilt: $d_v(A, B) = |addr_v(A) - addr_v(B)|$ (aus [12], Kapitel 2.3).

Im Gegensatz zu bekannten *Link State Routing* Protokollen und *Distance Vector Routing* Protokollen hält SSR keine Zustandsinformationen für alle Zielknoten im Netzwerk. Die Datenbasis eines Knotens, die für das Weiterleiten von Nachrichten verwendet wird, besteht aus den physikalischen Nachbarn, dem virtuellen Vorgänger und Nachfolger des Knotens, sowie bei Gelegenheit gespeicherten Source Routen im Route Cache (siehe Kapitel 4.3) (aus [12], Kapitel 2.2).

Im Folgenden wird darauf eingegangen, wie Knoten die Source Routen zu unbekannten Zielen dynamisch aufbauen können. Beim *Dynamic Source Routing* (siehe [16]) müssen die Startknoten die gesamte Source Route bis hin zum Zielknoten konstruieren. Dies ist bei SSR nicht der Fall. SSR kombiniert den Ansatz der Source Routen mit einem dazwischenliegenden „Nachschlagen". Das bedeutet, falls ein Knoten nicht die komplette Source Route in seinem Cache gespeichert hat, dann erstellt dieser eine Source Route zu einem anderen Knoten. Dieser wird als *Zwischenknoten* bezeichnet. Die konstruierte Source Route wird an die bereits bestehende Source Route angefügt und die Nachricht entsprechend an den neuen Zwischenknoten weitergeleitet. Am Ende der Source Route kümmert sich der neue Zwischenknoten wie-

derum um die Nachricht. Das soeben beschriebene Verfahren konvergiert, wobei die volle Source Route vom Start- bis hin zum Zielknoten erzeugt wird.

Die Auswahl von Zwischenknoten geschieht mit Hilfe eines dreistufigen Prozesses (siehe [10], Kapitel 3.3), bei dem drei Bedingungen in strikter Reihenfolge ausgewertet werden. Hierbei bezeichnet, wie zu Beginn dieses Abschnitts erläutert, $d_v(A, B)$ die virtuelle Distanz und $d_p(A, B)$ die physikalische Distanz zwischen zwei Knoten A und B. Ein Knoten I_n wählt den nächsten Zwischenknoten I_{n+1} in Bezug auf den Zielknoten D, so dass gilt:

1. $d_v(I_{n+1}, D) < d_v(I_n, D)$
2. $d_p(I_n, I_{n+1}) = min$
3. $d_v(I_{n+1}, D) = min$

In Worten bedeutet dies, dass ein Knoten als der nächste Zwischenknoten I_{n+1} gewählt wird, falls dieser virtuell näher am Zielknoten D liegt als der momentane Knoten I_n (1. Bedingung). Gibt es mehrere Knoten, die dieses Kriterium erfüllen, so werden diejenigen Knoten ausgewählt, die mit der geringsten Anzahl von Hops (vom aktuellen Knoten aus) erreicht werden (2. Bedingung). Erfüllen mehrere Knoten auch die 2. Bedingung, so wird letztlich derjenige Knoten I_{n+1} ausgewählt, der die virtuelle Distanz zum Zielknoten minimiert (3. Bedingung).

Bei der Betrachtung der Konsistenz des Mechanismus ist zu erkennen, dass die virtuelle Distanz hin zum Zielknoten mit jedem Schritt abnimmt. Falls jeder Knoten mindestens eine Source Route zu seinem Vorgänger und Nachfolger im virtuellen Ring kennt, terminiert der im vorhergehenden beschriebene Algorithmus ausschließlich und garantiert am Zielknoten. Es gilt an dieser Stelle zu erwähnen, dass die somit erzeugten Source Routen nicht notwendigerweise die kürzesten Pfade darstellen. Jedoch kann jeder Knoten versuchen, beim Anfügen von Source Routen die Länge zu verkürzen. Dieser Vorgang wird als *Pruning* bezeichnet und wird im nachfolgenden Kapitel in Verbindung mit dem Route Cache erläutert (aus [10], Kapitel 3.3).

4.3 Route Cache

Die Datenbasis für das Weiterleiten von Nachrichten beinhaltet der Route Cache in Form von Source Routen. Der Route Cache besitzt eine beschränkte Größe und solange dieser nicht vollständig gefüllt ist, können neue Source Routen hinzugefügt werden. Bei vollem Cache werden die Einträge nach der *Least Recently Used* (LRU) Strategie verdrängt. Des weiteren werden die beiden Routen zum virtuellen Vorgänger und Nachfolger im Cache „festgehalten“. Somit wird verhindert, dass bei einer Kapazitätsüberschreitung eine der beiden Routen verdrängt wird. Dies ist nötig um den virtuellen Ring aufrecht zu erhalten (aus [11], Kapitel 2).

Eine fundamentale Operation, die im Route Cache implementiert ist, wird als *Source Route Concatenation* bezeichnet. Während des Weiterleitens einer Nachricht - von der Quelle zum Ziel - fügen die virtuellen Zwischenknoten jeweils Fragmente der Route hinzu. In vielen Fällen beinhaltet ein hinzugefügtes Fragment einen Knoten, der bereits Teil der Source Route der Nachricht ist. In diesem Fall kann die Source Route beschnitten werden (= *Pruning*), so dass die daraus resultierende Pfadlänge kürzer als die Summe der einzelnen Pfadlängen ist.

Beim Konkatenieren von zwei Source Routen an einem Zwischenknoten ist es in der Praxis möglich, dass dieser durch das Pruning aus der resultierenden Source Route entfernt wird. Die Nachrichten enthalten daher nicht nur die Source Route sondern einen zusätzlichen *Stub*-Pfad. Dieser verbindet den letzten Zwischenknoten, der entfernt wurde, mit der eigentlichen Source Route. Nach dem Erreichen der Hauptroute muss der Stub jedoch in der Nachricht erhalten bleiben, da einzelne Knoten ausfallen können und somit die Route Caches aktualisiert werden müssen. Falls eine Route auf Grund eines Knotenausfalls unterbrochen sein sollte, wird der letzte Zwischenknoten, erreichbar über den Stub, mit einer *Updatenachricht* informiert. Es wird dann die betreffende Source Route aus dem Cache des Zwischenknotens gelöscht. Eine weitere Updatenachricht wird erzeugt und an den Zwischenknoten gesendet, wenn der zuvor ausgefallene Knoten wieder erreichbar ist.

Zusammen mit den Updatenachrichten kommt das SSR Protokoll ohne reguläre Sondierungen aus. Falls eine Source Route unbemerkt ungültig wird, verweilt sie weiterhin im Route Cache bis sie verwendet wird und durch deren Verwendung den Versand einer Updatenachricht auslöst. Als Konsequenz folgt daraus, dass *Scalable Source Routing* keine Kontrollnachrichten erzeugt, solange keine Nutzdaten übertragen werden (aus [12], Kapitel 2.5).

5 Adressierung von GAOs

Dieses Kapitel befasst sich mit der Adressierbarkeit von *Global Accessible Objects* (GAOs). Es wird ein Verfahren beschrieben, mit dem jedes GAO, das für die Ausführung eines Java Programms durch die ACVM notwendig ist, garantiert erreicht werden kann. Jegliche Kommunikation, die zwischen den einzelnen Knoten des Netzwerks stattfindet, basiert auf einem Austausch von Nachrichten. Die Nachrichten werden mit Hilfe des SSR Protokolls vom Quell- zum Zielknoten geroutet. Nachfolgende Betrachtungen basieren auf der Annahme, dass Nachrichten verlustfrei im Netzwerk übertragen werden. Falls dies jedoch nicht garantiert werden kann, müssen für relevante Operationen zusätzliche Nachrichten als Bestätigungen in geeigneter Art und Weise versendet werden.

Zunächst gilt es zu klären, wie und auf welche lokalen Objekte und GAOs während der Ausführung eines Java Programms zugegriffen wird. Darüber hinaus ist es von Bedeutung, wie die einzelnen Objekte voneinander abhängig sind. Um dies zu erläutern, wird im Folgenden die Ausführung einer Java Anwendung durch die ACVM betrachtet. Zur Vereinfachung beschränkt sich die Betrachtung vorerst auf die lokale Ausführung einer Anwendung. Das bedeutet, dass zunächst keine GAOs auf Knoten existieren.

5.1 Referenzgraphen

Die Lebenszeit eines Objekts beginnt frühestens mit dem Start einer Java Anwendung durch die ACVM. Objekte repräsentieren, wie im Kapitel 3 beschrieben, Programmcode, Threads, Java Objekte und Java Arrays. Nach dem Start der Anwendung und während der Ausführung durch die ACVM existiert mindestens ein Thread. Dies ist der sogenannte *Hauptthread*, der auf den auszuführenden Programmcode verweist. Mit dem Hauptthread beginnend, werden potenziell weitere Java Threads, Java Objekte und Java Arrays erzeugt. Im Allgemeinen gilt, dass ein Thread Referenzen auf Programmcode, Java Objekte und Java Arrays speichern kann. Programmcode hingegen referenziert keine anderen Objekte. Darüber hinaus gilt für Java Objekte und Java Arrays, dass sie sich gegenseitig referenzieren können. Welche Objekte wiederum andere Objekte referenzieren ist dabei abhängig von der Struktur des ausgeführten Programms.

Die Referenzen zwischen den Objekten sind gerichtet. Unter Beachtung dieser Tatsache ergibt sich ein gerichteter Graph, der im Folgenden als *Referenzgraph* bezeichnet

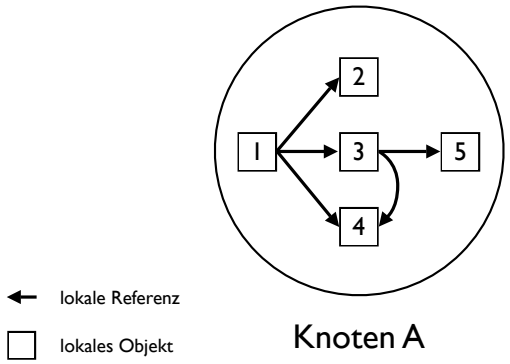


Abbildung 5.1: Lokaler Referenzgraph

wird. Jede durch die ACVM ausgeführte Java Anwendung besitzt einen eigenen Referenzgraphen, der sich programmbedingt dynamisch verändern kann. Das bedeutet, dass während der Laufzeit Objekte erzeugt und gelöscht werden, sowie vorhandene Objekte des Referenzgraphen ihre intern gespeicherten Referenzen verändern. Dies hat zur Folge, dass gerichtete Referenzen des Referenzengraphen auf neue, auf bereits bestehende oder auf keine Objekte mehr verweisen. Des weiteren kann es vorkommen, dass Objekte nicht mehr referenziert werden.

Nicht mehr referenzierte Objekte können prinzipiell gelöscht werden. An dieser Stelle muss zwischen den Objekttypen unterschieden werden. Java Objekte und Java Arrays werden automatisch durch die Garbage Collection komplett aus den somit isolierten Teilen des Referenzgraphen entfernt. Hingegen darf nicht mehr referenzierter Programmcode nicht gelöscht werden, da dieser zu einem späteren Zeitpunkt von neuen Threads benötigt werden kann. Darüber hinaus werden Threads nach deren Beendigung automatisch durch die Laufzeitumgebung gelöscht.

Im Folgenden bezeichnet LO_x ein lokales Objekt und GAO_y ein globales Objekt, mit dem lokal eindeutigen Identifizierungsmerkmal x bzw. mit dem global eindeutigen Identifizierungsmerkmal y. Abbildung 5.1 zeigt ein Beispiel eines Referenzgraphen mit fünf lokalen Objekten. Alle fünf Objekte sind auf dem Knoten A gespeichert. Falls LO_1 den Hauptthread repräsentiert, wird die zugehörige Java Anwendung lokal auf Knoten A ausgeführt. Zum Hauptthread existiert der auszuführende Programmcode, der in LO_2 gespeichert wird. Während des bisherigen Ausführungsverlaufs wurden beispielsweise weitere Java Objekte, genauer gesagt die lokalen Objekte LO_3, LO_4 und LO_5, erzeugt. LO_1 hält direkte Referenzen auf LO_2, LO_3 und LO_4, jedoch keine Referenz auf LO_5. Diese und die Referenz auf LO_4 wurden LO_3 während der Laufzeit übergeben.

Die Promotion eines lokalen Objekts des Referenzgraphen zu einem GAO, macht es erforderlich, dass alle von diesem GAO referenzierten lokalen Objekte ebenfalls zu GAOs promoted werden (siehe Kapitel 3). Die somit erzeugten GAOs können mit Hilfe des im nachfolgenden Kapitel beschriebenen Migrationsverfahrens im Netzwerk

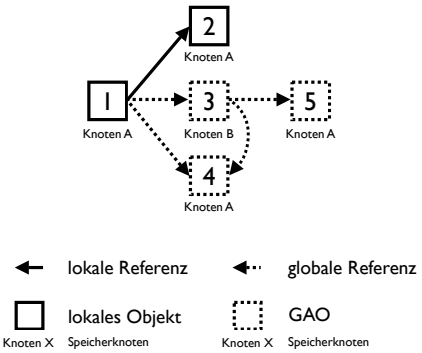


Abbildung 5.2: Referenzgraph mit GAOs

verteilt werden. Für die Beschreibung des Adressierungsverfahrens genügt es jedoch anzunehmen, dass die GAOs des Referenzgraphen bereits beliebig auf Knoten des Netzwerks verteilt wurden. Darüber hinaus ist es möglich, dass beim Starten der Anwendung statische Objekte als GAOs existieren und „beliebig" bzw. auf eine im Vorfeld festgelegte Art und Weise auf die Knoten verteilt wurden.

Wird beispielsweise LO_3 aus dem vorhergehenden Beispiel zu einem GAO promoted, so müssen ebenfalls LO_4 und LO_5 zu GAOs promoted werden. Durch eine anschließende Migration kann das durch die Promotion entstandene GAO_3 auf Knoten B übertragen werden. Es ergibt sich daraus der in Abbildung 5.2 gezeigte Referenzgraph, der nicht mehr lokal beschränkt ist.

5.2 Adressierungsverfahren

Die Bedingung für die korrekte Ausführung einer verteilten Anwendung ist die Erreichbarkeit aller Objekte, da potentiell jedes Objekt entlang der Kanten des Referenzgraphen auf andere Objekte zugreifen kann. Zugriffe können hierbei lesend und schreibend erfolgen. Mit Bezug auf die ACVM und der Ausführung von Threads bedeutet dies, dass ein Thread, genauer gesagt dessen Ausführungskontext, Referenzen enthält und an Hand dieser auf Objekte zugreifen kann. Die Referenzen können dabei einerseits von dem Thread selbst und andererseits von Objekten, auf die bereits während der Laufzeit des Threads zugegriffen wurde, stammen.

Alle ausgehenden, gerichteten Referenzen müssen für ein Objekt bekannt sein. Für den Fall von lokal referenzierten Objekten sind dies die lokalen Speicheradressen. Im Gegensatz dazu muss für ein referenziertes GAO zusätzlich zur eindeutigen SSR-Adresse des GAOs auch die eindeutige SSR-Adresse des Knotens, der das referenzierte GAO vorhält, gespeichert werden. Dieser Knoten wird als *Home Node* des zugehörigen GAOs bezeichnet. Somit ist es möglich, dass ein GAO unabhängig von dessen Adresse auf einem beliebigen Knoten vorgehalten werden kann.

Jeder Knoten speichert eine Art Abbildungstabelle, die die Zugehörigkeit von referenzierten GAOs zu Home Nodes auflistet. Zusätzlich muss zu jedem Eintrag der Abbildungstabelle gespeichert werden, wie viele Referenzen zu diesem Eintrag auf dem Knoten existieren. Ohne diese Information ist es nicht möglich zu entscheiden, ob nach einer Dereferenzierung eines Objekts der Eintrag entfernt werden darf oder nicht.

Ein Zugriff auf ein GAO, das auf einem entfernten Home Node gespeichert ist, läuft im Allgemeinen folgendermaßen ab. Zunächst wird die SSR-Adresse des Home Nodes mit Hilfe der Abbildungstabelle ermittelt. Anschließend wird eine Nachricht mittels des im Kapitel 4.2 beschriebenen SSR Routing Algorithmus zum Home Node geroutet. Schließlich wird die empfangene Nachricht am Home Node verarbeitet und gegebenenfalls eine Nachricht als Antwort versendet. Bei dieser Vorgehensweise muss die gesendete Nachricht im Schnitt von $O(\log(N))$ Knoten (siehe [12], Kapitel 2.1) weitergeleitet werden.

Je größer die Anzahl der Knoten im Netzwerk ist, desto wahrscheinlicher ist es, dass keiner der weiterleitenden Knoten in Verbindung mit dem zur Nachricht gehörenden Referenzgraphen steht. Umso mehr unabhängige Referenzgraphen auf dem Netzwerk aufsetzen, desto größer wird die durch die Weiterleitungen hervorgerufene Last im Gesamtnetzwerk. Darüber hinaus ist die Latenzzeit einer Nachricht abhängig von der Anzahl der für das Routing dieser Nachricht benötigten physikalischen Hops. Je mehr Hops pro Nachricht für die Zugriffe auf entfernt gespeicherte GAOs gebraucht werden, desto langsamer wird letztendlich die Anwendung ausgeführt.

Zur Vermeidung von hohen Latenzzeiten und zur Umgehung von unnötiger Verkehrslast im Gesamtnetzwerk ist die Verwendung zusätzlicher Informationen notwendig. Jeder Knoten speichert, wie bereits oben beschrieben, für alle vorgehaltenen GAOs die SSR-Adressen der Home Nodes der referenzierten GAOs. Als Verbesserung wird zusätzlich für jedes referenzierte GAO die Source Route gespeichert, die vom aktuell betrachteten Knoten zu dessen Home Node führt. Mit Hilfe der gespeicherten Source Routen kann somit ein Objekt alle direkt referenzierten GAOs ohne Verwendung des SSR Routing Algorithmus erreichen. Dafür genügt es, dass die Nachrichten entlang der gespeicherten Source Routen weitergeleitet werden. Nachfolgende Abbildung 5.3 stellt den oben gezeigten Referenzgraphen in Kombination mit Source Routen dar.

Auf Grund der Dynamik des Netzwerks und der damit einhergehenden Topologieveränderungen kann es vorkommen, dass Source Routen nicht mehr gültig sind. In diesem Fall muss eine neue Source Route hin zum Zielknoten gefunden werden. Es wird dabei auf den SSR Routing Algorithmus ausgewichen, wobei aber für die betreffende Nachricht die angeführten Nachteile der höheren Latenz und der unnötigen Verkehrslast auftreten. Ebenso ist es möglich, dass sich auf Grund von Topologieveränderungen bessere Source Routen für bereits gespeicherte Referenzen ergeben. Dies macht es nötig, dass beim Empfangen von Nachrichten, die mit der Adressierung von GAOs in Verbindung stehen, jeweils die Source Routen auf potentielle Verbesserungen überprüft werden müssen.

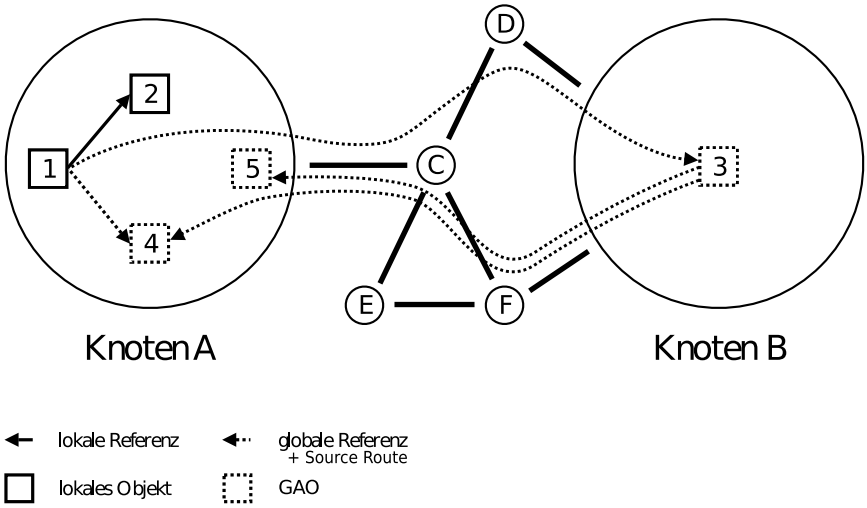


Abbildung 5.3: Referenzgraph mit GAOs unter Verwendung von Source Routen

Die Speicherung der Source Routen kann einerseits explizit in Verbindung mit den Referenzen und andererseits in Verbindung mit dem SSR Route Cache (siehe Kapitel 4.3) geschehen. Die explizite Speicherung bedeutet, dass die Source Routen unabhängig bzw. getrennt vom Route Cache gespeichert werden. Dies hat den Vorteil, dass für alle ausgehenden Referenzen alle Source Routen vorhanden sind. Jedoch wird bei einer großen Anzahl und Länge der Source Routen entsprechend viel zusätzlicher Speicher verbraucht. Dieser Nachteil wird umgangen, wenn die Source Routen innerhalb des Route Caches gespeichert werden. Die Größenbeschränkung des Route Caches führt dazu, dass nicht alle Source Routen der ausgehenden Referenzen gespeichert werden können. Je nach Anzahl ausgehender Referenzen pro Knoten und Häufigkeit der Verwendung, ist es nötig, den SSR Routing Algorithmus zu verwenden. Dabei treten wiederum die im vorhergehenden beschriebenen Nachteile auf. Für alle nachfolgenden Betrachtungen wird vorausgesetzt, dass die zu Referenzen zugehörigen Source Routen im Route Cache gespeichert werden.

Zugriffe auf entfernt vorgehaltene GAOs können sowohl lesend, als auch schreibend durchgeführt werden. Ob bei einem Zugriff das komplette GAO oder nur bestimmte Bereiche relevant sind, betrifft den internen Aufbau der GAOs. Der Aufbau der GAOs wird durch die Implementierung der ACVM festgelegt und ist für die Adressierung nicht von Bedeutung. Es gilt jedoch zu beachten, dass bei einer Unterteilung der GAOs die Zugriffsnachrichten erweitert werden müssen. Die Erweiterung umfasst diejenigen Informationen, die für die Zugriffe auf einzelne Bereiche der GAOs notwendig sind.

Für lesende Zugriffe wird auf dem Home Node des initiierenden Objekts zunächst überprüft, ob das zugegriffene Objekt lokal gespeichert ist. Ist dies der Fall, so terminiert der Zugriff, indem das lokale Objekt zurückgegeben wird. Andernfalls wird

mittels der gespeicherten Referenz, die auf das zu lesende GAO verweist, eine Nachricht erzeugt. Diese Nachricht wird entlang der Source Route zu dem Home Node des zu lesenden GAOs geroutet. Der Empfang der Nachricht auf dem Zielknoten initiiert einen Zugriff auf das zu lesende GAO. Nach einem erfolgreichen Zugriff wird das GAO und die mit diesem GAO gespeicherten Referenzen, genauer gesagt die Zuordnungen der referenzierten GAOs zu deren Home Nodes und optional die zugehörigen Source Routen, mit einer neuen Nachricht entlang der invertierten Referenz bzw. Source Route zurückgesendet. Am ursprünglichen Knoten angekommen, wird das empfangene GAO an das lesend zugreifende Objekt übergeben. Zusätzlich werden die mit den Referenzen übertragenen Zuordnungen und Source Routen in die lokalen Datenstrukturen eingefügt.

Es sei an dieser Stelle angemerkt, dass bei der Verwendung von Source Routen eine Invertierung nicht garantiert werden kann, falls das Netzwerk unidirektionale Verbindungen besitzt. Sollte dies der Fall sein, so muss auf den SSR Routing Algorithmus ausgewichen werden.

Ein schreibender Zugriff verläuft ähnlich zu einem lesenden Zugriff. Greift ein Objekt schreibend auf ein GAO zu, so wird zunächst überprüft, ob dieses lokal gespeichert ist. Falls dies der Fall ist, so wird das lokal gespeicherte GAO mit den veränderten Daten überschrieben bzw. aktualisiert. Ist das GAO nicht lokal gespeichert, so wird eine Nachricht, die das veränderte GAO bzw. nur dessen veränderte Daten beinhaltet, erzeugt und mit Hilfe der gespeicherten Referenz gesendet. Der Empfang der Nachricht am Home Node des schreibend zugegriffenen GAOs löst schließlich dessen Aktualisierung aus und beendet den schreibenden Zugriff des Objekts.

Mit Bezug auf die ACVM muss für lesende und schreibende Zugriffe auf lokale und globale Objekte eine Synchronisation gewährleistet werden, um Nebenläufigkeiten auszuschließen. Das heißt die Reihenfolge der Zugriffe darf nicht beliebig sein, da ansonsten die Berechnungsergebnisse nicht deterministisch und mitunter von den Latenzzeiten des Netzwerks abhängig sind. Die Synchronisation kann mittels verschiedener Mechanismen erzielt werden. In Kapitel 3 wurde bereits *Software Transactional Memory* als eine Realisierungsmöglichkeit angesprochen. Der Aspekt der Nebenläufigkeit wird in dieser Arbeit jedoch nicht betrachtet.

Bisher wurde darauf eingegangen, wie GAOs adressiert werden können. Dabei wurde angenommen, dass GAOs bereits auf mehrere Knoten im Netzwerk verteilt wurden. Einerseits kann die Verteilung von GAOs vor der Laufzeit der betreffenden Anwendung festgelegt sein. Andererseits gibt es die Möglichkeit, dass GAOs gezielt zwischen den Knoten migrieren können. Nachfolgendes Kapitel beschreibt die dabei entstehenden Probleme und potentielle Lösungsansätze.

6 Migrationsverfahren

Eine Anforderung an *Ambient Intelligence* ist die adaptive Anpassung des Systems an die gegenwärtige Situation. Dazu gehören zum Beispiel der Austausch von Sensorinformationen und parallel ausführbare Berechnungen. Des weiteren soll es möglich sein, einzelne Komponenten aus dem ambienten System zu entfernen ohne dabei die Funktionalität des Gesamtsystems zu gefährden. Darüber hinaus kann es erforderlich sein, dass ein Knoten auf Grund von Ressourcenbeschränkungen selten verwendete Objekte „auslagert". Ebenso ist es sinnvoll, viele Zugriffe auf ein Objekt lokal durchzuführen, anstatt wiederholt auf entfernt vorgehaltene Objekte zuzugreifen. Auf Grund dessen ist es erforderlich, dass GAOs von den Home Nodes, auf denen sie vorgehalten werden, entfernt und auf besser geeigneten Knoten gespeichert werden. Dieser Vorgang wird als Migration bezeichnet.

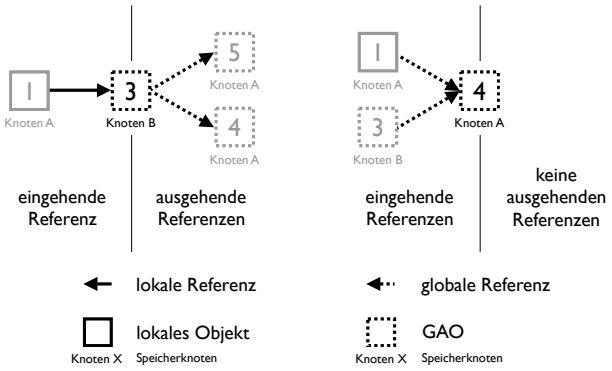
Das aktuell betrachtete Kapitel befasst sich mit der Migration von GAOs. Es werden zunächst die dabei auftretenden Probleme hervorgehoben. Anschließend werden mehrere Lösungsansätze für Probleme, die durch Migration entstehen, erläutert und abschließend gegenübergestellt. Es finden dabei aber keine Diskussionen bezüglich den Fragestellungen

- wann werden Migrationen initiiert,
- wohin werden GAOs migriert und
- welche Strategien bzw. Heuristiken gibt es für Migrationen

statt. Daher wird angenommen, dass eine übergeordnete Instanz, im Folgenden als *Migrationsinstanz* bezeichnet, die Migrationen veranlasst. Wie diese Instanz aussieht, spielt jedoch keine Rolle. Der zentrale Punkt der Betrachtung ist die Funktionsweise des Mechanismus, der für die Migrationen verantwortlich ist.

Eine Migration findet immer entlang einer Source Route statt. Diese Source Route wird als *Migrationsroute* bezeichnet und verbindet den ursprünglichen Home Node mit dem neuen Home Node. Wenn die Migration eines GAOs durchgeführt wird, muss der Migrationsinstanz die Migrationsroute im Vorfeld bekannt sein. Der Mechanismus zum Auffinden der Source Route ist aber für die eigentliche Migration bedeutungslos.

Als Beispiel sei an dieser Stelle der SSR Routing Algorithmus (siehe Kapitel 4.2) angeführt, mit dessen Hilfe die gesuchte Migrationsroute gefunden werden kann. Dazu wird vom Home Node, von dem ein GAO migriert werden soll, eine Nachricht an den Knoten, der das Ziel der Migration darstellt, gesendet. Der Zielknoten antwortet wiederum mit einer Nachricht, die die invertierte Source Route der ersten Nachricht



(a) GAO_3 (b) GAO_4

Abbildung 6.1: Ein- und ausgehende Referenzen

enthält. Empfängt der Home Node die Antwortnachricht, erhält dieser automatisch die Migrationsroute, die von der Migrationsinstanz verwendet werden kann. Für die weiteren Betrachtungen von Migrationen wird die jeweils benötigte Migrationsroute als im Vorfeld gegeben angenommen.

Nachdem eine Migration durch die Migrationsinstanz veranlasst wurde, wird zunächst vom ursprünglichen Home Node eine Nachricht an den neuen Home Node entlang der Migrationsroute gesendet. Diese Nachricht enthält zum einen das zu migrierende GAO und zum anderen dessen ausgehende Referenzen. Am neuen Home Node angekommen, wird das GAO und die Referenzen gespeichert.

Nach Abschluss einer Migration befindet sich das Gesamtsystem in einem inkonsistenten Zustand. Der Grund dafür liegt darin, dass jedes GAO des Referenzgraphen, genauer gesagt dessen Home Node, eine Menge von ausgehenden Referenzen speichert. Daraus folgt, dass zu einem GAO auch eine bestimmte Menge von eingehenden Referenzen assoziiert ist. Die eingehenden Referenzen setzen sich aus einer Teilmenge von ausgehenden Referenzen anderer GAOs und lokaler Objekte zusammen. Abbildung 6.1 zeigt, ausgehend von dem in Abbildung 5.2 dargestellten Referenzgraphen, die zugehörige Menge der ein- und ausgehenden Referenzen für GAO_3 und GAO_4.

Eine Migration endet mit der Problemstellung, dass alle ein- und ausgehenden Referenzen des migrierten GAOs nicht mehr gültig sind. Im Detail bedeutet dies für eingehende Referenzen, dass die Zuordnung des aktuell migrierten GAOs zu dessen Home Node nicht mehr stimmt. Zusätzlich kann nicht garantiert werden, dass jedem betreffenden Knoten eine Source Route, die zum neuen Home Node führt, bekannt ist. Bei ausgehenden Referenzen sind die zugehörigen Source Routen inkonsistent, da deren Startknoten der ursprüngliche Home Node ist. Jedoch behalten die Zuordnungen von referenzierten GAOs zu Home Nodes ihre Gültigkeit. Somit ist es erforderlich, dass nach einer Migration Maßnahmen getroffen werden bzw. der

Migrationsmechanismus so gewählt wird, dass die Referenzen und Source Routen korrigiert werden können.

Im Folgenden wird zunächst ein Verfahren erläutert, mit dem die ausgehenden Referenzen, genauer gesagt die Source Routen, wieder ihre Gültigkeit erhalten. Anschließend werden im Kapitel 6.2 Maßnahmen bezüglich ungültiger, eingehender Referenzen diskutiert.

6.1 Ausgehende Referenzen

In Anbetracht der Tatsache, dass die Migrationsroute am neuen Home Node bekannt ist, können die Source Routen, die zu den ausgehenden Referenzen des migrierten GAOs gehören, wieder in einen konsistenten Zustand überführt werden. Dazu muss die invertierte Migrationsroute mit allen ausgehenden Source Routen konkateniert werden:

$$\text{outgoingSourceRoute}_{new} = \text{(-migrationRoute)} + \text{outgoingSourceRoute}_{old}$$

Dies hat zur Folge, dass bei einem Zugriff auf ein migriertes GAO, entlang einer aktualisierten ausgehenden Referenz bzw. Source Route, zunächst die Nachricht an den ursprünglichen Home Node gesendet wird. Dies geschieht entlang der invertierten Migrationsroute. Ab dem Home Node wird die ursprüngliche Source Route traversiert. Befand sich die ausgehende Referenz vor der Migration in einem konsistenten Zustand, so ist garantiert, dass der Zielknoten, der das referenzierte GAO vorhält, erreicht werden kann. Für die garantierte Erreichbarkeit wird angenommen, dass sich die Topologie des Netzwerks nicht geändert hat. Sollte es jedoch der Fall sein, dass Source Routen sind nicht mehr gültig sind, so muss auf den SSR Routing Algorithmus zurückgegriffen werden.

Werden entgegen der Annahme Source Routen explizit gespeichert, so kann nach einer Migration die Verbesserung der gespeicherten, ausgehenden Source Routen angestoßen werden. Es kann vorkommen, dass der neue Home Node „Abkürzungen“ zu Knoten innerhalb der jeweilig betrachteten Source Route kennt. Somit kann diese mit der Abkürzung kombiniert werden und resultiert schließlich in einer kürzeren Source Route. Während jeder Kombination von Source Routen ist darauf zu achten, dass Schleifen und Abkürzungen erkannt werden. Dementsprechend können Teile der Source Routen verworfen werden. Werden hingegen die Source Routen im Route Cache (siehe Kapitel 4.3) gespeichert, so ergibt sich die Verbesserungen der Source Routen automatisch durch die Struktur des Route Caches.

Die Wiederherstellung der Konsistenz bezüglich ausgehender Referenzen nach einer Migration erzeugt keine Last innerhalb des Netzwerks. Es müssen nur die notwendigen Konkatenationen und eventuellen Verbesserungen der Source Routen auf dem neuen Home Node angestoßen werden. Der dadurch für den betreffenden Knoten

erzeugte Arbeitsaufwand ist durch die Anzahl und Länge der zu den ausgehenden Referenzen gehörigen Source Routen des migrierten GAOs beschränkt.

6.2 Eingehende Referenzen

Wie bereits am Anfang des Kapitels erläutert, besitzt jedes GAO eine bestimmte Menge an eingehenden Referenzen. Nach der Migration eines GAOs befindet sich die Menge der eingehenden Referenzen in einem inkonsistenten Zustand. Das bedeutet, dass für alle eingehenden Referenzen auf das migrierte GAO die Zuordnung zu dessen Home Node nicht mehr gültig ist. Für die zugehörigen Source Routen gilt, dass diese garantiert zum vorhergehenden Home Node des migrierten GAOs führen. Jedoch ist es fraglich, ob Source Routen zum neuen Home Node bekannt sind.

Nachfolgende Abbildung stellt den Referenzgraphen aus Kapitel 5 nach der Migration von GAO_4 dar. GAO_4 wurde ausgehend von Knoten A entlang der Migrationsroute $A - C - D$ zu Knoten D migriert. Diese Migration resultiert in einem inkonsistenten Zustand, da LO_1 und GAO_3 Referenzen auf GAO_4 besitzen und auf Knoten A verweisen.

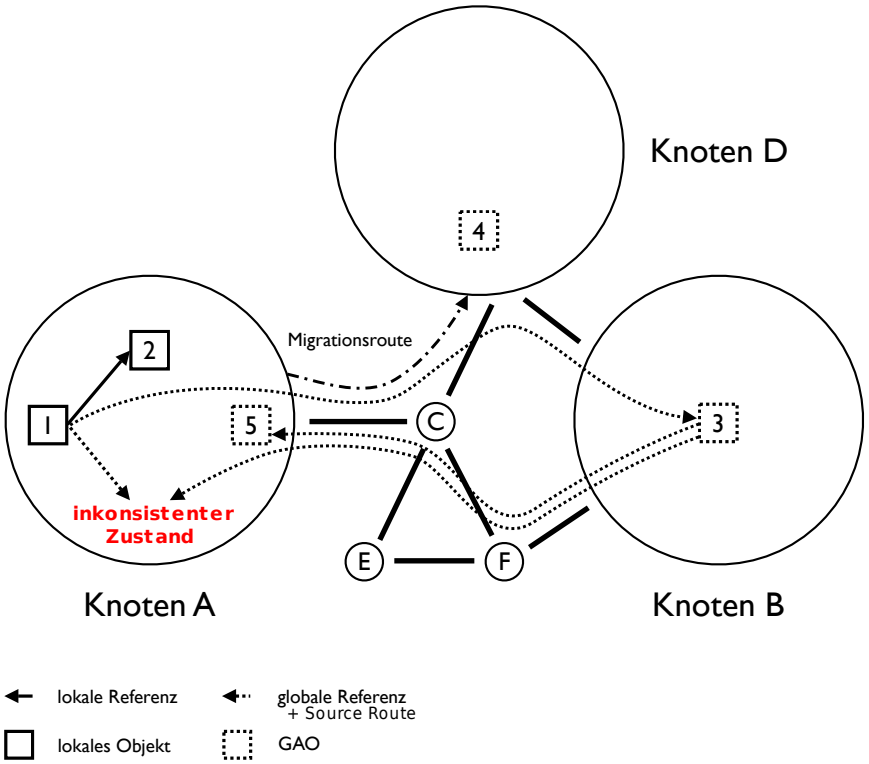


Abbildung 6.2: Inkonsistenter Zustand nach Migration von GAO_4

Die Anpassung lokal gespeicherter Objekte mit ausgehenden Referenzen auf das migrierte GAO stellt kein Problem dar. Diese Referenzen werden durch einen Eintrag in der Abbildungstabelle repräsentiert. Zur Anpassung muss nach der Migration die Zuordnung des migrierten GAOs zum neuen Home Node in der Abbildungstabelle aktualisiert werden. Zusätzlich wird die Migrationsroute im Route Cache gespeichert. Dies ist erforderlich, damit für spätere Zugriffe der lokalen Objekte, die das migrierte GAO referenzieren, eine Source Route, die zum neuen Home Node des migrierten GAOs führt, verwendet werden kann. Der Arbeitsaufwand ist somit durch die Operation *insert()* für die Abbildungstabelle und den Route Cache beschränkt. In Abbildung 6.2 trifft dies auf LO_1 zu.

Zur Wiederherstellung der Konsistenz des Gesamtsystems müssen schließlich alle eingehenden Referenzen, die von nicht lokal gespeicherten Objekten ausgehen, angepasst werden. Dies ist jedoch nicht möglich, da aus der Sicht des alten und neuen Home Nodes nicht bekannt ist, welche GAOs im System Referenzen auf das aktuell migrierte GAO besitzen. Es ist somit erforderlich, zusätzlichen Aufwand zu betreiben, damit schließlich nach Migrationen garantiert auf alle referenzierten GAOs zugegriffen werden kann. Diesbezüglich gibt es mehrere Lösungsansätze, die als *Migrationsverfahren* bezeichnet werden. Die Lösungsansätze beeinflussen die Art und Weise des Ablaufs der Migrationen und die damit einhergehenden Maßnahmen zur Gewährleistung der Konsistenz. Daraus folgt eine garantierte, permanente Adressierbarkeit von migrierten GAOs. Dabei unterscheiden sich diese zum einen durch den jeweils benötigten Mehraufwand bezüglich des verwendeten Speichers und zum anderen durch den zusätzlichen Kommunikationsaufwand zwischen den einzelnen Knoten des Netzwerks.

Nachfolgend werden vier Migrationsverfahren erläutert, die das Problem von inkonsistenten, eingehenden Referenzen lösen bzw. umgehen. Die Tabelle 6.1 gibt einen kurzen Überblick zu den einzelnen Verfahren und deren Besonderheiten. Für die jeweiligen Details wird an dieser Stelle auf die entsprechenden Kapitel verwiesen. Im Anschluss an die Darstellung der einzelnen Strategien werden diese gegenübergestellt und miteinander verglichen.

6.2.1 Bidirektionale Referenzen

Der wohl am einfachsten zu realisierende, jedoch sehr kostenintensive Ansatz zur Aktualisierung von eingehenden Referenzen ist die Verwendung von bidirektionalen Referenzen. Jedes GAO speichert zusätzlich zu den ausgehenden Referenzen die invertierten Repräsentanten für alle eingehenden Referenzen. Das heißt es muss die SSR-Adresse jedes GAOs gespeichert werden, welches das aktuell betrachtete GAO mit einer ausgehenden Referenz referenziert. Zusätzlich ist es notwendig, die SSR-Adresse der zugehörigen Home Nodes zu speichern. Darüber hinaus können optional die Source Routen, die zu den Home Nodes führen, gespeichert werden. Dies entspricht den invertierten ausgehenden Referenzen der referenzierenden GAOs. Für

lokale Referenzen ist es nicht notwendig, diese bidirektional zu speichern, da lokale Objekte nicht migrieren können und folglich lokale Referenzen immer gültig sind. Somit wird aus dem gerichteten Referenzgraphen (siehe Kapitel 5.1) ein teilweise bidirektionaler Referenzgraph. Nachfolgende Abbildung zeigt den Referenzgraphen aus Abbildung 5.2 mit bidirektionalen globalen Referenzen.

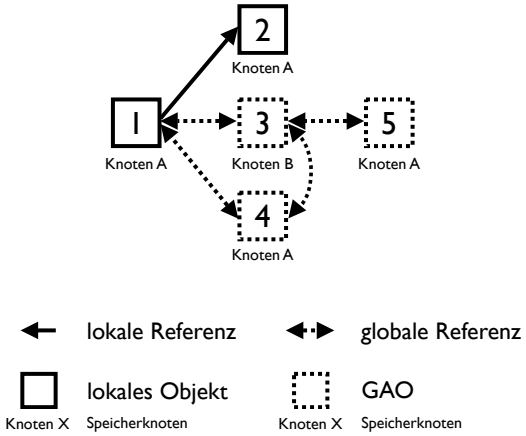


Abbildung 6.3: Referenzgraph mit bidirektionalen Referenzen

Im Falle einer Migration können nicht nur die ausgehenden, sondern auch die eingehenden Referenzen aktualisiert werden. Es wird dabei folgendermaßen vorgegangen: Zunächst wird eine *Migrationsnachricht*, die das zu migrierende GAO enthält, entlang der Migrationsroute zum neuen Home Node gesendet. Zusätzlich wird, ausgehend vom ursprünglichen Home Node, pro gespeicherter, invertierter, eingehender Referenz eine sogenannte *Aktualisierungsnachricht*, die die Migrationsroute beinhaltet, an die referenzierenden GAOs gesendet. Des weiteren muss für jede ausgehende Referenz des migrierten GAOs ebenfalls eine Aktualisierungsnachricht versendet werden, da jede ausgehende Referenz eine eingehende Referenz für ein anderes GAO darstellt. Für diese Referenzen existieren invertierte, eingehende Referenzen, die aktualisiert werden müssen. Nach Abschluss der Aktualisierungen wird auf dem ursprünglichen Home Node das migrierte GAO endgültig entfernt. Es sei an dieser Stelle angemerkt, dass die Migrations- und die Aktualisierungsnachricht jeweils die Information, welches GAO entlang welcher Migrationsroute von welchem Home Node zu welchem Home Node migriert wurde, enthalten.

Empfängt ein Knoten eine Aktualisierungsnachricht, so werden die betreffenden Referenzen aktualisiert. Dazu wird der neue Home Node des migrierten GAOs in die Abbildungstabelle eingetragen. Zusätzlich wird die Migrationsroute in den Route Cache eingefügt, wobei die implizite Konkatenation der bereits gespeicherten Source Routen mit der Migrationsroute stattfindet. Somit enthält der Route Cache unter der Voraussetzung, dass keine Source Routen verdrängt werden, eine Source Route zum alten Home Node und die Migrationsroute zum neuen Home Node.

Auf dem neuen Home Node wird nach dem Erhalt der Migrationsnachricht die Aktualisierung der ausgehenden und der invertierten, eingehenden Referenzen angestoßen. Die ausgehenden Referenzen werden dabei, wie bereits im Kapitel 6.1 beschrieben, aktualisiert. Für die invertierten, eingehenden Referenzen gilt, dass die zugehörigen Source Routen nicht mehr zwingend gültig sind, da deren Ursprung beim alten Home Node liegt. Die Konkatenation mit der invertierten Migrationsroute behebt diesen Missstand.

$$\text{incomingSourceRoute}_{new} = \text{(-migrationRoute)} + \text{incomingSourceRoute}_{old}$$

Nachdem alle Referenzen aktualisiert wurden, befindet sich das Gesamtsystem wieder in einem konsistenten Zustand.

Ausgehend vom konsistenten Zustand des bidirektionalen Referenzgraphen (siehe Abbildung 5.2, jedoch mit bidirektionalen globalen Referenzen) wird GAO_4 von Knoten A nach Knoten D migriert. Nach der Migration und der Anpassung der ausgehenden und invertierten eingehenden Referenzen des migrierten GAOs befindet sich der Referenzgraph in einem inkonsistenten Zustand (siehe Abbildung 6.2, jedoch mit bidirektionalen globalen Referenzen). Die gespeicherten Referenzen von LO_1 und GAO_3, die auf den ursprünglichen Home Node (Knoten A) des migrierten GAOs verweisen, müssen noch aktualisiert werden. Dies geschieht mittels der Aktualisierungsnachrichten. Der Empfang der Aktualisierungsnachrichten löst die Anpassung der betroffenen Referenzen von LO_1 auf Knoten A und GAO_3 auf Knoten B aus. Schließlich befindet sich der Referenzgraph wieder in einem konsistenten Zustand (siehe Abbildung 6.4).

Schreib- und Lesezugriffe auf migrierte GAOs erfordern keine zusätzlichen Aktionen, da ausgehende Referenzen immer gültig sind. Die Vorgehensweise für Schreib- und Lesezugriffe wurde bereits in Kapitel 5.2 beschrieben. Jedoch erfordert das Erzeugen und Löschen von Referenzen auf entfernt gespeicherte GAOs das Versenden von Aktualisierungsnachrichten. Dies wird durch die bidirektionalen Referenzen erforderlich. Einem GAO muss jede neu erzeugte, eingehende Referenz mitgeteilt werden, damit die invertierte, eingehende Referenz gespeichert werden kann. Genauso muss das Löschen einer ausgehenden Referenz dem referenzierten GAO mitgeteilt werden, so dass dieses die invertierte, eingehende Referenz entfernt.

Die Speicherung von invertierten eingehenden Referenzen, die zu den referenzierenden Objekten führen, birgt zwei Probleme. Beide basieren darauf, dass die Anzahl der eingehenden Referenzen für ein GAO prinzipiell sehr groß werden kann. Eine hohe Anzahl von ein- und ausgehenden Referenzen führt dazu, dass bei einer Migration sehr viele Aktualisierungsnachrichten gesendet werden müssen und das Netzwerk somit zusätzlich belastet und im Extremfall sogar überlastet wird. Des weiteren müssen die Referenzen gespeichert werden. Die zusätzliche Speicherung von allen invertierten eingehenden Referenzen kann unter Umständen so viel Speicherplatz erfordern, dass der betreffende Knoten nicht mehr „handlungsfähig" ist.

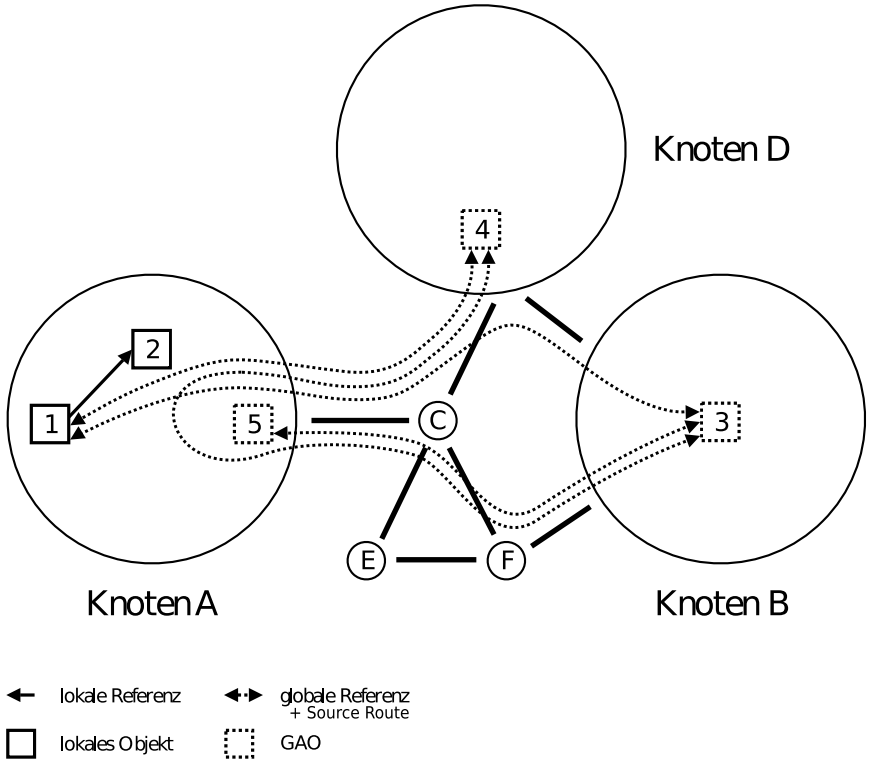


Abbildung 6.4: Konsistenter Zustand nach Migration von GAO_4

Die Anzahl der eingehenden Referenzen ist dabei von dem ausgeführten Programm abhängig. Als Beispiel sei an dieser Stelle ein Sensorknoten angeführt, dessen Sensordaten über ein GAO zugänglich sind. Interessieren sich viele Objekte der Anwendung für diese Sensordaten, so referenzieren diese das betreffende GAO. Überschreitet die Anzahl der eingehenden Referenzen eine gewisse Obergrenze, so ist der Speicher des betreffenden Knotens durch die invertierten, eingehenden Referenzen vollständig belegt.

6.2.2 Distributed Hash Table

Der im Folgenden erläuterte Ansatz zur Umgehung von inkonsistenten, eingehenden Referenzen basiert auf der Verwendung einer *Distributed Hash Table* (DHT) (siehe [3]). Eine DHT ist eine Datenstruktur, die in der Funktionsweise einer Hashtabelle entspricht, jedoch auf einem verteilten System aufsetzt. Im Allgemeinen ist es möglich, Tupel der Form (Schlüssel, Daten) innerhalb der verteilten Hashtabelle, das heißt auf den Knoten des Netzwerks, zu speichern. Mittels der Schlüssel können die Daten wieder gelesen werden.

Die Funktionalität einer DHT kann mit Hilfe des SSR Protokolls realisiert werden.

Zusätzlich zum Routing auf der Vermittlungsschicht werden die Daten auf allen SSR-Knoten des Netzwerks verteilt gespeichert. Dies ist der Funktionsweise von *Overlay DHTs*, wie zum Beispiel Chord (siehe [31]), ähnlich. Die Schlüssel der zu speichernden Daten werden aus dem SSR Adressraum gewählt. Ein Knoten speichert diejenigen Daten, deren Schlüssel zwischen der SSR-Adresse des jeweiligen Knotens und dessen virtuellen Nachfolgers liegen (aus [11], Kapitel 2).

Mit Bezug auf die Adressierung und Migration von GAOs werden die Zuordnungen von allen GAOs zu deren Home Nodes innerhalb der DHT gespeichert. Das bedeutet, dass eine Abbildungstabelle nicht lokal auf dem jeweiligen Knoten, sondern global in der DHT gespeichert wird. Die einzelnen Tabellen werden jedoch nicht separat betrachtet. Es ergibt sich somit für alle GAOs eine einzige, global verfügbare Abbildungstabelle. Die DHT wird mittels des SSR Protokolls realisiert. Als Daten der Hashtabelle werden die SSR-Adressen der Home Nodes gespeichert und als Schlüssel dienen die SSR-Adressen der zugehörigen GAOs.

Die Funktionsweise entspricht dem in [28] vorgeschlagenen *Orakel*. Jedoch unterscheidet sich das Orakel dahingehend, dass nur die Zuordnungen von statischen GAOs gespeichert werden. Statische GAOs entsprechen einerseits dem Programmcode und andererseits den statischen Teilen der Java Klassen, die nur einmal je Klasse existieren. Das Orakel kann zur Speicherung der globalen Abbildungstabelle verwendet werden, indem es um die Zuordnungen der nicht-statischen GAOs erweitert wird. Jedoch wird im Folgenden von einer eigenständigen DHT ausgegangen.

Ändert sich die Zuordnung eines GAOs zu dessen Home Node, so befindet sich die DHT in einem inkonsistentem Zustand. Jedes Erzeugen und Löschen, sowie jede Migration eines GAOs erfordert die Aktualisierung der DHT. Dazu wird der Versand einer *Aktualisierungsnachricht* auf dem Knoten, der das Erzeugen, Löschen oder Migrieren eines GAOs ausführt, initiiert. Als Zieladresse der Nachricht wird die SSR-Adresse des betreffenden GAOs gewählt und dessen neuer Home Node in der Nachricht vermerkt. Daraus folgt, dass die Nachricht zu demjenigen Knoten geroutet wird, der die Zuordnung innerhalb der DHT vorhält. Am Zielknoten angekommen, wird für ein erzeugtes GAO eine neue Zuordnung gespeichert. Beim Löschen eines GAOs muss der empfangende Knoten die Zuordnung löschen bzw. für eine Migration die Zuordnung aktualisieren.

Befindet sich die DHT in einem konsistenten Zustand, so kann von einem Thread, durch Verwendung der ausgehenden Referenzen eines beliebigen GAOs, auf ein referenziertes GAO zugegriffen werden. Bevor ein Zugriff stattfindet, muss zunächst mit Hilfe der DHT die Zuordnung des referenzierten GAOs zu dessen Home Node ermittelt werden. Dies geschieht durch eine *Anfragenachricht*, die an den Knoten gesendet wird, der die Zuordnung des zugegriffenen GAOs speichert. Nach dem Erhalt der Anfragenachricht wird diese Nachricht an den zugehörigen Home Node weitergesendet. Der SSR Routing Algorithmus routet diese Nachricht zum Home Node. Dieser sendet wiederum eine *Antwortnachricht* zurück an den Knoten, der den Zugriff initiiert hat.

Wie bereits im Kapitel 5.2 beschrieben, können zur Verbesserung zusätzlich zu den Abbildungstabellen die Source Routen gespeichert werden. Dadurch können die Lese- oder Schreibnachrichten, nach der Ermittlung der Source Route durch die DHT, direkt zum Home Node geroutet werden. Jedoch erfordert dies einen Mehraufwand bezüglich des verwendeten Speicherplatzes der DHT. Für jede gespeicherte Zuordnung muss somit zusätzlich die Source Route, die zum Home Node führt, gespeichert werden.

Jeder Zugriff auf ein GAO erfordert eine Abfrage der DHT. Dies erzeugt für eine Vielzahl von parallelen Zugriffen eine hohe Last auf dem gesamten Netzwerk. Paralİele Zugriffe ergeben sich durch Multithreading oder der Ausführung voneinander unabhängiger Anwendungen. Darüber hinaus erhöht der Zugriff auf die DHT die Latenzzeit für einen Zugriff auf ein GAO. Zur Vermeidung einer hohen Netzwerklast und zur Leistungssteigerung, wird das in Kapitel 5 erläuterte Verfahren verwendet. Das bedeutet, dass jeder Knoten die ausgehenden Referenzen lokal in einer Abbildungstabelle und zusätzlich die zugehörigen Source Routen im Route Cache speichert. Falls vorhanden, wird für einen Zugriff auf ein GAO zunächst die gespeicherte Referenz und die zugehörige Source Route verwendet. Wurde das GAO im Vorfeld nicht migriert, so endet der Zugriff erfolgreich. Nach Migrationen oder bei nicht vorhandenen Referenzen muss durch die DHT die Zuordnung des zugegriffenen GAOs zu dessen Home Node ermittelt werden. Anschließend kann der Zugriff und nach dessen Abschluss die Aktualisierung bzw. die Speicherung der ausgehenden Referenz erfolgen.

Ein Nachteil der DHT bezüglich des Migrationsproblems ist die globale Verteilung der Daten auf die Knoten des Netzwerks. Dies hat zur Folge, dass ein Knoten Daten speichern muss, die für ihn nicht von Bedeutung sind. Die Anzahl ist dabei durch die Gesamtanzahl aller vorhandenen GAOs im System beschränkt. Jedoch ist es nicht zu vermeiden, dass GAOs zu Knoten, die nur über sehr beschränkte Ressourcen verfügen, zugeordnet werden. Bei solchen Knoten wird bei einer entsprechenden Gesamtanzahl von GAOs die verfügbare Speichermenge überschritten, wodurch diese ihre eigentlichen Aufgaben nicht mehr korrekt ausführen können.

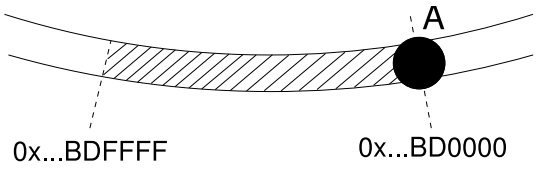


Abbildung 6.5: 16-Bit Adressbereich eines Knotens
Aus [28], Abbildung 1

Dieses Problem lässt sich umgehen, indem leistungsfähigere Knoten die Daten aus der DHT speichern. Um dies zu erreichen, müssen die GAOs, genauer gesagt deren SSR-Adressen, diesen Knoten zugeordnet werden. Dazu wird für jeden SSR-Knoten ein Adressbereich für die Zuordnung von GAOs reserviert. Der Adressbereich beginnt

bei dem jeweils betrachteten Knoten. Die Größe des Bereichs ist davon abhängig, wie viele GAOs maximal einem Knoten zugeordnet werden können. Werden zum Beispiel für diesen Adressbereich die letzten 16-Bit der SSR-Adressen verwendet, so ergibt sich daraus, dass die letzten 16-Bit der Knotenadressen den Wert „0“ haben. GAOs, die einem Knoten zugeordnet sind, unterscheiden sich von dessen Knotenadresse somit nur in den letzten 16-Bit. Abbildung 6.5 zeigt den für GAOs reservierten Adressbereich (schraffierte Fläche) beginnend bei Knoten A (siehe [28]).

Nachteil dieser Vorgehensweise ist die Tatsache, dass leistungsfähigere Knoten im Netzwerk vorhanden und dessen SSR-Adressen bekannt sein müssen. Dies erfordert einen zusätzlichen Mechanismus und erhöht somit wiederum die Komplexität. Des weiteren stellen diese Knoten einen „Flaschenhals” dar, da sie eine erhöhte Anzahl von Anfragen bearbeiten müssen.

6.2.3 Lokal konsistente Adressringe

Zur Adressierung von GAOs wird der im Kapitel 5 vorgeschlagene Mechanismus verwendet. Um die Adressierbarkeit von migrierten GAOs gewährleisten zu können, wird das Konzept von *Local Consistent Rings* (LCR) verwendet. Der LCR Mechanismus basiert auf der Erzeugung und Aufrechterhaltung eines virtuellen Adressrings für jedes GAO. Der Begriff der Lokalität bezieht sich auf diejenigen GAOs, die aus der Sicht des Referenzgraphen zur unmittelbaren Nachbarschaft des betreffenden GAOs gehören.

Im Folgenden wird ein zufällig ausgewähltes GAO, das als *Ringmaster* bezeichnet wird, betrachtet. Dem ausgewählten GAO wird ein virtueller Adressring zugewiesen. Dieser Ring besteht aus dem Ringmaster und allen GAOs, die direkte Referenzen auf den Ringmaster besitzen. Die Referenzen sind aus der Sicht des Ringmasters eingehende Referenzen. Es ist jedoch nicht nötig, die vom Ringmaster ausgehend referenzierten GAOs in den virtuellen Adressring mit aufzunehmen. Der Grund dafür ist, dass nur eingehende Referenzen ihre Gültigkeit verlieren können. Der Ringmaster speichert zur Aufrechterhaltung des zugehörigen virtuellen Adressrings zwei Referenzen, eine Referenz zum virtuellen Vorgänger und eine Referenz zum virtuellen Nachfolger. Die Struktur der virtuellen Adressringe ist dabei an den globalen virtuellen SSR Adressring (siehe Kapitel 4.1) angelehnt.

Jedes GAO, das den betrachteten Ringmaster referenziert, speichert zusätzlich zur Referenz, die auf den Ringmaster verweist, zwei weitere Referenzen. Die beiden zusätzlichen Referenzen verweisen auf die virtuellen Nachbarn innerhalb des virtuellen Adressrings des referenzierten Ringmasters. Die Speicherung erfolgt in Kombination mit dem zugehörigen Eintrag des referenzierten Ringmasters in der jeweiligen Abbildungstabelle. Ein Eintrag der Abbildungstabelle wird somit zu einem 4-Tupel von SSR-Adressen (referenziertes GAO, Home Node, virtueller Vorgänger, virtueller Nachfolger) erweitert. Optional werden die beiden Source Routen, die zu

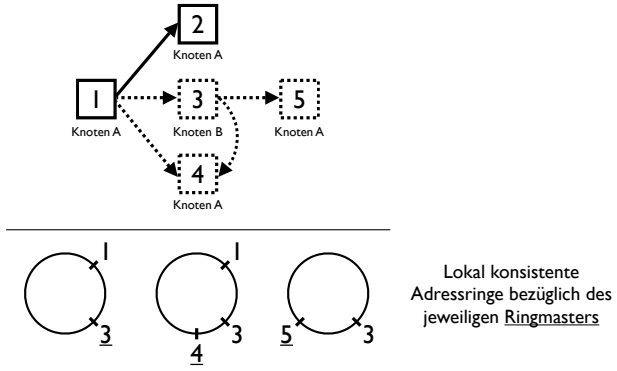


Abbildung 6.6: Referenzgraph mit lokalen Adressringen

den Home Nodes der virtuellen Vorgänger und Nachfolger führen, im Route Cache gespeichert. Daraus folgt, dass der Speicherbedarf für eine Referenz konstant um den Faktor zwei zunimmt. Der Speicherbedarf eines Knotens steigt linear mit der Zahl der ausgehenden Referenzen. Die Abbildung 6.6 zeigt den in Abbildung 5.2 dargestellten Referenzgraphen. Zusätzlich wurde zu jedem GAO der entsprechende virtuelle Adressring mit eingezeichnet.

Nachfolgend werden das dynamische Erzeugen und Löschen von Referenzen, sowie Migrationen von GAOs behandelt. Diese Operationen beeinflussen die Struktur der virtuellen Adressringe. Es ist jedoch von Bedeutung, dass jeder Adressring geschlossen gehalten wird. Geschlossen bedeutet in diesem Zusammenhang, dass es möglich ist, eine Nachricht ausgehend vom Ringmaster nur mittels der Vorgänger- bzw. der Nachfolgerreferenzen einmal um den gesamten Adressring zu senden. Solange der Adressring sich in einem konsistenten Zustand befindet, kann unter Garantie jeder Knoten des Adressrings erreicht werden. Diese Tatsache wird bei Zugriffen, die nach einer Migration eines GAOs stattfinden, ausgenutzt.

Erzeugt ein GAO, der *Initiator*, eine neue Referenz auf ein anderes GAO, den Ringmaster, so muss der Ringmaster den Initiator in seinen virtuellen Adressring integrieren. Zunächst sendet der Initiator eine *Integrationsnachricht* an den Ringmaster. Die Nachricht wird beim Erhalt vom Ringmaster innerhalb seines Adressrings weitergeleitet. Zur Weiterleitung wird der SSR Routing Algorithmus (siehe Kapitel 4.2) verwendet. Im schlechtesten Fall werden dabei fast alle zugehörigen GAOs des virtuellen Rings traversiert. Die Integrationsnachricht gelangt durch das Routing zu demjenigen GAO, dessen SSR-Adresse den geringsten virtuellen Abstand zur SSR-Adresse des zu integrierenden GAOs aufweist. Dieses GAO übernimmt dann die Rolle des *Integrators*.

Falls ein GAO für den nachfolgenden Adressbereich zuständig ist, so gelten für die SSR-Adressen des Integrators und Initiators die beiden folgenden Beziehungen:

1. $d_v(Integrator, Initiator) = min$
2. $addr_v(Integrator) < addr_v(Initiator)$

Hierbei bezeichnet $d_v(A, B)$ den virtuellen Abstand der beiden SSR-Adressen A und B und $addr_v(G)$ die virtuelle Adresse des GAOs G (vgl. Kapitel 4.1 und 4.2).

Nachdem der Integrator die Integrationsnachricht empfangen hat, sendet dieser eine Antwortnachricht zurück an den Initiator. Diese Nachricht enthält die Referenzen zu den neuen Nachbarn, die der Initiator in Kombination mit der zum Ringmaster gehörenden Referenz speichert. Die Vorgängerreferenz zeigt dabei auf den Integrator und die Nachfolgerreferenz auf den ehemaligen Nachfolger des Integrators. Anschließend informiert der Integrator mit Hilfe einer Aktualisierungsnachricht seinen virtuellen Nachfolger über dessen neuen Vorgänger, den Initiator. Schließlich aktualisiert der Integrator die Referenz seines virtuellen Nachfolgers mit der Referenz, die zum Initiator führt. Alle bisher ausgetauschten Nachrichten enthalten immer die Information über den zugehörigen Ringmaster. Ansonsten ist es nicht möglich zu unterscheiden, ob sich eine Nachricht auf die Referenz zu GAO X oder auf die Referenz zu GAO Y bezieht.

Das Löschen einer Referenz wird von demjenigen GAO initiiert, welches diese Referenz tatsächlich besitzt. Dieses GAO wird wiederum als Initiator bezeichnet. Wird eine Referenz auf den Ringmaster entfernt, so muss der zugehörige virtuelle Adressring des Ringmasters aktualisiert werden. Das bedeutet, dass der Initiator aus dem virtuellen Adressring des Ringmasters entfernt wird. Dazu informiert der Initiator, genauer gesagt der zugehörige Home Node, die zu der Ringmasterreferenz gespeicherten virtuellen Nachbarn, so dass diese ihren virtuellen Vorgänger bzw. den virtuellen Nachfolger aktualisieren. Anschließend entfernt der Initiator die Referenz aus seinem Speicher. Optional kann zusätzlich auch die im Route Cache gespeicherte Source Route gelöscht werden.

Migriert ein GAO, der Ringmaster, von einem Knoten zu einem anderen Knoten, so ist der zum Ringmaster zugehörige, lokale, virtuelle Adressring nicht mehr geschlossen. Dies hat den Grund, dass die beiden virtuellen Nachbarn nach der Migration keine gültigen Referenzen mehr auf den Ringmaster besitzen. Nach Abschluss der Migration ist es also nötig, dass der Ringmaster vom neuen Home Node ausgehend jeweils eine Nachricht an den virtuellen Vorgänger und Nachfolger sendet. Mittels dieser Nachricht erhalten beide virtuellen Nachbarn den neuen Home Node und somit befindet sich der lokale, virtuelle Adressring wieder in einem konsistenten Zustand. Zusätzlich zur Aktualisierung der Zuordnung des migrierten GAOs zu dessen Home Node können die zu den Vorgänger- und Nachfolgerreferenzen zugehörigen Source Routen durch Konkatenation mit der Migrationsroute in einen gültigen Zustand überführt werden. Für den virtuellen Vorgänger des migrierten GAOs gilt:

$$\text{successorSourceRoute}_{new} = \text{successorSourceRoute}_{old} + \text{migrationRoute}$$

Bezüglich des virtuellen Nachfolgers wird die Source Route folgendermaßen konkateniert:

$$\text{predecessorSourceRoute}_{new} = \text{predecessorSourceRoute}_{old} + \text{migrationRoute}$$

Darüber hinaus müssen auf dem neuen Home Node des migrierten GAOs die Source Routen, die zu den virtuellen Nachbarn führen, ebenfalls aktualisiert werden:

$$\text{predecessorSourceRoute}_{new} = (\text{-migrationRoute}) + \text{predecessorSourceRoute}_{old}$$

$$\text{successorSourceRoute}_{new} = (\text{-migrationRoute}) + \text{successorSourceRoute}_{old}$$

Die Konkatenation entspricht dabei dem im Kapitel 5.2 beschriebenen Vorgehen zur Aktualisierung der ausgehenden Referenzen und deren zugehörigen Source Routen. Diese Entsprechung ist dadurch begründet, dass die Vorgänger- und Nachfolgerreferenzen aus der Sicht des migrierten GAOs ausgehende Referenzen sind.

Schreibende und lesende Zugriffe auf GAOs werden entsprechend, wie in Kapitel 5.2 beschrieben, ausgeführt. Probleme bei Zugriffen treten erst dann auf, wenn GAOs migriert werden, da nach einer Migration alle eingehenden Referenzen mit Ausnahme der Vorgänger- und Nachfolgerreferenzen nicht mehr gültig sind. Greift ein Thread entlang einer nicht mehr gültigen Referenz auf ein migriertes GAO zu, so erhält es eine Nachricht vom ursprünglichen Home Node, dass das GAO migriert wurde. Daraufhin wird die Zugriffsnachricht entlang der virtuellen Vorgänger- bzw. Nachfolgerreferenz gesendet, die in Kombination mit der ursprünglichen Referenz gespeichert wurden. Das heißt es wird die Suche nach dem im Vorfeld erfolglos zugegriffen Ringmaster innerhalb des zugehörigen virtuellen Adressrings gestartet. Die Nachricht wird solange entlang des Adressrings weitergeleitet, bis der Ringmaster erreicht wurde. Am Ringmaster angekommen, antwortet dieser direkt dem zugreifenden GAO. Solange der Adressring geschlossen ist, wird der Ringmaster garantiert gefunden und der Zugriff erfolgreich abgeschlossen.

Die Suche nach dem Ringmaster nach einer Migration bzw. dem Integrator beim Erzeugen von Referenzen erfordert im schlechtesten Fall die Traversierung von $n-2$ GAOs. Hierbei beschreibt n die Anzahl der GAOs, die in den zugehörigen Adressring integriert sind. Daraus folgt, dass der damit verbundene Zeitbedarf abhängig von der Anzahl der im Adressring integrierten GAOs ist. Die Anzahl der GAOs eines Adressrings kann bis zur Gesamtzahl aller GAOs des ambienten Systems anwachsen. Um sehr hohe Latenzzeiten, die sich zwangsläufig bei der Suche ergeben, zu vermeiden, kann jedes GAO zusätzliche Abkürzungsreferenzen auf andere GAOs desselben Adressrings speichern. Da die Abkürzungen an den jeweiligen Adressring gebunden sind, muss für jede Referenz die auf einem Knoten existiert, jeweils eine eigene Menge von Abkürzungen gespeichert werden. Dies stellt einen erheblichen Speicherbedarf dar. An dieser Stelle kann mit Hilfe einer Begrenzung der Anzahl von Abkürzungen pro Referenz der Speicheraufwand limitiert werden. Jedoch geht damit eine Erhöhung der Latenzzeit für die Suche des Ringmasters und des Integrators einher.

6.2.4 Weiterleitungen

Die bisher vorgestellten Verfahren beruhten darauf, dass aus Gründen der Konsistenz während bzw. nach einer Migration zusätzliche Nachrichten versendet werden müssen. Dabei hängt die Anzahl dieser Nachrichten vom gewählten Mechanismus ab. Im Folgenden wird ein Verfahren präsentiert, das nach einer Migration ohne weiteren Nachrichtenversand auskommt. Das Verfahren verwendet, ebenso wie die vorhergehenden Ansätze, das im Kapitel 5 erläuterte Adressierungsverfahren.

Die Migration eines GAOs basiert auf der Erstellung einer exakten Kopie des zu migrierenden GAOs und dessen Übertragung an den neuen Home Node. Nicht mehr referenzierte Kopien werden während der Laufzeit der ACVM durch die durchgeführte Garbage Collection (siehe Kapitel 3) gelöscht. Das Funktionsprinzip gibt dem Verfahren den Namen *GAO Cloning And Garbage Collecting* (GCAGC). Im Folgenden wird die übertragene und auf dem neuen Home Node gespeicherte Kopie des GAOs als *effective GAO* (efGAO) bezeichnet. Darüber hinaus wird das GAO, das nicht übertragen wurde und auf dem ursprünglichen Home Node zurück geblieben ist, als *cloned GAO* (clGAO) bezeichnet. Daraus folgt, dass durch Migration eines efGAOs dieses zu einem clGAO wird.

Eine Anforderung für GCAGC ist, dass GAOs nur gelesen werden können. Um dennoch schreibende Zugriffe zu ermöglichen, werden GAOs mit Versionen versehen. Dazu erhält jedes GAO einen *Versionszähler* oder einen vergleichbaren Mechanismus, wie zum Beispiel einen Zeitstempel. Wird ein GAO durch schreibenden Zugriff verändert, so wird zunächst eine exakte Kopie des GAOs erstellt. Danach wird diese Kopie entsprechend dem schreibenden Zugriff geändert, der Versionszähler erhöht und das neue GAO lokal gespeichert. Im weiteren Sinn entspricht ein schreibender Zugriff dem Ablauf einer Migration, der im Folgenden erläutert wird.

Wie bereits beschrieben, wird während der Migration eines GAOs eine exakte Kopie erzeugt. Diese Kopie enthält alle ausgehenden Referenzen. Die ausgehenden Referenzen bestehen dabei aus den SSR-Adressen der referenzierten GAOs. Zusätzlich werden die zugehörigen Home Nodes und die Source Routen, die zu den Home Nodes führen, mit in der Kopie gespeichert. Anschließend wird die erzeugte Kopie mittels einer Migrationsnachricht entlang der Migrationsroute an den neuen Home Node übertragen. Nach erfolgreichem Abschluss der Übertragung wird das ehemalige efGAO nun als clGAO bezeichnet und entsprechend markiert. Zusätzlich speichert das clGAO eine Referenz, die auf das efGAO verweist. Diese Referenz wird als *Head Referenz* bezeichnet und setzt sich aus den Informationen, die für die Migration notwendig sind, zusammen. Das bedeutet im Detail, dass die Head Referenz aus der SSR-Adresse des efGAOs, aus dessen zugehörigen neuen Home Node und der Migrationsroute besteht. Des weiteren können der Inhalt des clGAOs und alle zugehörigen ausgehenden Referenzen, bis auf die Head Referenz, verworfen werden. Dies hat zur Folge, dass das clGAO lediglich eine Weiterleitung hin zum efGAO darstellt.

Empfängt ein Knoten eine Migrationsnachricht, so wird das darin übertragene efGAO auf diesem Knoten gespeichert. Sollte durch vorhergehende Migrationen bereits ein clGAO auf dem Knoten vorhanden sein, so wird dieses mit dem efGAO überschrieben bzw. aktualisiert, falls clGAOs Weiterleitungen darstellen. Ebenso werden alle ausgehenden Referenzen des efGAO durch Konkatenation mit der Migrationsroute wieder in einen konsistenten Zustand überführt (siehe Kapitel 6.1). An diesem Punkt ist die Migration beendet.

Wiederholte Migrationen desselben efGAOs führen auf Grund der Head Referenzen zu einer einfach verketteten, linearen Liste. Innerhalb dieser Liste sind alle clGAOs, die auf verschiedenen Knoten vorgehalten werden, referenziert. Falls auf einem Knoten durch einen Schreibzugriff mehrere Versionen eines GAOs vorhanden sind, so erscheint nur das GAO mit der neuesten Version in der Listenstruktur. Mittels dieser Listenstruktur ist es somit möglich, ausgehend von einem clGAO das zugehörige efGAO zu finden, indem immer die Head Referenz traversiert wird. Abbildung 6.7 zeigt die zweifache Migration von GAO_3 des Referenzgraphen aus Abbildung 5.2 und die sich daraus ergebende Listenstruktur. GAO_3 wurde von Knoten A entlang der Migrationsroute $A - C - F - B$ zu Knoten B und anschließend zu Knoten D migriert. Es ist zu erkennen, dass die ausgehende Referenz von LO_1 immer noch auf das ursprüngliche clGAO verweist.

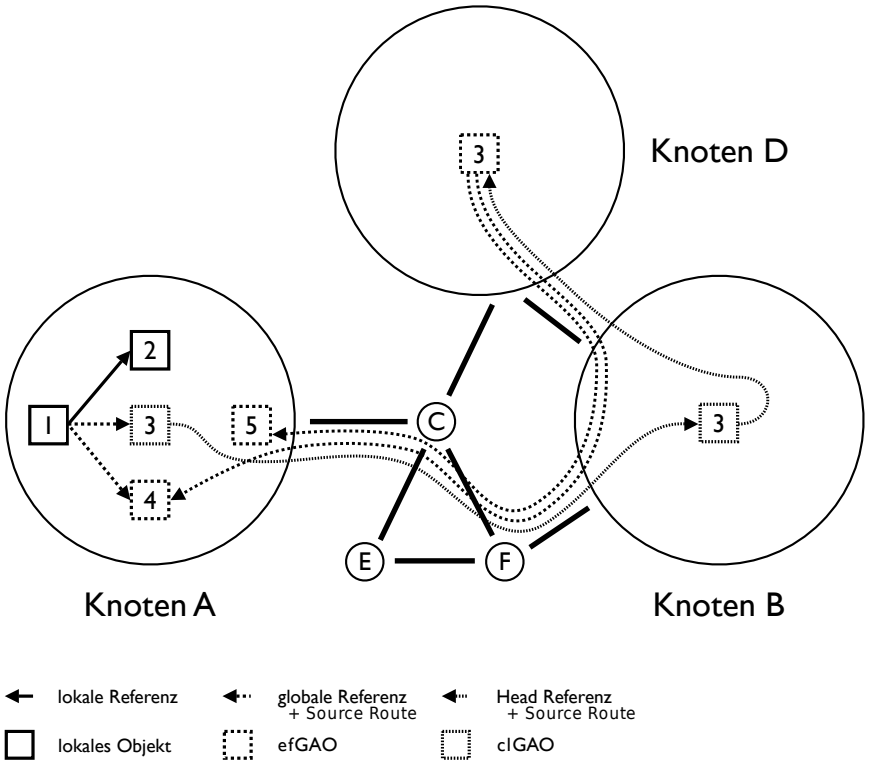


Abbildung 6.7: Aufbau der Listenstruktur nach zwei Migrationen

Zur Aufrechterhaltung der Listenstruktur ist es nicht erlaubt, clGAOs zu migrieren. Unter der Annahme einer konsistenten Listenstruktur würde die Migration eines clGAOs die bereits bestehende Head Referenz überschreiben. Damit wäre das efGAO nicht mehr zu erreichen.

Im Folgenden wird beschrieben, wie lesende und schreibende Zugriffe auf bereits migrierte GAOs erfolgen. Greift ein GAO auf ein anderes GAO zu, so wird eine Zugriffsnachricht entlang der entsprechenden Referenz gesendet. Der Knoten, der die Nachricht empfängt, überprüft zunächst, ob das angeforderte GAO lokal gespeichert ist. Dieses entspricht entweder einem clGAO oder einem efGAO. Im Falle eines clGAOs wird die empfangene Nachricht entlang der Head Referenz weitergeleitet. Die Weiterleitung der Zugriffsnachricht endet bei demjenigen Knoten, der das efGAO vorhält. Handelt es sich um einem schreibenden Zugriff, so muss eine neue Version des efGAO erzeugt werden. Im Falle eines lesenden Zugriffs wird das efGAO durch eine Antwortnachricht entlang der invertierten Referenz zurückgesendet. Der Empfänger der Antwortnachricht ist der Thread, der den Zugriff initiiert hat, bzw. dessen Home Node.

Durch die soeben beschriebene Vorgehensweise finden Lese- und Schreibzugriffe immer auf das efGAO statt. Dies hat zur Folge, dass die Versionen eines GAOs innerhalb der Listenstruktur aufsteigend sortiert sind. Somit repräsentiert das efGAO die aktuellste Version des betreffenden GAOs. Es sei an dieser Stelle angemerkt, dass die Sortierung der Listenstruktur keinen Einfluss auf die Synchronisierung von Lese- und Schreibzugriffen hat. Die Synchronisierung ist auf Grund von Nebenläufigkeiten bei der Ausführung von Programmen durch die ACVM notwendig. Wie bereits im Kapitel 3 angeführt, kann dafür der STM Mechanismus verwendet werden.

Nach einer Migration eines efGAOs behalten die eingehenden Referenzen weiterhin ihre Gültigkeit. Sie verweisen nicht mehr auf das efGAO, sondern auf das clGAO. Jedoch besteht der Nachteil, dass bei einem Zugriff auf ein clGAO die Zugriffsnachricht entsprechend der Head Referenz weitergeleitet werden muss. Zur Vermeidung von ständigen Weiterleitungen von Nachrichten müssen die eingehenden Referenzen, die auf clGAOs verweisen, aktualisiert werden. Das Ziel der Aktualisierung ist es, dass schließlich alle eingehenden Referenzen auf das efGAO zeigen. Diesbezüglich gibt es einerseits die Möglichkeit durch lesende Zugriffe und andererseits durch die Garbage Collection die Referenzen zu aktualisieren. Beide Verfahren sind voneinander unabhängig und können nebeneinander durchgeführt werden. Im Folgenden wird auf beide Verfahren eingegangen.

Der lesende Zugriff auf ein GAO geschieht entlang einer Referenz. Der Ablauf des Zugriffs wurde bereits weiter oben beschrieben und endet mit dem Erhalt einer Antwortnachricht. Beim Empfang der Antwortnachricht muss überprüft werden, ob die lokal gespeicherte Referenz, die für den Zugriff verwendet wurde, auf das clGAO oder das efGAO verweist. Zur Überprüfung wird derjenige Home Node, der in Verbindung mit der verwendeten Referenz gespeicherte ist, mit dem Quellknoten der Antwortnachricht verglichen. Stimmen beide nicht überein, so wurde die Zugriffsnachricht

an das efGAO weitergeleitet. In diesem Fall verweist die gespeicherte Referenz auf das clGAO und muss mit Hilfe der Informationen aus der Antwortnachricht aktualisiert werden. Das bedeutet, dass die Zuordnung von GAO zum Home Node in der Abbildungstabelle mit dem Quellknoten der Antwortnachricht aktualisiert wird. Zusätzlich wird die invertierte Source Route der Antwortnachricht, die direkt zum Home Node des efGAOs führt, in den Route Cache eingetragen.

Die durch Migration entstandenen clGAOs müssen aus dem System entfernt werden. Ansonsten steigt der Speicherbedarf für die clGAOs immer weiter an und führt schließlich dazu, dass das verteilte System nicht mehr in der Lage ist, Anwendungen auszuführen. Ein clGAO darf erst entfernt werden, wenn dieses keine eingehenden Referenzen mehr besitzt. Ist dies der Fall, so werden clGAOs automatisch durch die Garbage Collection, die Teil der ACVM ist, entfernt. Darüber hinaus führt die Garbage Collection eine Aktualisierung derjenigen Referenzen durch, die auf clGAOs verweisen.

Zur weiteren Betrachtung wird für die Garbage Collection ein verteilter *Mark and Sweep* Algorithmus (siehe [25]) angenommen. Dessen Details sind an dieser Stelle, bis auf die Tatsache, dass bei jedem neuen Durchlauf der Garbage Collection die von allen GAOs gespeicherten Referenzen traversiert werden, nicht von Bedeutung. Bei der Ausführung der Garbage Collection werden auch diejenigen Referenzen traversiert, die zu clGAOs führen. Die Traversierung wird mit Bezug auf GCAGC als eine Art lesender Zugriff realisiert. Der Unterschied zum oben beschriebenen lesendem Zugriff liegt darin, dass bei der Traversierung einer Referenz, die auf ein efGAO verweist, keine Antwortnachricht versendet wird. Wird hingegen auf ein clGAO zugegriffen, so wird die *Traversierungsnachricht* an das efGAO entlang der Head Referenz weitergeleitet. Der Empfang einer weitergeleiteten Traversierungsnachricht veranlasst den Versand einer Antwortnachricht an den Ursprung der Traversierungsnachricht. Beim Empfang der Antwortnachricht, wird automatisch die Referenz vom clGAO auf das efGAO gesetzt. Nachdem die Garbage Collection für das gesamte System ausgeführt wurde und in der Zwischenzeit keine weiteren Migrationen stattgefunden haben, verweisen alle Referenzen auf efGAOs. Im nächsten Durchlauf der Garbage Collection werden alle clGAOs, die nicht mehr referenziert werden, zum Entfernen markiert und schließlich gelöscht. Wird ein GAO gelöscht, so werden auch alle gespeicherten Referenzen des betreffenden GAOs mit entfernt.

Die Ausführung der Garbage Collection ist für die ACVM notwendig, damit Speicherressourcen durch nicht mehr benötigte Objekte freigegeben werden. Durch die Garbage Collection werden, wie im vorhergehenden Absatz beschrieben, zusätzlich die nicht mehr referenzierten clGAOs gelöscht. Die durchschnittliche Anzahl der sich im Netzwerk befindlichen clGAOs ist dabei abhängig vom Verhältnis der Häufigkeit der Migrationen zur Anzahl der durchgeführten Garbage Collections. Das bedeutet, dass bei sehr häufigen Migrationen, die Anzahl der clGAOs steigt. Dementsprechend sollte sich die Rate der Garbage Collections an die Häufigkeit der Migrationen anpassen. Dabei muss jedoch gewährleistet werden, dass das verteilte System immer

noch operativ bleibt. Es besteht die Gefahr, dass eine Vielzahl von ausgetauschten Nachrichten, hervorgerufen durch Migrationen und Garbage Collections, das darunterliegende Netzwerk überlasten.

6.3 Gegenüberstellung

In Kapitel 6.2 wurden vier Migrationsverfahren zur Lösung bzw. Umgehung von inkonsistenten, eingehenden Referenzen erläutert. Alle vier Verfahren basieren auf der Verwendung der in Kapitel 5 erläuterten Adressierung mittels Referenzen, Zuordnung der referenzierten GAOs zu Home Nodes und der Vermeidung des SSR Routing Algorithmus durch die zusätzliche Speicherung von Source Routen zu den Home Nodes der GAOs. Durch die verschiedenen Vorgehensweisen bei der Migration ergeben sich teils gravierende Unterschiede für den jeweils zusätzlich notwendigen Aufwand. Die Migrationsverfahren können anhand der beiden Kriterien

- Speicherbedarf und
- Kommunikationsaufwand

analysiert und miteinander verglichen werden. Die Analyse der beiden Kriterien bezieht sich ausschließlich auf den für das jeweilige Migrationsverfahren erforderlichen Mehraufwand. Das bedeutet, dass der Aufwand für das Adressierungsverfahren nicht berücksichtigt wird, da dieser für alle Verfahren gleich und nicht zu vermeiden ist.

Bei der Gegenüberstellung wird jedoch betrachtet, dass das Verfahren in Kombination mit der ACVM (siehe Kapitel 3) verwendet werden soll. Dies hat zur Folge, dass auf Speichereffizienz geachtet werden muss. Der Grund für die Speichereffizienz sind die beschränkten Speicherressourcen von Mikrocontrollern, die als Zielplattform für die ACVM verwendet werden. Darüber hinaus sollten die Latenzzeiten für Zugriffe auf GAOs, die nicht lokal vorgehalten werden, minimiert werden, um eine schnelle Ausführung von Programmen zu erreichen.

6.3.1 Speicherbedarf

Im Folgenden wird der zusätzlich benötigte Speicher pro Knoten für das jeweilige Migrationsverfahren analysiert. Der Mehraufwand an belegtem Speicher ergibt sich bei allen Verfahren durch die zusätzliche Speicherung von Referenzen.

Wie im Kapitel 5 beschrieben, werden pro Referenz ein Eintrag in der Abbildungstabelle und dem Route Cache gespeichert. Verweisen zwei Referenzen auf dasselbe GAO, so ergibt sich hierdurch kein Mehraufwand, da die Einträge gleich sind. Im weiteren Verlauf wird diese Tatsache jedoch nicht beachtet und somit vom schlechtesten Fall ausgegangen. Das bedeutet, dass auf einem Knoten keine zwei Referenzen

vorgehalten werden, die auf dasselbe GAO verweisen. Mit dieser Betrachtung ergibt sich die obere Schranke für den zusätzlichen Speicherbedarf pro Knoten.

Bidirektionale Referenzen

Um bidirektionale Referenzen zu erhalten, werden zusätzlich zu den ausgehenden Referenzen eines GAOs die invertierten, eingehenden Referenzen gespeichert (siehe Kapitel 6.2.1). Wird das gesamte System und die darin existierenden Referenzen betrachtet, steigt der Speicheraufwand nur konstant um den Faktor zwei. Jedoch kann, wie bereits im Kapitel 6.2.1 erläutert, die Anzahl eingehender Referenzen pro GAO und folglich pro Knoten sehr groß werden. Für ein GAO ist die Anzahl der eingehenden Referenzen durch die Gesamtanzahl von GAOs beschränkt. Das bedeutet, dass alle GAOs auf ein einziges GAO verweisen. Da, wie oben beschrieben, gleiche Referenzen keinen Mehraufwand darstellen, ist die Anzahl eingehender Referenzen pro Knoten ebenso durch die Gesamtanzahl von GAOs beschränkt.

Distributed Hash Table

Bei Verwendung einer *Distributed Hash Table* (DHT) (siehe Kapitel 6.2.2) wird pro GAO eine Referenz, genauer gesagt die Zuordnung zu dessen Home Node, in der DHT gespeichert. Für die „Konstruktion" der DHT wird angenommen, dass die SSR-Adressen der GAOs und der Knoten mittels einer geeignet gewählten Hash Funktion gleichverteilt auf den virtuellen Adressring abgebildet werden (siehe [28], Kapitel 4). Daraus folgt, dass pro Knoten r Referenzen gespeichert werden müssen. Für r gilt die Beziehung:

$$r \approx \frac{\text{Anzahl GAOs}}{\text{Anzahl Knoten}}$$

Auf Grund limitierter Speicherressourcen können pro Knoten nur eine maximale Anzahl x von GAOs vorgehalten werden. x ist hierbei von der Größe der GAOs abhängig und stellt einen durchschnittlichen Wert dar. Wird angenommen, dass alle Knoten über die gleichen Speicherressourcen verfügen und x somit für alle Knoten gleich ist, ergibt sich aus der obigen Beziehung:

$$r \approx \frac{\text{Anzahl GAOs}}{\text{Anzahl Knoten}} \leq \frac{x \cdot (\text{Anzahl Knoten})}{\text{Anzahl Knoten}} = x$$

Somit stellt x die obere Grenze der pro Knoten, innerhalb der DHT, gespeicherten Referenzen dar.

Lokal konsistente Adressringe

Im Gegensatz zum Migrationsverfahren der bidirektionalen Referenzen werden für lokal konsistente Adressringe zusätzliche Referenzen nicht für eingehende, sondern für ausgehende Referenzen gespeichert. Das bedeutet im Detail, dass ein Knoten für jedes gespeicherte GAO zwei Referenzen auf das Vorgänger- und das Nachfolger-GAO vorhält. Darüber hinaus müssen, zusätzlich zu jeder ausgehenden Referenz, die zugehörigen virtuellen Nachbarn und Abkürzungen innerhalb des virtuellen Adressrings gespeichert werden (siehe Kapitel 6.2.3).

Für die maximale Anzahl r zusätzlich zu speichernder Referenzen eines Knotens ergibt sich folglich:

$$r \leq 2 \cdot x + 2 \cdot y + y \cdot z$$

x bezeichnet hierbei, wie im vorhergehenden Abschnitt der DHT, die maximale Anzahl gespeicherter GAOs und y die maximale Anzahl der ausgehenden Referenzen pro Knoten. x und y sind durch die lokalen Speicherressourcen eines Knotens beschränkt. Darüber hinaus gibt z die Anzahl der pro Referenz gespeicherten Abkürzungen an und ist eine Konstante.

Weiterleitungen

Das Migrationsverfahren basierend auf Weiterleitungen (siehe Kapitel 6.2.4) benötigt die Erweiterung jedes GAOs um den Migrationsstatus und die Head Referenz. Mit Hilfe des Migrationsstatus kann zwischen *cloned GAOs* (clGAOs) und *effective GAOs* (efGAOs) unterschieden werden. Die Realisierung benötigt ein Bit und ist im Vergleich zur Größe eines GAOs zu vernachlässigen.

Solange keine Migrationen stattfinden, speichert die Head Referenz keinen Verweis auf das nächste Element in der Listenstruktur. Somit existieren für die Head Referenz keine Einträge in der Abbildungstabelle und dem Route Cache. Folglich kann die Existenz der Head Referenz vernachlässigt werden. Finden hingegen Migrationen statt, so wird der Inhalt der GAOs verworfen und die Head Referenz verweist auf das nachfolgende efGAO in der Listenstruktur. Die Größe eines clGAOs wird durch das Löschen des Inhalts auf die Größe einer Referenz reduziert. Bei Betrachtung des gesamten Systems ergibt sich im Durchschnitt für r, die Anzahl der zusätzlich gespeicherten Head Referenzen pro Knoten, die folgende Beziehung:

$$r = \frac{\text{Anzahl clGAOs}}{\text{Anzahl Knoten}}$$

Im schlechtesten Fall muss ein Knoten jedoch alle clGAOs speichern.

Es ist zu erkennen, dass n ohne weitere Bedingungen unbeschränkt wäre. Auf Grund der durchgeführten Garbage Collection werden alle clGAOs garantiert aus dem System entfernt und somit alle Head Referenzen gelöscht. Folglich ist n einerseits von der Migrationsrate und andererseits von der Rate der Garbage Collections abhängig.

Die Tabelle 6.2 fasst die oben erläuterten Schranken für die verschiedenen Migrationsverfahren zusammen. Die Schranken beziehen sich auf den im schlechtesten Fall benötigen Speicherbedarf pro Knoten, der über das in Kapitel 5 erläuterte Adressierungsverfahren hinausgeht.

6.3.2 Nachrichtenbedarf

Nach der Darstellung des zusätzlich notwendigen Speicherbedarfs für jedes Migrationsverfahrens wird an dieser Stelle auf den erforderlichen Kommunikationsaufwand eingegangen. Nachfolgende Tabellen 6.3 bis 6.7 führen die Anzahl der ausgetauschten Nachrichten bezüglich der Operationen

- Lesender und schreibender Zugriff auf ein GAO,
- Erzeugen und Löschen eines GAOs
- Erzeugen einer Referenz auf ein GAO,
- Löschen einer Referenz auf ein GAO und
- Migration eines GAOs

auf. Die Kosten, die für eine Garbage Collection anfallen, sind vom verwendeten Algorithmus abhängig und werden an dieser Stelle nicht beachtet. Jedoch gilt, dass beim Löschen eines GAOs, hervorgerufen durch die Garbage Collection, sich die Kosten für das Löschen des GAOs (siehe unten) zu den Gesamtkosten des Garbage Collection Algorithmus addieren.

Alle vier Migrationsverfahren basieren auf demselben Adressierungsverfahren (siehe Kapitel 5). Die Initialisierung erfordert für alle Verfahren keine Kommunikationskosten. Erst das Erzeugen und Löschen von GAOs und Referenzen benötigt einen zusätzlichen Aufwand. Darüber hinaus ergeben sich bei Zugriffen auf migrierte GAOs Unterschiede in der Anzahl der ausgetauschten Nachrichten. Die Anzahl ist dabei vom verwendeten Migrationsverfahren abhängig. Im Folgenden wird deswegen für Lese- und Schreibzugriffe angenommen, dass das zugegriffene GAO im Vorfeld migriert wurde. Die Anzahl der bereits stattgefundenen Migrationen ist beliebig.

Die Operationen zum Erzeugen und Löschen von GAOs werden nur lokal auf dem betreffenden Knoten durchgeführt. Es bedarf dafür keines Nachrichtenaustausches. Ruft das Erzeugen oder Löschen eines GAOs jedoch das Erzeugen oder Löschen von Referenzen hervor, so ergibt sich für die Referenzen ein zusätzlicher Kommunikationsaufwand. Die beiden Tabellen 6.5 und 6.6 geben Auskunft über die jeweils anfallenden Kommunikationskosten pro Referenz. Letztlich ist der Mehraufwand für das Erzeugen oder Löschen von GAOs durch die Anzahl der Referenzen beschränkt. Eine Ausnahme stellt die DHT dar. Es wird beim Erzeugen und Löschen eines GAOs dessen Zuordnung in der DHT gespeichert bzw. gelöscht (siehe Tabelle 6.4). Somit

erfordert das Erzeugen und Löschen von Referenzen auf ein GAO keinen weiteren Kommunikationsaufwand, da dessen Zuordnung bereits in der DHT gespeichert ist.

Bei Betrachtung der Tabelle 6.3 fällt auf, dass die Verwendung der bidirektionalen Referenzen bei einem Zugriff auf ein migriertes GAO keine zusätzlichen Kosten verursacht. Alle anderen Migrationsverfahren benötigen hingegen einen durch R, G bzw. L bestimmten Overhead. R zählt die Anzahl der virtuellen Hops, die bei der Abfrage der DHT und der Weiterleitung der Zugriffsnachricht an den Home Node des zugegriffenen GAOs, benötigt werden. R ist somit von der Performanz des SSR Routing Algorithmus abhängig. Im Gegensatz dazu zählt G die Anzahl der virtuellen Hops, die benötigt werden, um das migrierte GAO innerhalb des zugehörigen virtuellen Adressrings zu finden. Im schlechtesten Fall muss der komplette Adressring traversiert werden. Jedes GAO, das in diesen Adressring integriert ist, referenziert das migrierte GAO. Daraus folgt, dass die Größe des virtuellen Adressrings und somit die Anzahl der virtuellen Hops von der Anzahl der eingehenden Referenzen abhängig ist. L hingegen gibt die Länge der Listenstruktur des zugegriffenen GAOs und die damit einhergehende Anzahl von Weiterleitungen entlang dieser Listenstruktur an. Die Länge ergibt sich aus der Anzahl der Migrationen des zugehörigen GAOs und ist nicht von der Größe des Netzwerks und der Anzahl der eingehenden Referenzen abhängig.

Auf Grund der unterschiedlichen Abhängigkeiten von R, G und L ist es schwierig eine Aussage zu treffen, welche Relationen zwischen den einzelnen Faktoren bestehen. L kann jedoch beeinflusst werden. Einerseits kann die Häufigkeit der Garbage Collections erhöht und andererseits die Anzahl der Migrationen reduziert werden. Anhand dieser beiden Parameter ist es möglich, während der Laufzeit eines Programms die Leistung des ambienten Systems für lesende und schreibende Zugriffe aktiv zu beeinflussen. Für die beiden Verfahren Distributed Hash Table und lokal konsistente Adressringe gibt es keine derartigen Möglichkeiten. Sie sind immer mit dem Overhead von R bzw. G belegt.

Die Tabellen 6.4 bis 6.7 geben einen Überblick über die Anzahl der Nachrichten, die versendet werden müssen, um das verwendete Migrationsverfahren konsistent zu halten. Dabei beziehen sich die Tabellen auf die Operationen Erzeugen und Löschen von Referenzen und GAOs, sowie Migrationen von GAOs. Während der Ausführung eines Java Programms werden in Abhängigkeit der Programmstruktur eine Vielzahl von Referenzen erzeugt (= gespeichert), gelöscht und so verändert, dass diese auf andere GAOs verweisen. Die Veränderungen der Referenzen werden auf das Löschen der ursprünglichen Referenz und das Erzeugen einer neuen Referenz abgebildet. Folglich ist es wichtig, dass die beiden Referenz-Operationen effizient ausführbar sind.

Das Verfahren basierend auf Weiterleitungen (GCAGC) ist bezüglich dieser Operationen und der Migration von GAOs optimal. Es müssen keine zusätzlichen Nachrichten an andere Knoten versendet werden. Bidirektionale Referenzen benötigen

für die Referenz-Operationen einen konstanten Mehraufwand. Migrationen hingegen sind von der Anzahl der ein- und ausgehenden Referenzen abhängig. Bei der Verwendung von lokal konsistenten Adressringen (LCR) bedarf die Speicherung von Referenzen eine nicht konstante Anzahl ausgetauschter Nachrichten. Der Overhead für das Löschen von Referenzen und Migrationen von GAOs ist konstant. Für die DHT sind zwar die Referenz-Operationen mit keinem zusätzlichen Kommunikationsaufwand verbunden, jedoch bedarf das Erzeugen, Löschen und Migrieren von GAOs immer R Weiterleitungen der gesendeten Nachrichten.

Werden alle Operationen mit Bezug auf den Kommunikationsaufwand in Betracht gezogen, so erscheinen die Verfahren der bidirektionalen Referenzen und der Weiterleitungen als die aussichtsreichsten Kandidaten für eine Realisation als Migrationsmechanismus. Wird jedoch zusätzlich die Speichereffizienz mit berücksichtigt, so fällt die Wahl auf GCAGC, da für bidirektionale Referenzen der tatsächliche Speicherbedarf von der Gesamtanzahl der GAOs abhängig ist. Im Gegensatz dazu ist das DHT Verfahren am speichereffizientesten, benötigt aber im Vergleich zu GCAGC beim Erzeugen und Löschen von GAOs einen zusätzlichen Nachrichtenaufwand.

Auf Grund der Eignung von GCAGC als Migrationsverfahren wurde dieses ausgewählt und implementiert. Das nachfolgende Kapitel 7 gibt einen Überblick über die Implementierung. Auf der Implementierung aufbauend wurde das Verhalten von GCAGC evaluiert. Kapitel 8 beschreibt die Rahmenbedingungen und die Ergebnisse der Evaluation.

Migrations-verfahren	Abkürzung	Kurzbeschreibung
Bidirektionale Referenzen	BiRefs	• Speicherung von Referenzen, die auf referenzierende GAOs verweisen ⇒ invertierte, eingehende Referenzen • Aktualisierung dieser Referenzen nach Migrationen
Distributed Hash Table	DHT	• Verwendung einer globalen Abbildungstabelle • Aktualisierung der DHT nach jeder Änderung von Zuordnungen der GAOs zu dessen Home Nodes • Keine Speicherung von Source Routen in der DHT
Lokal konsistente Adressringe	LCR	• Jedes GAO besitzt einen virtuellen Adressring, der alle referenzierenden GAOs beinhaltet • Suche nach dem migrierten GAO innerhalb des zu diesem GAO zugehörigen Adressrings • Integration und Entfernen eines GAOs in bzw. aus einem Adressring bei Änderung der ausgehenden Referenz
Weiterleitungen	GCAGC	• Migrierte GAOs hinterlassen eine Weiterleitung, die auf den neuen Home Node verweist • Zugriffe erfolgen entlang der Weiterleitungen • Nicht mehr benötigte Weiterleitungen werden durch die Garbage Collection entfernt • Garbage Collection aktualisiert Referenzen • GAOs unterliegen einer Versionierung

Tabelle 6.1: Übersicht über Migrationsverfahren

Migrationsverfahren	Speicherbedarf pro Knoten
Bidirektionale Referenzen	$\leq n$
Distributed Hash Table	$\leq x$
Lokal konsistente Adressringe	$\leq 2 \cdot x + 2 \cdot y + y \cdot z$
Weiterleitungen	Durchschnittlich: $= \frac{k}{m}$ Schlechtester Fall: $\leq k$

$x \mathrel{\widehat{=}}$ maximale Anzahl gespeicherter GAOs pro Knoten
$y \mathrel{\widehat{=}}$ maximale Anzahl ausgehender Referenzen pro Knoten
$z \mathrel{\widehat{=}}$ konstante Anzahl gespeicherter Abkürzungen pro Referenz
$k \mathrel{\widehat{=}}$ Anzahl clGAOs
$n \mathrel{\widehat{=}}$ Anzahl GAOs
$m \mathrel{\widehat{=}}$ Systemgröße (= Anzahl Knoten)

Tabelle 6.2: Zusätzlich benötigter Speicherbedarf pro Knoten

Lesender und schreibender Zugriff	Zusätzliche Nachrichten		Erklärung
Bidirektionale Referenzen	0		Keine zusätzlichen Nachrichten
Distributed Hash Table	R	R	Weiterleitungen der Zugriffsnachricht an die DHT
Lokal konsistente Adressringe	$1 + G$	1	„GAO migriert“ Nachricht
		G	Weiterleitungen der Zugriffsnachricht innerhalb des Adressrings des zugegriffenen GAOs
Weiterleitungen	L	L	Weiterleitungen der Zugriffsnachricht entlang der Listenstruktur des efGAOs

$R \mathrel{\widehat{=}}$ Anzahl der virtuellen Hops des SSR Routing Algorithmus
$G \mathrel{\widehat{=}}$ Anzahl der virtuellen Hops innerhalb der GAO-Adressringe
$L \mathrel{\widehat{=}}$ Anzahl der clGAOs innerhalb einer GAO-Listenstruktur

Tabelle 6.3: Lesender und schreibender Zugriff auf ein GAO

Erzeugen und Löschen eines GAOs	Anzahl Nachrichten	Erklärung
Bidirektionale Referenzen	0	Keine zusätzlichen Nachrichten
Distributed Hash Table	R	R Weiterleitungen der Aktualisierungsnachricht des DHT-Eintrags
Lokal konsistente Adressringe	0	Keine zusätzlichen Nachrichten
Weiterleitungen	0	Keine zusätzlichen Nachrichten

$R \mathrel{\widehat{=}}$ Anzahl der virtuellen Hops des SSR Routing Algorithmus

Tabelle 6.4: Erzeugen und Löschen eines GAOs (ohne Referenzen)

Referenz-erzeugung	Anzahl Nachrichten		Erklärung
Bidirektionale Referenzen	1	1	Aktualisierungsnachricht
Distributed Hash Table	0		Keine zusätzlichen Nachrichten
Lokal konsistente Adressringe	$3 + G$	1	Integrationsnachricht
		G	Weiterleitungen der Integrationsnachricht
		1	Aktualisierungsnachricht an den Nachfolger des Integrators
		1	Aktualisierungsnachricht an den Initiator
Weiterleitungen	0		Kein zusätzlicher Aufwand

$R \mathrel{\widehat{=}}$ Anzahl der virtuellen Hops des SSR Routing Algorithmus
$G \mathrel{\widehat{=}}$ Anzahl der virtuellen Hops innerhalb der GAO-Adressringe

Tabelle 6.5: Erzeugung einer Referenz auf ein GAO

Referenz-löschung	Anzahl Nachrichten	Erklärung	
Bidirektionale Referenzen	1	1	Aktualisierungsnachricht
Distributed Hash Table	0		Keine zusätzlichen Nachrichten
Lokal konsistente Adressringe	2	2	Aktualisierungsnachrichten an die virtuellen Nachbarn
Weiterleitungen	0		Kein zusätzlicher Aufwand

$R \mathrel{\widehat{=}}$ Anzahl der virtuellen Hops des SSR Routing Algorithmus

Tabelle 6.6: Löschen einer Referenz auf ein GAO

Migration eines GAOs	Anzahl Nachrichten	Erklärung	
Bidirektionale Referenzen	$1 + y + z$	1 y z	Migrationsnachricht Nachrichten zur Aktualisierung der eingehenden Referenzen Nachrichten zur Aktualisierung der ausgehenden Referenzen
Distributed Hash Table	$1 + R$	1 R	Migrationsnachricht Weiterleitungen der Aktualisierungs-nachricht des DHT-Eintrags
Lokal konsistente Adressringe	$1 + 2$	1 2	Migrationsnachricht Aktualisierungsnachrichten an die virtuellen Nachbarn
Weiterleitungen	1	1	Migrationsnachricht

$y \mathrel{\widehat{=}}$ Anzahl der eingehenden Referenzen
$z \mathrel{\widehat{=}}$ Anzahl der ausgehenden Referenzen
$R \mathrel{\widehat{=}}$ Anzahl der virtuellen Hops des SSR Routing Algorithmus

Tabelle 6.7: Migration eines GAOs

7 Implementierung

Im Rahmen dieser Diplomarbeit wurde im Kapitel 6 der *GAO Cloning And Garbage Collecting* (GCAGC) Mechanismus zur Adressierung und Migration von *Global Accessible Objects* (GAOs) vorgestellt. Die Gegenüberstellung mit den anderen Migrationsverfahren zeigt, dass GCAGC im Bezug auf die zusätzlich benötigten Speicherressourcen und der Anzahl ausgetauschter Nachrichten effizient ist. Zur späteren Evaluation (siehe Kapitel 8) und Bestätigung der theoretischen Betrachtung wurde der GCAGC Mechanismus implementiert. Die Implementierung fand in C++ (siehe [22]) statt und baut auf dem bereits vorhandenen Quellcode des *Scalable Source Routing* (SSR) Protokolls auf.

Es sei angemerkt, dass die momentane Realisierung des GCAGC Mechanismus von der Implementierung der ACVM abstrahiert wurde. Dies ermöglicht zum einen eine schrittweise Umsetzung von GCAGC und zum anderen kann die Evaluation mittels eines Simulators erfolgen. Ohne eine geeignete Abstraktion der ACVM wäre es erforderlich, für jeden simulierten Knoten eine Instanz der ACVM auszuführen. Ab einer gewissen Netzgröße ist dies auf Grund der für die Ausführung der verteilten ACVM zusätzlich benötigten Rechenkapazitäten nicht mehr durchführbar. Zusätzlich müsste eine verteilt ausführbare Garbage Collection, die die Adressierbarkeit von GAOs voraussetzt, bereits zur Verfügung stehen. Zum Zeitpunkt dieser Arbeit ist dies nicht der Fall.

Diese Betrachtungsweise hat jedoch zur Folge, dass die ACVM und der GCAGC Mechanismus in einem zusätzlichen Schritt miteinander verbunden werden müssen. Dieser Schritt findet nicht im Rahmen dieser Diplomarbeit statt. Es wird an den entsprechenden Stellen auf Berührungspunkte zwischen den Komponenten von GCAGC und der ACVM hingewiesen.

Nachfolgend werden die für GCAGC wichtigen Klassen, deren Zusammenspiel und die Abhängigkeiten mit der Implementation des SSR Protokolls dargestellt. Darüber hinaus werden die Komponenten beschrieben, die für die Abstraktion und Simulation der Funktionsweise der ACVM benötigt werden.

7.1 cGAO

Zentrale Komponente zur Implementierung eines Adressierungs- und Migrationsmechanismus ist die Klasse *cGAO*. Jede Instanz dieser Klasse stellt eine Repräsentation des zugehörigen GAOs dar. Zusätzlich werden die Referenzen und Informationen, die

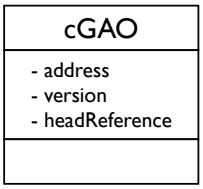


Abbildung 7.1: cGAO - UML Klassendiagramm

für die Laufzeitumgebung und die Adressierbarkeit des jeweiligen GAOs notwendig sind, gespeichert.

Im Detail bedeutet dies, dass eine cGAO Klasseninstanz den Speicherbereich enthält, der durch das zugehörige GAO definiert ist. Darüber hinaus wird für jedes GAO die eindeutige SSR-Adresse innerhalb der Instanz gespeichert. Für den GCAGC Mechanismus wird die cGAO Klasse um Informationen bezüglich

- der Adresse,
- der Version und
- der Head Referenz

erweitert. Die Adresse dient als global eindeutiges Identifikationsmerkmal der cGAOs und wird aus dem SSR Adressraum gewählt. cGAOs bzw. deren Inhalte sind nur lesbar. Durch einen schreibenden Zugriff wird das betreffende cGAO exakt kopiert, dessen Inhalt verändert und die Version erhöht. Der Migrationsstatus ist implizit über die Head Referenz gekennzeichnet. Verweist diese auf kein weiteres cGAO so handelt es sich bei dem aktuell betrachteten cGAO um das *effective GAO* (efGAO), andernfalls um ein *cloned GAO* (clGAO). Bei clGAOs verweist die Head Referenz auf das efGAO bzw. auf das nächste clGAO, welches wiederum in Richtung des efGAOs verweist.

Bezüglich der Simulation werden die zu einem GAO gehörenden ausgehenden Referenzen in einer geeigneten Datenstruktur gehalten. Somit ist das dynamische Hinzufügen und Entfernen von Referenzen möglich. Im Gegensatz dazu ist dieser Aufwand für die ACVM nicht notwendig, da für jedes Objekt die Anzahl der zur Laufzeit verwendeten Referenzen während des Übersetzungsvorgangs bekannt ist. Deren Speicherung erfolgt somit statisch.

7.2 cGlobalReference

Im Kapitel 5 wurde beschrieben, dass eine globale Referenz auf ein GAO aus der eindeutigen SSR-Adresse des referenzierten GAOs und des zugehörigen Home Nodes besteht. Zur Verbesserung wird zusätzlich die Source Route verwendet, die zu dem

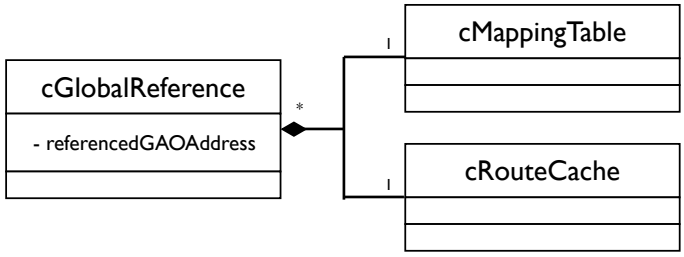


Abbildung 7.2: cGlobalReference - UML Klassendiagramm

Home Node des referenzierten GAOs führt. Source Routen werden hierbei innerhalb des SSR Route Caches gespeichert.

Die Klasse cGlobalReference dient als Repräsentation beider Informationen. Jede Instanz speichert explizit die SSR-Adresse des referenzierten GAOs. Zusätzlich hält die Klasse zwei statische Zeiger auf die Abbildungstabelle und den Route Cache des zugehörigen Knotens. Mittels der Abbildungstabelle wird der Home Node des referenzierten GAOs ermittelt. Die Source Route, die zu diesem Home Node führt, wird durch den Route Cache zur Verfügung gestellt. Die Abbildungstabelle und der Route Cache existieren genau einmal auf dem lokal zugehörigen Knoten. In Abbildung 7.2 wird die Abbildungstabelle durch die Klasse cMappingTable und der Route Cache durch die Klasse cRouteCache repräsentiert.

Mit Hilfe einer cGlobalReference können somit Nachrichten in Verbindung mit der Source Route an den zugehörigen Home Node gesendet werden. Es ist dabei nicht erforderlich, dass die Source Route vollständig vorliegt. Sollte die zu einem Home Node zugehörige Source Route nur unvollständig oder gar nicht im Route Cache gespeichert sein, so muss der SSR Routing Algorithmus (siehe Kapitel 4.2) verwendet werden.

7.3 cMsgGAO

Die Adressierung von GAOs basiert auf dem Austausch von Nachrichten zwischen den involvierten Knoten des verteilten Systems. Dazu ist es notwendig, dass Nachrichten vom Quellknoten zum Zielknoten geroutet werden können. Das SSR Protokoll realisiert die hierfür notwendigen Mechanismen. Für Ende-zu-Ende Nachrichten existiert in der Implementierung des SSR Protokolls die Klasse cMsgConnect.

Die Klasse cMsgGAO ist die Oberklasse aller Nachrichten, die mit der Adressierung und Migration von GAOs involviert ist. Diese Nachrichten sind ausschließlich Ende-zu-Ende Nachrichten. Sie werden entlang von cGlobalReferences, genauer gesagt entlang der Source Routen gesendet und anhand des vorgegebenen Pfads durch das Netzwerk geroutet. Aus diesem Grund beerbt cMsgGAO die Klasse cMsgConnect und erhält somit deren Ende-zu-Ende Übertragungsfähigkeit.

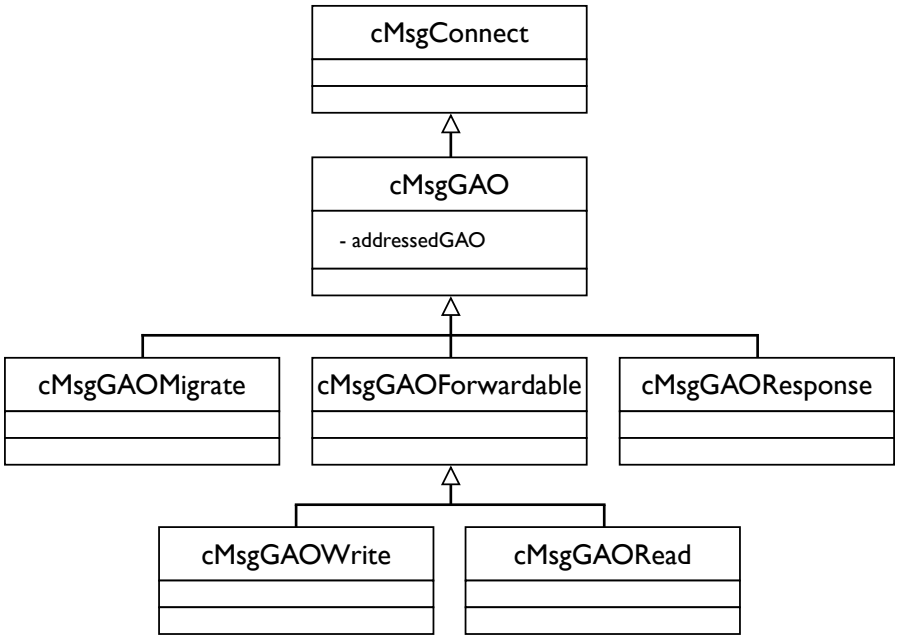


Abbildung 7.3: cMsgGAO - UML Klassendiagramm

Darüber hinaus erweitert cMsgGAO die Klasse cMsgConnect um Informationen, die für die Adressierung und Migration von GAOs benötigt werden. Dazu zählt die eindeutige SSR-Adresse des GAOs, das durch die cMsgGAO Instanz adressiert wird. Die Klassen cMsgGAORead, cMsgGAOResponse und cMsgGAOWrite stellen den lesenden und schreibenden Zugriff auf GAOs zur Verfügung. Des weiteren wird die Klasse cMsgGAOMigrate für Migrationen von GAOs verwendet. Innerhalb der entsprechenden Unterklassen von cMsgGAO werden die jeweils benötigten, klassenspezifischen Funktionen implementiert. Der Empfang einer cMsgGAO Nachricht löst die Ausführung der zum Nachrichtentyp gehörenden *handle()* Funktion aus. Diese wiederum führt klassenspezifische Codeteile aus.

Nach Migrationen von GAOs ist das Weiterleiten von Nachrichten, die für den lesenden oder schreibenden Zugriff verwendet werden, entlang der Head Referenz (siehe Kapitel 6.2.4) notwendig. Diesbezüglich existiert die Klasse cMsgGAOForwardable, die als direkte Unterklasse von cMsgGAO die entsprechende Funktionalität zur Verfügung stellt. cMsgGAOForwardable erweitert als Oberklasse die beiden Klassen cMsgGAORead und cMsgGAOWrite, da diese nach Migrationen von GAOs weitergeleitet werden müssen.

7.4 Simulatorspezifische Komponenten

Die zuvor beschriebene Implementierung des GCAGC Verfahrens wird mit Hilfe einer Simulationsumgebung evaluiert (siehe Kapitel 8). Die Simulationsumgebung umfasst dabei das darunterliegende Netzwerk, das SSR Protokoll und das GCAGC Verfahren. Zusätzlich wird die Funktionsweise der ACVM simuliert. Auf Grund der daraus resultierenden Abstrahierung der ACVM bedarf es einiger Komponenten, die eine realitätsnahe Abbildung auf die Simulationsumgebung gewährleisten. Die Art der Abbildung wird dabei durch GCAGC beeinflusst. Das heißt es müssen diejenigen Komponenten der ACVM simuliert werden, die einen Einfluss auf das Adressierungsverfahren haben.

Diesbezüglich wird eine Klasse benötigt, die die Verwaltung und Speicherung von cGAOs innerhalb der Simulationsumgebung realisiert. Des weiteren bedarf es einer Garbage Collection, da das GCAGC Verfahren auf der regelmäßigen Ausführung einer Garbage Collection basiert und davon die Leistungsfähigkeit von GCAGC abhängt. Für eine realitätsnahe Abbildung der ACVM ist es ebenfalls notwendig, die Reihenfolge der Objektzugriffe innerhalb der Simulationsumgebung genauso wie in der lokal lauffähigen ACVM durchzuführen. Dazu wurde eine Ablaufsteuerung implementiert, die Objektzugriffe, Migrationen und Garbage Collections initiiert. Die Ausführung der Abläufe, die von der Ablaufsteuerung initiiert werden, geschieht durch die jeweils betroffenen Komponenten. Diese werden im Folgenden beschrieben.

7.4.1 cObjectRegister

Die Klasse cObjectRegister speichert in einer internen Datenstruktur die auf dem jeweiligen Knoten vorgehalten Objekte und ermöglicht lesenden und schreibenden Zugriff. Es findet dabei eine Überprüfung statt, ob das aktuell zugegriffene Objekt lokal oder auf einem entfernten Knoten gespeichert ist. Im letzteren Fall handelt es sich um ein GAO und es wird der im Kapitel 5 beschriebene Adressierungsmechanismus verwendet. Andernfalls kann der Zugriff lokal abgearbeitet werden. Des weiteren existiert die Funktionalität, die für Migrationen von GAOs erforderlich ist.

Pro Knoten existiert das cObjectRegister genau einmal. Ein modifiziertes *Singleton* Entwurfsmuster (siehe [4]) gewährleistet diese Einmaligkeit für jeden Knoten und stellt darüber hinaus einen einfachen Zugriff auf diese Instanz zur Verfügung.

In Hinblick auf die Integration des cObjectRegisters mit der ACVM ist zu beachten, dass bei jedem Objektzugriff dessen lokale Speicherung überprüft werden muss. Auf Grund der Häufigkeit von Objektzugriffen, ist ein entsprechend performanter Ansatz obligatorisch und betrifft den Aufbau und die Funktionalität der internen Datenstruktur.

7.4.2 cRuntimeMgmt

Der Programmablauf der ACVM wird durch die Klasse cRuntimeMgmt simuliert. Dazu gehört zum einen die Abbildung der Objektzugriffe auf die Simulationsumgebung und andererseits die Durchführung der Garbage Collection. Des weiteren initialisiert die Klasse cRuntimeMgmt die statische Verteilung der GAOs auf die Knoten des Netzwerks zu Beginn der Simulation. Jeder Knoten des Netzwerks besitzt eine Instanz von cRuntimeMgmt.

Gesteuert werden die Abläufe durch die sequentielle Abarbeitung des *TraceLogs* (siehe Kapitel 7.4.3). Dieser gibt die für den jeweiligen Simulationsschritt auszuführende Aktion vor. Das Fortschreiten von Simulationsschritt zu Simulationsschritt wird durch den eigentlichen Simulator (siehe Kapitel 8.2) mittels eines Timers durchgeführt. Nach Ablauf des Timers wird durch den Simulator die Funktion *next()* aufgerufen und somit zum nächsten Simulationsschritt übergegangen.

Alle im Simulator pro Knoten existierenden Instanzen von cRuntimeMgmt verfügen über ein „globales Wissen“, das den aktuell gültigen Simulationsschritt darstellt. Das bedeutet, dass Simulationsschritt i auf Knoten A ausgeführt wird, falls dies auf Grund des simulierten Programmablaufs notwendig ist. Zum Beispiel erfordert die Migration eines GAOs, dass die Migrationsnachricht von dem Knoten, der das zu migrierende efGAO speichert, versendet wird. Die nachfolgende Inkrementierung des Simulationsschrittes ist auf Grund des globalen Wissens für alle Knoten gültig. Der nächste Simulationsschritt $i+1$ kann somit auf einem anderen Knoten B ausgeführt werden.

7.4.3 cTraceLog

Die Klasse cTraceLog stellt eine Schnittstelle zur Verfügung, mit deren Hilfe die durch die Ablaufsteuerung ausgeführten Aktionen beeinflusst werden können. Bei den Aktionen handelt es sich um

- lesende Zugriffe auf GAOs,
- schreibende Zugriffe auf GAOs,
- Migrationen von GAOs,
- Durchführung von Garbage Collections.

Dazu werden die jeweiligen Aktionen und die dazugehörigen Informationen innerhalb einer internen Datenstruktur gespeichert. Zum Beispiel existiert in cTraceLog für einen lesenden Zugriff auf ein GAO die Information, von welchem GAO auf welches GAO zugegriffen wird. Im Gegensatz dazu ist für eine Migration von Interesse, welches GAO zu welchem neuen Knoten migriert wird.

Für alle im vorhergehenden genannten Aktionen ist der ausführende Knoten von Interesse. Dieses Wissen stellt der cTraceLog zur Verfügung und ermöglich somit die Initiierung des nächsten Simulationsschrittes auf dem entsprechenden Knoten. Die Ermittlung dieses Knotens ist dabei abhängig von der auszuführenden Aktion und erfordert gegebenenfalls ein „globales Wissen“, das für Simulationszwecke vorhanden ist. Zur Reproduzierbarkeit werden die Aktionen in einer Datei, dem sogenannten *TraceFile*, gespeichert und damit die Instanz von cTraceLog initialisiert. Für ein Beispiel eines TraceFiles wird an dieser Stelle auf Kapitel 8.2 und die Abbildung 8.1 verwiesen.

7.4.4 cGarbageCollection

Der GCAGC Mechanismus basiert auf der Ausführung einer Garbage Collection. Dadurch werden alle nicht mehr referenzierten Objekte von allen Knoten des verteilten Systems gelöscht. In Kombination mit GCAGC werden alle Referenzen, die auf *cloned GAOs* verweisen, aktualisiert. Nach der Durchführung der Garbage Collection verweisen alle sich im System befindlichen Referenzen auf deren zugehörige *effective GAOs*.

In Ermangelung eines bestehenden Verfahrens zur Durchführung einer verteilten Garbage Collection wurde ein *Mark and Sweep* Algorithmus (siehe [34]) implementiert. Der Algorithmus arbeitet in zwei Phasen und wird zentral durch den Simulator auf allen Objekten des Systems ausgeführt. Das bedeutet, dass beide Phasen globalen Zugriff auf alle Objekte des verteilten Systems benötigen. Die Objekte werden von den einzelnen Knoten mittels der zugehörigen cObjectRegister zur Verfügung gestellt. Eine derartige Funktionsweise ist nur innerhalb des Simulators gegeben.

Nachdem die Garbage Collection gestartet wurde, beginnt die Markierungsphase. Ausgehend vom Wurzelobjekt (= Hauptthread) des Referenzgraphen (siehe Kapitel 5.1) wird dieser zunächst mit dem aktuellen Zeitstempel markiert und anschließend alle ausgehenden Referenzen traversiert. Jede traversierte Referenz führt zu einem weiteren Objekt. Im Falle eines efGAOs wird dieses markiert und dessen ausgehende Referenzen ebenfalls traversiert. Handelt es sich bei dem aktuellen Objekt jedoch um ein clGAO, so wird dessen Head Referenz und die eventuell folgenden Head Referenzen bis hin zum efGAO traversiert. Darüber hinaus wird die Referenz, die auf das clGAO verweist, mit dem efGAO aktualisiert. Dies wird solange fortgesetzt, bis alle efGAOs des zusammenhängenden Referenzgraphen komplett markiert wurden. Anschließend beginnt die Aufräumphase. Hierbei werden für jeden Knoten alle vorgehaltenen Objekte auf eine Markierung überprüft. Ist ein Objekt mit dem aktuellen Zeitstempel markiert, so befindet sich dieses in der Zusammenhangskomponente des Referenzgraphen, der das Wurzelobjekt enthält. Andernfalls wird das Objekt nicht mehr referenziert und kann gelöscht werden, da es vom eigentlichen Referenzgraphen isoliert wurde.

8 Evaluation

Dieses Kapitel beschäftigt sich mit der Evaluation des in Kapitel 6.2.4 beschriebenen Migrationsverfahrens *GAO Cloning And Garbage Collecting* (GCAGC), das auf dem Adressierungsverfahren aus Kapitel 5 aufsetzt und Weiterleitungen verwendet. Bezüglich der Evaluation wird dabei das Verhalten von GCAGC hinsichtlich der Performanz betrachtet. Die Performanz ist, wie im Kapitel 6.3 erläutert, vom zusätzlich erforderlichen Speicher- und Kommunikationsaufwand abhängig, der über den unvermeidbaren Aufwand des verwendeten Adressierungsverfahrens hinausgeht. An Hand welcher Kriterien die Performanz gemessen wird, erläutert Kapitel 8.1.

Die Evaluation basiert auf OMNet++ (siehe [33]). Dabei wird ein Netzwerk und der durch die ACVM definierte Programmablauf simuliert. Auf dem simulierten Netzwerk wird das *Scalable Source Routing* (SSR) Protokoll (siehe Kapitel 4) ausgeführt, das um den GCAGC Mechanismus erweitert wurde. Auf der somit realisierten Adressierbarkeit von GAOs setzt die abstrahierte Abbildung der ACVM auf. Das Kapitel 8.2 beschreibt die Details der zur Evaluation verwendeten Simulationsumgebung.

Abschließend werden die durch die Simulation erhaltenen Evaluationsergebnisse im Kapitel 8.3 dargestellt. Darüber hinaus findet eine Bewertung der Ergebnisse statt.

8.1 Evaluationskriterien

Zentraler Punkt der Evaluation ist die Fragestellung, wie sich das GCAGC Verfahren bei lesenden und schreibenden Zugriffen auf GAOs, die auf verschiedenen Knoten des Netzwerks vorgehalten werden, bezüglich der Performanz verhält. Das dynamische Erzeugen und Löschen von GAOs und Referenzen wird nicht betrachtet, da diese Operationen in Verbindung mit GCAGC keinen zusätzlichen Aufwand hervorrufen (siehe Kapitel 6.3).

Zugriffe auf GAOs werden durch die Längen der GAO-Listenstrukturen beeinflusst, die durch Migrationen von GAOs entstehen. Die Länge einer Listenstruktur ist durch die Anzahl der zugehörigen clGAOs definiert. Jeder Zugriff auf ein clGAO erfordert die Weiterleitung der Nachricht an das efGAO und erzeugt folglich einen zusätzlichen Kommunikationsaufwand in Abhängigkeit der Listenlänge. Des weiteren legt die Länge der Listenstrukturen den zusätzlichen Speicheraufwand von GCAGC fest, da die clGAOs, genauer gesagt die Head Referenzen, gespeichert werden müssen. Wie im Kapitel 6.3.1 gezeigt, ist die Länge bzw. die Anzahl der clGAOs nicht beschränkt, sondern von der Migrationsrate und der Rate der Garbage Collections abhängig. Diese Abhängigkeit gilt es zu evaluieren.

Ein weiteres Evaluationskriterium ist die Anzahl der Weiterleitungen beim Zugriff auf ein GAO. Die Anzahl ist immer kleiner oder gleich der Länge der jeweiligen Listenstruktur, da ein GAO auf ein clGAO verweisen kann, welches sich an einer beliebigen Position innerhalb der Liste befindet. Dies ist dadurch begründet, dass die Antwortnachricht bezüglich eines lesenden Zugriffs die verwendete Referenz auf das efGAO aktualisiert. Abwechselnde Migrationen und lesende Zugriffe auf dasselbe GAO resultieren somit in ausgehenden Referenzen, die auf clGAOs mit unbestimmten Positionen innerhalb der Listenstruktur verweisen. Zusätzlich zur Anzahl der Weiterleitungen ist die Anzahl der sich aus den Weiterleitungen ergebenden physikalischen Hops von Interesse. Die Anzahl der physikalischen Hops gilt es zu minimieren.

Zur Evaluation der Anzahl von Weiterleitungen ist es wichtig, dass für Objektzugriffe ein realer Ablauf eines Programms (siehe Kapitel 8.2) zu Grunde gelegt wird. Mit Hilfe des realen Programmablaufs werden während der Simulation Zugriffe auf GAOs initiiert. Somit werden Referenzen, wie oben beschrieben, entsprechend aktualisiert. Daraus folgt, dass die in der Praxis zu erwartenden Ergebnisse besser durch die Simulation widergespiegelt werden. Über alle durchgeführten Zugriffe wird der Durchschnitt für die Anzahl der Weiterleitungen gebildet. Je näher dieser Wert an Null liegt, desto mehr nähert sich GCAGC dem optimalen Verfahren an.

Das optimale Verfahren entspricht dem in Kapitel 5 beschriebenen Adressierungsverfahren. Für Migration werden alle betroffenen und somit ungültigen, eingehenden Referenzen automatisch aktualisiert, so dass diese auf den neuen Home Node verweisen. Die Aktualisierungen sind in Bezug auf das optimale Verfahren mit keinen Kosten verbunden. Da dies ohne zusätzliche Informationen nicht möglich ist (siehe Kapitel 6.2), entspricht das optimale Verfahren einem theoretischen Modell.

Zusammenfassend werden für GCAGC die Kriterien

- Länge der Listenstrukturen (= Anzahl von clGAOs)
- Anzahl Weiterleitungen bei Zugriffen auf GAOs
- Anzahl physikalischer Hops bei Zugriffen auf GAOs

evaluiert. Alle Kriterien werden hinsichtlich des Durchschnitts, der sich aus allen Zugriffen ergibt, ausgewertet.

8.2 Simulationsumgebung

Die Evaluation von GCAGC basiert auf einer Simulationsumgebung, die aus dem eigentlichen Netzwerksimulator und einer Simulation des Programmablaufs der ACVM besteht. Als Netzwerksimulator wird OMNet++ (siehe [33]) verwendet. OMNet++ ist eine C++-basierte, diskrete Ereignissimulation mit dem Hauptziel der Simulation von Computernetzwerken. Hierbei können die verschiedensten Netzgrößen und

Netztopologien simuliert werden. Zur Evaluation wird für die Netztopologie eine reguläre Gitterstruktur vorausgesetzt. Dies vereinfacht den Migrationsmechanismus, da die Migrationsroute auf Grund der bekannten Netztopologie einfach ermittelt werden kann. Die Größe der Gitterstruktur wird während der Evaluation verändert. Somit kann ein nicht offensichtlicher Zusammenhang von GCAGC zur Anzahl der Knoten aufgezeigt werden. Auf dem simulierten Netzwerk setzt das SSR Protokoll auf, wobei jeder Knoten der Simulation eine unabhängige SSR Instanz ausführt. Die Instanzen enthalten die für GCAGC notwendigen Codeteile.

Die Programmablaufsimulation der ACVM wird durch die Klasse cRuntimeMgmt (siehe Kapitel 7.4.2) realisiert. Hierzu wird zu einem bestimmten Simulationszeitpunkt die Ausführung des Programmablaufs durch den OMNet++ Simulator gestartet. Im weiteren Verlauf der Simulation werden Objektzugriffe durch den simulierten Programmablauf initiiert und mittels GCAGC auf dem simulierten Netzwerk abgebildet. Jeder erfolgreich ausgeführte Programmschritt setzt einen Timer des Simulators, nach dessen Ablauf der nächste Programmschritt ausgelöst wird. Dies entspricht der Ausführung bzw. Simulation eines Threads der realen ACVM.

Der Programmablauf wird durch ein *Trace File* definiert und zur Initialisierung der Klasse cTraceLog (siehe Kapitel 7.4.3) benötigt. cTraceLog dient der Programmablaufsimulation als Schnittstelle für den Zugriff auf das Trace File. Im vorhergehenden wurde für das Evaluationskriterium „Anzahl von Weiterleitungen“ die Notwendigkeit hervorgehoben, dass der Programmablauf realen Programmen entsprechen muss. Aus diesem Grund wird das Trace File aus der lokal lauffähigen Implementierung der ACVM durch einen dreistufigen Prozess generiert. Im ersten Schritt wird eine Java Anwendung, die zur Evaluation verwendet werden soll, in einen BLOB transkodiert. Im zweiten Schritt findet die Ausführung des BLOBs auf einer realen, unter Linux lauffähigen ACVM statt. Während der Laufzeit des Programms wird auf die zur Ausführung relevanten, lokalen Objekte zugegriffen. Die Objektzugriffe und viele weitere Details zum internen Ablauf werden von der ACVM ausgegeben. Im dritten Schritt wird diese Ausgabe mit Hilfe eines Scripts analysiert, die Objektzugriffe extrahiert und ein Trace File erstellt.

Zusätzlich zu den Objektzugriffen werden auch die Zeitpunkte, an denen Migrationen und Garbage Collections ausgeführt werden sollen, innerhalb des Trace Files festgelegt. Dies hat den Vorteil, dass die Simulationsergebnisse reproduzierbar sind. Die Häufigkeit von Migrationen und Garbage Collections ist von der jeweils gewählten Rate abhängig. Das bedeutet, dass die Migrationsrate das Verhältnis der Anzahl der Migrationen zur Anzahl aller Operationen (= lesende und schreibende Zugriffe, Migrationen, Garbage Collections) darstellt. Für die Rate der Garbage Collections gilt dies analog. Darüber hinaus kann die zur Erstellung des Trace Files verwendete *Migrationsstrategie* variiert werden. Die Migrationsstrategie legt fest, welche GAOs zur Migration ausgewählt und auf welche Knoten diese schließlich migriert werden. Es sind mehrere Strategien (siehe [19] und Kapitel 9) denkbar, die aber nicht im Rahmen dieser Arbeit diskutiert werden. Bezüglich der Evaluation von GCAGC

step	dstNode	migrateObject	fromObject	toObject	mode
0	-	-	1	2	1
1	-	-	1	3	1
2	-	-	-	-	4
3	B	3	-	-	3
4	-	-	1	2	2
5	C	3	-	-	3
6	-	-	3	4	1
⋮	⋮	⋮	⋮	⋮	⋮

	global eindeutige SSR-Adressen für			
Knoten	GAOs	GAOs	GAOs	

lesender Zugriff ≙ 1
schreibender Zugriff ≙ 2
Migration ≙ 3
Garbage Collection ≙ 4

Abbildung 8.1: Ausschnitt aus einem Trace File

und der damit einhergehenden Simulation wird für Migrationen die Zufallsstrategie verwendet. Das bedeutet, dass das Objekt und der Zielknoten für jede Migration zufällig und voneinander unabhängig gewählt werden.

Das Trace File stellt eine sequenzielle Abfolge von lesenden und schreibenden Objektzugriffen, Migrationen und Garbage Collections dar. Abbildung 8.1 zeigt einen Ausschnitt aus einem Trace File, das den in Abbildung 5.1 dargestellten Referenzgraphen durch Migrationen derart im Netzwerk verteilt, dass der Referenzgraph aus Abbildung 6.7 entsteht. Die erste Spalte (*step*) gibt für jede Zeile den entsprechenden Programmablaufschritt und die letzte Spalte (*mode*) die zu diesem Schritt zugehörige Operation an. Als Operationen stehen lesende und schreibende Zugriffe (mode=1 und mode=2), Migrationen (mode=3) und Garbage Collections (mode=4) zur Verfügung. Für lesende und schreibende Zugriffe wird die Information benötigt, auf welches Objekt zugegriffen wird (*toObject*). Darüber hinaus muss innerhalb des simulierten Programmablaufs bekannt sein, welches Objekt (*fromObject*) durch den simulierten Thread für den Zugriff auf ein anderes Objekt verwendet wird. Somit kann die für diesen Zugriff benötigte Referenz ermittelt werden. Das Trace File beinhaltet für Migrationen die Information, welches Objekt (*migrateObject*) zu welchen Knoten (*dstNode*) migriert werden soll. Da sich der Home Node des betreffenden efGAOs während der Simulation durch Migrationen ändern kann, muss der Home Node und die Migrationsroute im Vorfeld durch die Simulationsumgebung ermittelt werden. Die Garbage Collection benötigt keinerlei weitere Informationen, da sie global auf dem simulierten Netzwerk ausgeführt wird. Einträge im Trace File bezüglich

der Objekte und Knoten stellen immer deren eindeutige SSR-Adressen dar.

Zusätzlich zum Trace File wird ein statischer Referenzgraph erstellt. Mit Hilfe dieses Referenzgraphen werden zu Beginn der Simulation durch die Programmablaufsteuerung alle Objekte und deren zugehörige Referenzen auf einem ausgewählten Knoten initialisiert. Zur Vereinfachung der Abbildung der ACVM wurde auf die Verwendung eines dynamischen Referenzgraphen verzichtet. Darüber hinaus werden alle Objekte des Referenzgraphen bereits bei der Initialisierung als GAOs betrachtet. Somit sind Promotionen von Objekten zu GAOs (siehe Kapitel 3) nicht notwendig. Diese Betrachtungsweise stellt für das Adressierungsverfahren den schlechtesten Fall dar.

Ein dynamischer Referenzgraph unterscheidet sich von dem statischen Referenzgraphen dahingehende, dass Objekte und Referenzen während der Ablaufsimulation dynamisch erzeugt und gelöscht werden. Für das GCAGC Verfahren ist das Erzeugen und Löschen von Objekten und Referenzen mit keinem zusätzlichen Kommunikationsaufwand und keinem zusätzlichen Speicheraufwand verbunden (siehe Kapitel 6.3.2). Jedoch kann es auf Grund des statischen Referenzgraphen vorkommen, dass ein nicht mehr vorhandenes Objekt migriert wird. Da auf dieses Objekt nicht mehr zugegriffen wird, werden die Evaluationskriterien bezüglich Objektzugriffe nicht beeinflusst. Somit stellt der Verzicht auf einen dynamischen Referenzgraphen keine Beschränkung für die Evaluation von GCAGC dar.

Die Erstellung des Trace Files und des statischen Referenzgraphen basiert auf der Ausgabe der ACVM, die wiederum von der ausgeführten Java Anwendung abhängig ist. Als Anwendungen wurden für die Evaluation zwei dynamische Datenstrukturen ausgewählt, da diese in realen Programmen häufig verwendet werden. Bei den Datenstrukturen handelt es sich um eine einfach verkettete Liste und einen Rot-Schwarz Baum. Die interne Struktur des Rot-Schwarz Baums (siehe [14]) und die damit verbundenen Operationen zur Reorganisation der Struktur erfordern Objektzugriffe, die das Adressierungsverfahren zusätzlich „belasten". Zum Vergleich wurde eine einfach verkettete Liste gewählt, die keine Reorganisation und somit keine zusätzlichen Objektzugriffe benötigt. Beide Datenstrukturen realisieren ein Wörterbuch, das die Operationen Einfügen, Löschen und Finden von Daten unterstützt. Jedes Element des Wörterbuchs repräsentiert ein Objekt des statischen Referenzgraphen. Beide Implementierung basieren auf [26] und werden separat für die Evaluation von GCAGC verwendet.

Mit Hilfe der beiden Wörterbuchimplementierungen werden Trace Files, wie bereits erläutert, erstellt und die Einträge für die Ausführung von Migrationen und Garbage Collections entsprechend der gewählten Häufigkeiten erzeugt. Zur endgültigen Berechnung von allen Simulationen werden die Parameter

- Netzgröße
- Migrationsrate
- Rate der Garbage Collections
- Verwendete Wörterbuchimplementierung

systematisch variiert, die entsprechenden Konfigurationsdateien erzeugt und die Simulationen gestartet. Die Daten, die sich durch alle Simulationsläufe ergeben und deren Bewertung, werden im Folgenden beschrieben.

8.3 Evaluationsergebnisse

Nach Abschluss der Simulationsläufe wurden die Daten entsprechend der im Kapitel 8.1 beschriebenen Evaluationskriterien ausgewertet. Die Details zu den Simulationen und der daraus resultierenden Ergebnisse werden im Folgenden dargestellt.

Die Rahmenbedingungen für jeden Simulationslauf sind, wie bereits oben angeführt, von mehreren Parametern abhängig. Als Netzgröße wurde die Anzahl der Knoten für die Gitterstruktur auf 9, 25 und 49 Knoten festgelegt. Auf Grund der zeitlichen Beschränkung war es nicht möglich, weitere Gitter mit 100, 1000 und mehr Knoten zu simulieren. Jedoch können durch die Ergebnisse der simulierten Netzgrößen Rückschlüsse auf Gitter mit einer größeren Anzahl von Knoten gezogen werden. Darüber hinaus wurde derjenige Knoten zur Simulation des ausführenden Hauptthreads ausgewählt, der sich in der Mitte der Gitterstruktur befindet. Damit geht einher, dass die Größe des Netzwerks dem *Migrationsradius* der Migrationsstrategie entspricht. Der Migrationsradius ist durch die maximal mögliche Hop-Distanz zwischen einem migrierten GAO und dem Knoten, der den Thread simuliert, definiert. Für die Rate der Garbage Collections (GC Rate) wurden die Simulationen mit 0 % und 1 % Anteil im Vergleich zur Gesamtanzahl der Operationen ausgeführt. Eine Garbage Collection Rate von 0 % stellt für den GCAGC Mechanismus den schlechtesten Fall dar, da somit keine vollständige Aktualisierung von allen ausgehenden Referenzen stattfindet.

Des weiteren wird die Migrationsrate im Bereich von 0 % bis 70 % verändert. Eine Migrationsrate von 70 % stellt für ein verteiltes System einen extremen Wert dar und würde während des Betriebs eines realen Systems nicht so hoch gewählt werden. Jedoch ergeben sich somit die Grenzen des GCAGC Mechanismus während der Simulationen. Es sei an dieser Stelle angemerkt, dass die tatsächliche Migrationsrate, die sich unter realen Umständen in der verteilten ACVM ergibt, von der verwendeten Migrationsstrategie und den somit festgelegten Betriebszielen abhängig ist.

In der Tabelle 8.1 werden die Randbedingungen aufgeführt, die sich aus den zur Evaluation verwendeten Datenstrukturen ergeben. Die hohe Anzahl von verwendeten Referenzen folgt aus der Beschränkung auf einen statischen Referenzgraphen und

der iterativen Traversierung der jeweiligen Datenstruktur. Wird auf ein Element der Datenstruktur zugegriffen, so traversiert der Hauptthread unter Verwendung des „Wurzelobjekts“ der Datenstruktur alle anderen Objekte und benötigt dazu deren zugehörige Referenzen. Dies beeinflusst aber nicht die Charakteristik der Zugriffe.

Gesamtanzahl	Lineare Liste	Rot-Schwarz Baum
Objekte	125	126
Verwendete Referenzen	202	206
Lesende Zugriffe	15700	10621
Schreibende Zugriffe	393	1963

Tabelle 8.1: Randbedingungen der Datenstrukturen

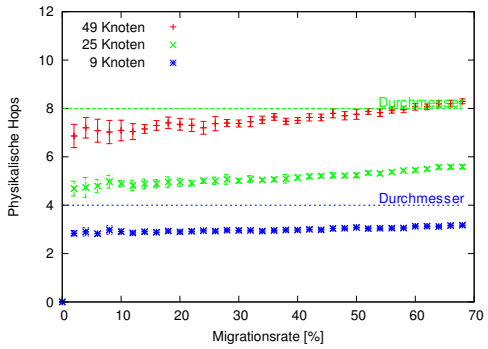
Nachfolgende Diagramme bzw. Abbildungen stellen das Verhalten von GCAGC bezüglich der Migrationsrate dar. An der x-Achse wird die Migrationsrate in Prozent angetragen. Die Evaluationskriterien werden auf der y-Achse angetragen. Die „Kurve“ eines Diagramms entspricht einer bestimmten Rate von Garbage Collections. Jeder Punkt der Kurve entspricht einem Simulationslauf, der mit einer bestimmten Parameterbelegung durchgeführt wurde. Zur Vermeidung von statistischen Fehlern wird jeder Simulationslauf mit unabhängig voneinander erzeugten Trace Files wiederholt berechnet. Der sich aus der mehrmaligen Berechnung ergebende mittlere Fehler (siehe [13], Kapitel 3) ist zu jedem Punkt mit eingezeichnet.

Anzahl physikalischer Hops

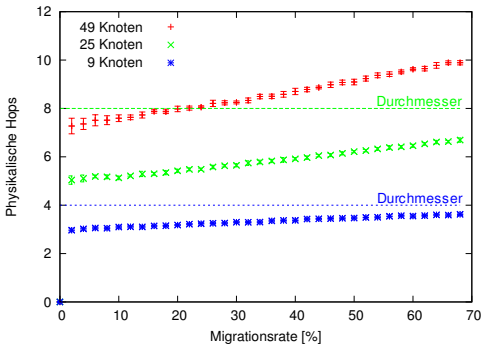
Nachfolgende Abbildung zeigt für beide Datenstrukturen die durchschnittliche Anzahl physikalischer Hops, die für einen Zugriff auf ein entferntes GAO benötigt werden. Jeder Zugriff erfordert das Versenden einer Nachricht an den Home Node des betroffenen GAOs. Die Nachrichten werden dabei entlang der Referenzen, genauer gesagt der zugehörigen Source Routen durch das Netzwerk geroutet und erreichen den Zielknoten mit einer bestimmten Anzahl physikalischer Hops.

Im schlechtesten Fall (0 % GC Rate) gilt für die Anzahl der physikalischen Hops, dass diese linear ansteigen (siehe Abbildungen 8.2(a) und 8.2(b)). Je größer das Netzwerk ist, desto steiler erfolgt der Anstieg. Der Anstieg ist darüber hinaus davon abhängig, welche Datenstruktur verwendet wird. Im Gegensatz dazu gilt für ein Gitter mit 9 Knoten und der Rot-Schwarz Baum Implementierung, dass die Anzahl der Hops nahezu konstant ist und somit eine Nachricht durchschnittlich rund 3 Hops pro GAO-Zugriff benötigt (siehe Abbildung 8.2(a)).

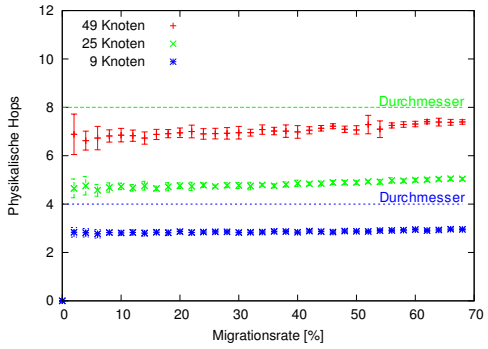
Wird die Ausführung von Garbage Collections mit einer Rate von 1 % simuliert, so ergibt sich für die kleinste Gittergröße keine nennenswerte Veränderung. Für die anderen beiden Gitter fällt der Anstieg flacher aus, insbesondere für die lineare Liste auf dem Gitter mit 49 Knoten (siehe Abbildung 8.2(d)). Eine weitere Erhöhung der



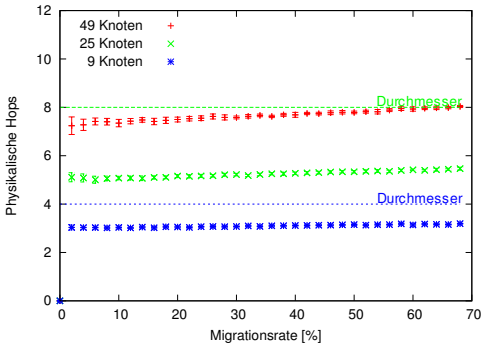
(a) Rot-Schwarz Baum, 0 % GC Rate



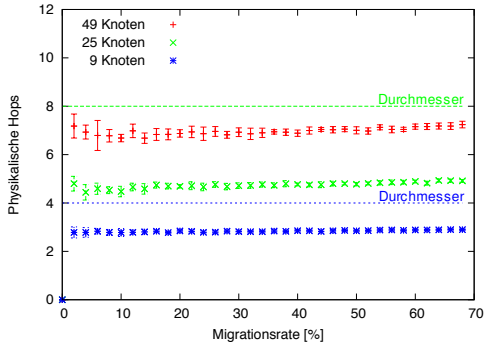
(b) Lineare Liste, 0 % GC Rate



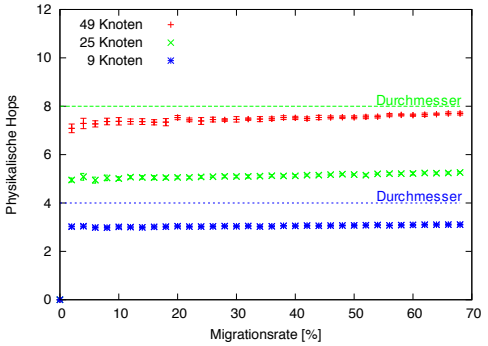
(c) Rot-Schwarz Baum, 1 % GC Rate



(d) Lineare Liste, 1 % GC Rate



(e) Rot-Schwarz Baum, 2 % GC Rate



(f) Lineare Liste, 2 % GC Rate

Abbildung 8.2: Durchschnittliche Anzahl physikalischer Hops für einen GAO-Zugriff

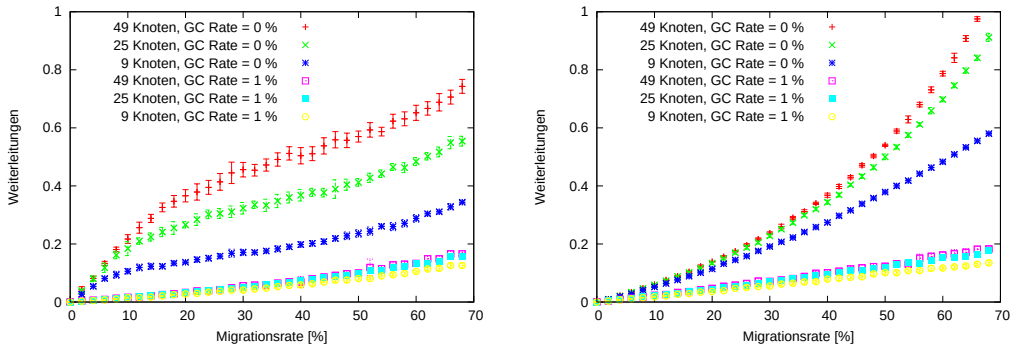
Garbage Collection Rate auf 2 % (siehe Abbildung 8.2(e) und 8.2(f)) und darüber hinaus führt zu keiner weiteren signifikanten Verbesserung bzw. Beschränkung der physikalischen Hops.

Bei Betrachtung der annähernd konstanten Anzahl von physikalischen Hops, bedingt durch die Garbage Collection, ist zu erkennen, dass diese unterhalb des jeweiligen

Durchmessers des Netzwerks liegen (siehe Abbildung 8.2). Dies ist einerseits auf die Aktualisierung der ausgehenden Referenzen durch die Garbage Collection und andererseits auf die Performanz des SSR Protokolls zurückzuführen. Die Verwendung von Source Routen in Kombination mit Referenzen und die damit einhergehende Erkennung von Schleifen bei der Konkatenation von Source Routen nach Migrationen, vermeidet unnötigen Aufwand.

Anzahl Weiterleitungen

Die durchschnittliche Anzahl, wie häufig eine Nachricht beim Zugriff auf ein GAO auf Grund einer Migration weitergeleitet werden muss, wird in Abbildung 8.3 dargestellt. Die Anzahl der Weiterleitungen bezieht sich dabei nicht auf physikalische Hops, sondern auf die Weiterleitungen entlang der GAO-Listenstruktur.



(a) Rot-Schwarz Baum (b) Lineare Liste

Abbildung 8.3: Durchschnittliche Anzahl von Weiterleitungen für einen GAO-Zugriff

Ein Wert von Null zusätzlichen Weiterleitungen stellt keinen Overhead dar und entspricht dem Aufwand des Adressierungsverfahrens, auf dem GCAGC basiert. Dies ist bei einer Migrationsrate von 0 % der Fall. Bedarf es hingegen einer Weiterleitung, so bedeutet dies, dass jeder Zugriff auf ein GAO im Durchschnitt von einem Knoten entlang der zugehörigen Listenstruktur weitergeleitet werden muss.

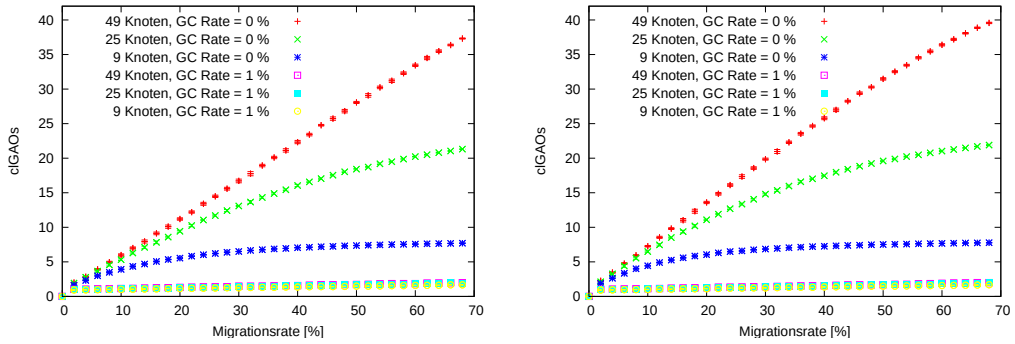
Aus den Simulationsergebnissen ist zu erkennen, dass die Anzahl der Weiterleitungen von der Netzgröße abhängig ist. Die Netzgröße beschränkt die maximale Anzahl der sich im System befindlichen clGAOs und somit auch die Anzahl der maximal möglichen Weiterleitungen (siehe nachfolgenden Abschnitt „Anzahl von clGAOs"). Darüber hinaus beeinflusst die verwendete Datenstruktur die Anzahl der Weiterleitungen. Dies ist auf die unterschiedliche Zugriffscharakteristik und der damit verbundenen Aktualisierung der Referenzen zurückzuführen.

Für die lineare Liste zeigt sich, dass die Anzahl der Weiterleitungen nicht linear, sondern exponentiell ansteigt, falls keine Garbage Collection durchgeführt wird.

Das bedeutet, dass für das Gitter mit 49 Konten, ab einer Migrationsrate von 65 %, durchschnittlich jeder Zugriff von mindestens einem Knoten entlang der zugehörigen Head Referenz weitergeleitet werden muss. In Kombination mit einer Garbage Collection Rate von 1 % wird die Anzahl der Weiterleitungen auf einen linearen Anstieg limitiert und reduziert somit deren Wert signifikant. Eine weitere Erhöhung der Garbage Collection Rate auf 2 % führt zu einer weiteren Abflachung des linearen Anstiegs.

Anzahl von clGAOs

Jede Migration eines efGAOs erzeugt ein neues clGAO und benötigt für die Speicherung der Head Referenz zusätzlichen Speicher. Abbildung 8.4 zeigt für ein GAO die zu erwartende Anzahl gespeicherter clGAOs für Garbage Collection Raten von 0 % und 1 % in Abhängigkeit der Migrationsrate.



(a) Rot-Scharz Baum (b) Lineare Liste

Abbildung 8.4: Durchschnittliche Anzahl von clGAOs für ein GAO

Für das Gitter mit 9 Knoten zeigt sich für eine Garbage Collection Rate von 0 %, dass sich die Anzahl der clGAOs an den Wert 8 annähern. Jeder Knoten des Netzwerks speichert somit für ein GAO entweder ein clGAO oder das efGAO. Daraus folgt, dass im schlechtesten Fall der zusätzliche Speicherbedarf pro Knoten von der Gesamtanzahl der GAOs abhängig ist. Es sei an dieser Stelle angemerkt, dass dieses Verhalten dadurch begründet ist, dass die Migration eines efGAO ein bereits vorhandenes clGAO am neuen Home Node überschreibt bzw. aktualisiert.

Diese Betrachtungsweise kann für die beiden größeren Gitterstrukturen analog übertragen werden. Jedoch lässt sich nicht eindeutig die Beschränkung, sondern nur der Trend zur Beschränkung, auf 24 bzw. 48 clGAOs aus den beiden Diagrammen erkennen. Zur eindeutigen Erkennung der Beschränkung wäre es notwendig, die Simulationen mit Migrationsraten größer als 70 % zu berechnen. Leider war dies aus zeitlichen Gründen im Rahmen dieser Diplomarbeit nicht mehr möglich.

Weitere Simulationen

Über die oben beschriebenen Evaluationskriterien hinausgehend, wurden noch weitere Simulationsläufe mit geänderten Parametern und Evaluationskriterien betrachtet. Zum einen wurden für die Evaluationskriterien nicht der Durchschnitt über alle GAOs, sondern nur für das GAO mit der maximalen Anzahl von Zugriffen (maxGAO) gebildet. Die daraus resultierenden Simulationsergebnisse für Gitter mit 9 und 25 Knoten sind im Appendix in den Abbildungen A.1, A.2 und A.3 dargestellt. Im Allgemeinen lässt sich für Zugriffe auf das maxGAO kein gravierender Unterschied im Vergleich zu den obigen Evaluationsergebnissen feststellen. Bei genauerer Betrachtung ist zu erkennen, dass die Rot-Schwarz Baum Implementierung bezüglich der durchschnittlichen Anzahl von Weiterleitungen für den Zugriff auf das maxGAO einen Unterschied zu oben aufweist. Dies ist durch die Zugriffscharakteristik des maxGAOs bedingt.

Zum anderen wurde die Migrationsstrategie dahingehend abgeändert, dass für Migrationen das Migrationsziel nicht beliebig innerhalb des Gitters wählbar ist. Genauer gesagt wurde die Länge der Migrationsroute auf einen Hop begrenzt und für Gitter mit 9 und 25 Knoten die Simulationen berechnet. Dies entspricht somit der stärksten Beschränkung. Die daraus resultierenden Abbildung A.4 bis A.6 sind im Appendix zu finden.

Die Beschränkung der Migrationsroute auf eine Länge von einem Hop beeinflusst die Evaluationskriterien. Für das Gitter mit 9 Knoten ist sowohl für die Implementierung des Rot-Schwarz Baums, als auch für die lineare Liste fast keine Veränderung zu erkennen. Deutlicher fällt der Unterschied für das Gitter mit 25 Knoten aus. Bei Migrationsraten unterhalb von 10 % steigt die durchschnittliche Anzahl von physikalischen Hops im Vergleich zu den Diagrammen aus Abbildung 8.2 erst an, bis sie im weiteren Verlauf annähernd konstant zwischen 4 und 5 Hops verweilt (abgesehen von Abbildung A.4(b)). Dies ist darauf zurückzuführen, dass efGAOs ihre Lokalität zueinander behalten, solange die Migrationsrate niedrig genug ist und im Gegensatz zu unbeschränkten Migrationsrouten nicht willkürlich im Gitter verteilt werden. Darüber hinaus erfolgt der Anstieg für die Anzahl von clGAOs und Weiterleitungen flacher. Dies ist vor allem für die Anzahl der clGAOs der Rot-Scharz Baum Implementierung zu erkennen (siehe Abbildung A.6(a)). Weitere Simulationen zeigen, dass eine Verlängerung der Migrationsroute von einem auf zwei oder mehr Hops für das Gitter mit 9 Knoten nicht mehr sinnvoll ist, da somit fast jeder Knoten des Netzwerks erreicht werden kann und dies der unbeschränkten Migrationsroute entspricht. Darauf folgt, dass die Länge der Migrationsroute immer unterhalb des Durchmessers des Netzwerks liegen sollte, damit eine Verbesserung für die Evaluationskriterien eintritt.

Lesende Zugriffe auf GAOs und die damit verbundene Aktualisierung der ausgehenden Referenzen beeinflussen die Performanz von GCAGC. Abbildungen A.7, A.8 und A.9 stellen die Evaluationskriterien für Gitter mit 9 und 25 Knoten dar, die

sich ohne Aktualisierung der ausgehenden Referenzen bei einer Garbage Collection Rate von 0 % ergeben. Abgesehen von der Anzahl der clGAOs, die fast identisch zu den obigen Ergebnissen ist, werden die beiden anderen Evaluationskriterien negativ beeinflusst. Die Anzahl der physikalischen Hops steigt soweit an, so dass diese überhalb des jeweiligen Netzwerkdurchmessers liegt. Der Grund dafür ist die hohe Anzahl von Weiterleitungen, die sich aus der fehlenden Aktualisierung der Referenzen ergibt. Darüber hinaus ist zu erkennen, dass die Anzahl der Weiterleitungen sowohl für die Implementierung des Rot-Schwarz Baums, als auch für die verkettete Liste gegen denselben konstanten Wert streben. Dies steht im Widerspruch zu den obigen Ergebnissen, die keine Beschränkung aufzeigen. Jedoch liegen diese Werte deutlich unterhalb der Werte aus Abbildung A.8. Es bleibt die Frage offen, ob die obigen Simulationsergebnisse ebenfalls beschränkt sind, wenn die Migrationsrate auf annähernd 100 % erhöht wird. Aus zeitlichen Gründen war dies nicht mehr möglich. Es sei an dieser Stelle angemerkt, dass alle anderen Simulationen, die im Rahmen dieser Arbeit durchgeführt wurden, immer mit der Aktualisierung der Referenzen berechnet wurden.

Zusammenfassend lassen sich die folgenden Aussagen treffen. Im schlechtesten Fall, wenn keine Garbage Collections durchgeführt werden, ist für kleine Gitter bzw. für einen kleinen Migrationsradius die Anzahl der physikalischen Hops und der clGAOs beschränkt. Die Anzahl der physikalischen Hops wird zusätzlich für geringe Migrationsraten reduziert, falls die Längen der Migrationsrouten unterhalb des Netzwerkdurchmessers liegen. Darüber hinaus benötigt ein Zugriff auf ein GAO durchschnittlich nur maximal eine zusätzliche Weiterleitung für Migrationsraten unterhalb von 65 %. Jedoch ist die konkrete Anzahl der Weiterleitungen für einen Zugriff abhängig von dem ausgeführten Programm und dessen Zugriffscharakteristik auf das GAO. Wird hingegen eine Garbage Collection verwendet, so resultiert dies für die Evaluationskriterien in einer signifikanten Abflachung des Overheads.

9 Zusammenfassung und Ausblick

Im Rahmen dieser Diplomarbeit wurde die Problematik der Adressierung von Objekten, genauer gesagt von GAOs, die auf Knoten eines Netzwerks verteilt vorgehalten werden, betrachtet. Der Fokus der Betrachtung richtete sich auf die AmbiComp Virtual Machine (siehe Kapitel 3), einer verteilten virtuellen Maschine für Java. Die Zielplattform der ACVM sind eingebettete Systeme, die zum Beispiel in Verbindung mit Sensorknoten verwendet werden. Eine Vielzahl von Knoten bilden ein Netzwerk, auf dem die ACVM aufsetzt. Die Erreichbarkeit der Knoten wird durch das Scalable Source Routing Protokoll (siehe Kapitel 4) gewährleistet. Dieses Protokoll wurde um ein Adressierungsverfahren erweitert, dass die Basis für eine verteilte Ausführung von Java Anwendungen durch die ACVM bildet.

Solange GAOs auf den Knoten verweilen, wird ein referenziertes GAO mittels der gespeicherten Zuordnung zu dessen Home Node adressiert (siehe Kapitel 5). In diesem Fall wird auf den SSR Routing Algorithmus zurückgegriffen. Jedoch involviert das Routing im schlechtesten Fall Teile des Netzwerks, die in keiner Beziehung mit der Programmausführung durch die ACVM stehen. Zur Vermeidung des Routings und zur Verbesserung der Performanz werden zusätzlich zu den Zuordnungen die entsprechenden Source Routen gespeichert. Somit entfällt für Knoten die Ausführung des SSR Routing Algorithmus und es muss letztendlich die Nachricht nur an den nächsten Hop weitergeleitet werden.

Aus Gründen der Lastverteilung, Parallelisierung und Redundanz ist es sinnvoll die Zuordnungen von GAOs zu Home Nodes aufzuheben und Migrationen zu erlauben. Jedoch geht mit dem einher, dass referenzierende GAOs inkonsistente Informationen bezüglich der Zuordnungen und der Source Routen der migrierten GAOs besitzen. Kapitel 6 beschreibt vier unterschiedliche Verfahren, die diese Problematik umgehen. Jedoch erfordert dies in Abhängigkeit des gewählten Verfahrens einen erheblichen Mehraufwand bezüglich des verwendeten Speichers und der Anzahl ausgetauschter Nachrichten.

Das GAO Cloning And Garbage Collecting Verfahren (siehe Kapitel 6.2.4) wurde ausgewählt und zur späteren Evaluierung implementiert (siehe Kapitel 7). GCAGC basiert auf der Verwendung von Weiterleitungen, die nach einer Migration eines GAOs auf den Zielknoten der Migration verweisen. Nicht mehr benötigte Weiterleitungen werden durch die verteilt ausgeführte Garbage Collection der ACVM entfernt. Darüber hinaus werden während der Ausführung der Garbage Collection alle Referenzen aktualisiert. Solange keine weiteren Migrationen stattfinden, werden nach dem Abschluss der Garbage Collection keine Weiterleitungen mehr benötigt,

Die Evaluation (siehe Kapitel 8) zeigt, dass die Leistungsfähigkeit des GCAGC Verfahrens durch die Garbage Collection Rate, der Migrationsrate und den Migrationsradius zu beeinflussen ist. Somit kann eine dynamische Anpassung von GCAGC an die Bedingungen des Netzwerks erfolgen.

Über diese Diplomarbeit hinausgehend ist es von Interesse, weitere Aspekte zur Steigerung der Performanz von GCAGC zu betrachten. Dazu zählt zum einen der Ansatz „Schreiben durch Migration“. Die Operationen für den schreibenden Zugriff auf ein GAO und die Migration desselben GAOs werden zusammengefasst und mit einer einzigen Schreib- und Migrationsnachricht auf dem Netzwerk ausgeführt. Zum anderen kann die GAO-Listenstruktur nach Lesezugriffen mit Hilfe der gesendeten Antwortnachrichten aktualisiert werden. Das bedeutet, dass die Antwortnachricht entlang der Listenstruktur zurückgesendet wird und jeder Knoten, der eine Weiterleitung (= clGAO) speichert, die Head Referenz auf das Ende der Listenstruktur aktualisiert. Dies entspricht einer Pfadkompression der Head Referenzen und erfordert die entgegengesetzte Traversierung der Listenstruktur. Darüber hinaus muss jedes Element der Listenstruktur nicht nur die Antwortnachricht weiterleiten, sondern auch die Head Referenz aktualisieren. Es stellt sich an dieser Stelle für beide Erweiterungen die Frage, ob der einhergehende Overhead in Relation mit der zu erwartenden Performanzsteigerung steht.

Ein Problem, dass in Verbindung mit dem Adressierungsverfahren auftreten kann ist die Tatsache, dass für Referenzen potentiell eine große Anzahl von Source Routen im Route Cache gespeichert werden müssen. Um den Verlust von Source Routen und die damit verbundene Verwendung des SSR Routing Algorithmus zu vermeiden, werden Source Routen an andere Knoten delegiert. Die Auswahl des Delegationsknotens basiert auf dem SSR Routing Algorithmus. Jedoch muss beachtet werden, dass der Delegationsknoten reproduzierbar ist. Delegierte Source Routen können somit aus dem Route Cache entfernt werden.

Abgesehen von der weiteren Optimierung von GCAGC muss die ACVM um eine Migrationsstrategie erweitert werden. Die Migrationsstrategie legt fest, unter welchen Umständen ein lokales Objekt oder GAO migriert wird. Zum Beispiel können Migrationen ab einer bestimmten Speicherbelegung eines Knotens initiiert werden. Ebenso kann ein rechenintensiver Thread, der parallel zu anderen Threads auf demselben Knoten existiert, eine Migration auslösen. Des weiteren bestimmt die Migrationsstrategie den Zielknoten und die Migrationsroute. Hierbei ergibt sich auf Grund des verteilten Systems und der Ermangelung eines globalen Wissens die Fragestellung, welcher Knoten noch genügend Ressourcen zur Verfügung stellt und somit als Migrationsziel in Frage kommt.

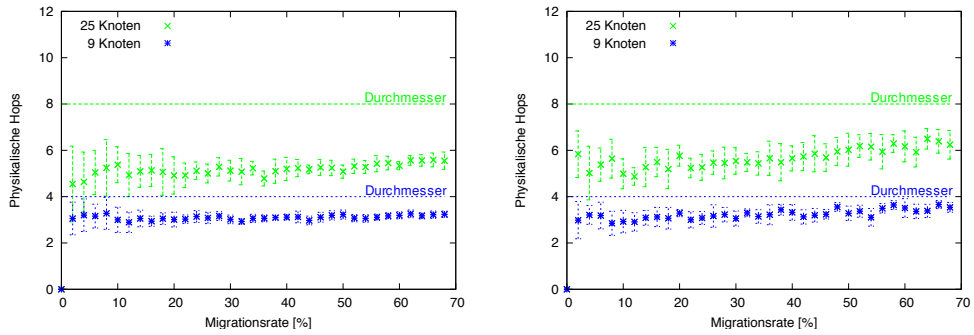
Darüber hinaus bedarf es einer verteilten Garbage Collection und eines Synchronisationsmechanismus. Die Fertigstellung von allen Komponenten und deren Integration in die ACVM resultiert schließlich in einer verteilt lauffähigen virtuellen Maschine für Java Anwendungen.

A Appendix

A.1 Weitere Simulationen

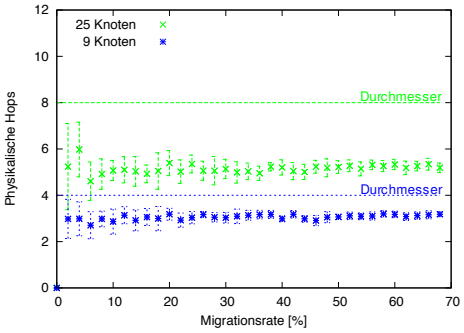
Nachfolgende Abbildungen A.1, A.2 und A.3 stellen die Evaluationskriterien bezüglich des GAOs mit der maximalen Anzahl von Zugriffen (maxGAO) dar. Für diese Diagramme liegen nur die Simulationsergebnisse eines einzelnen GAOs zu Grunde. Daraus resultiert der höhere statistische Fehler, da um den Faktor 125 bzw. 126 (= Anzahl Objekte der Datenstrukturen) weniger Daten zur Verfügung stehen. Des weiteren beziehen sich die Abbildungen A.4, A.5 und A.6 auf die Evaluationskriterien, die sich aus der Beschränkung der Migrationsroute auf einen Hop ergeben. Erfolgt bei lesenden Zugriffen keine Aktualisierung der ausgehenden Referenzen, so resultiert dies in den Simulationsergebnissen, die in den Abbildungen A.7, A.8 und A.9 dargestellt werden.

Alle Simulationsergebnisse wurden für Gitter mit jeweils 9 und 25 Knoten berechnet.

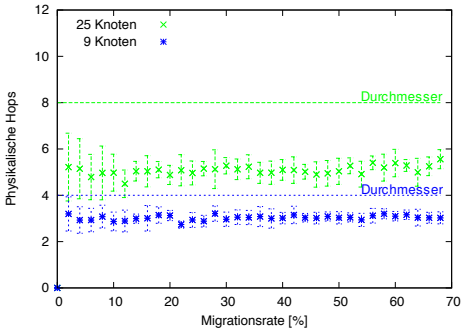


(a) Rot-Schwarz Baum, 0 % GC Rate (b) Lineare Liste, 0 % GC Rate

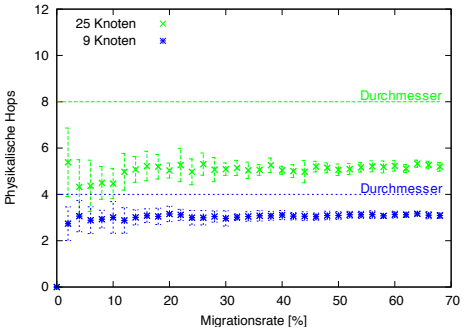
Abbildung A.1: Durchschnittliche Anzahl physikalischer Hops für einen Zugriff auf das maxGAO



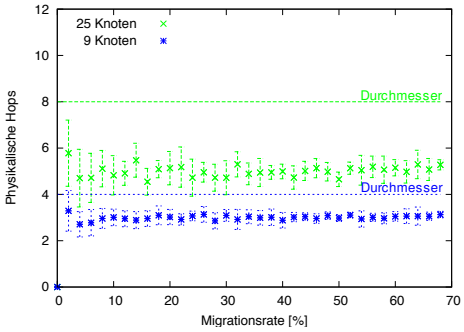
(c) Rot-Schwarz Baum, 1 % GC Rate



(d) Lineare Liste, 1 % GC Rate

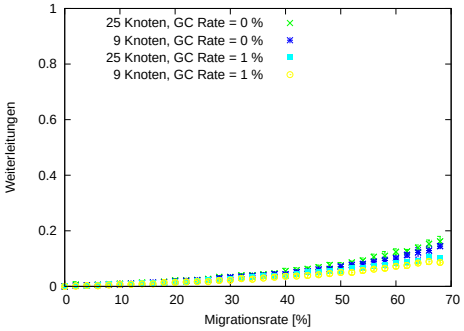


(e) Rot-Schwarz Baum, 2 % GC Rate

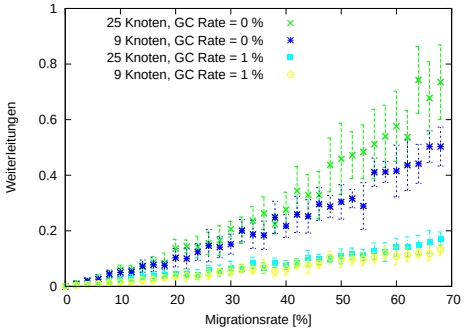


(f) Lineare Liste, 2 % GC Rate

Abbildung A.1: Durchschnittliche Anzahl physikalischer Hops für einen Zugriff auf das maxGAO

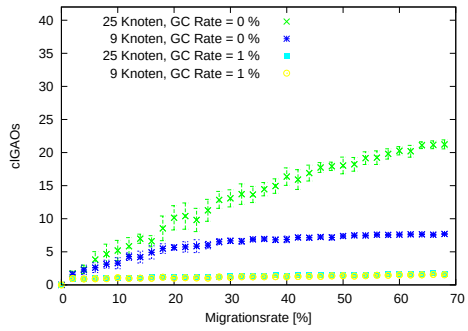


(a) Rot-Schwarz Baum

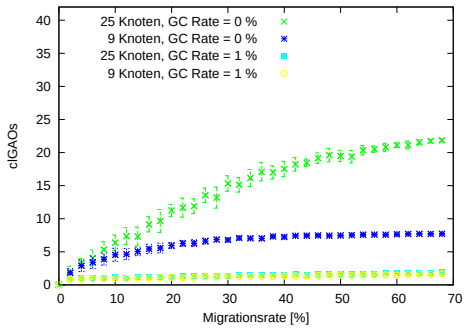


(b) Lineare Liste

Abbildung A.2: Durchschnittliche Anzahl von Weiterleitungen für einen Zugriff auf das maxGAO

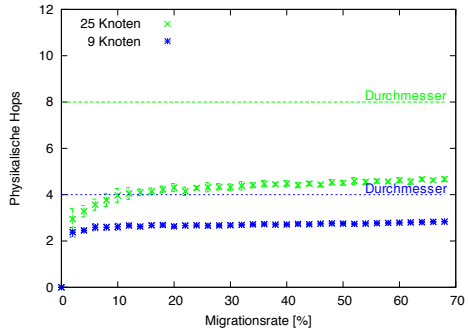


(a) Rot-Scharz Baum

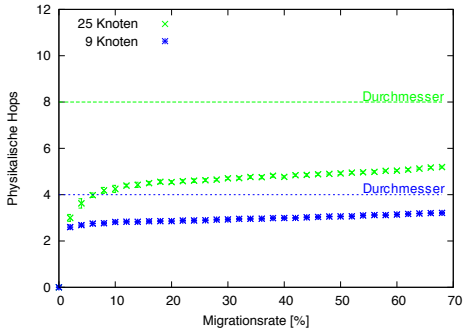


(b) Lineare Liste

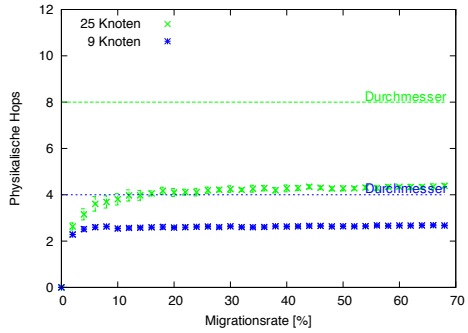
Abbildung A.3: Durchschnittliche Anzahl von clGAOs für das maxGAO



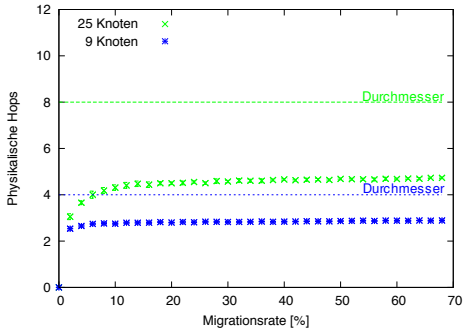
(a) Rot-Schwarz Baum, 0 % GC Rate



(b) Lineare Liste, 0 % GC Rate

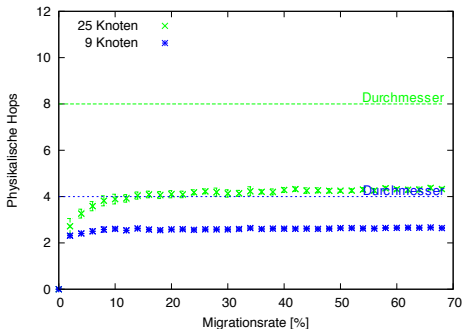


(c) Rot-Schwarz Baum, 1 % GC Rate

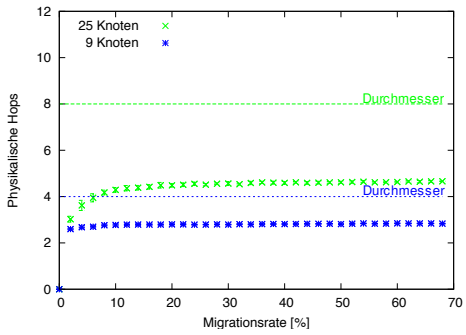


(d) Lineare Liste, 1 % GC Rate

Abbildung A.4: Durchschnittliche Anzahl physikalischer Hops für einen GAO-Zugriff, Migrationsroute auf einen Hop beschränkt

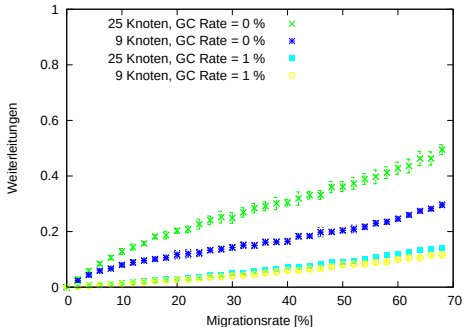


(e) Rot-Schwarz Baum, 2 % GC Rate

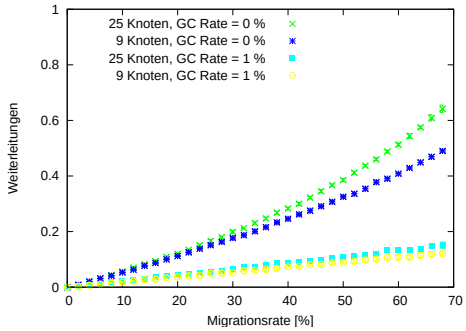


(f) Lineare Liste, 2 % GC Rate

Abbildung A.4: Durchschnittliche Anzahl physikalischer Hops für einen GAO-Zugriff, Migrationsroute auf einen Hop beschränkt

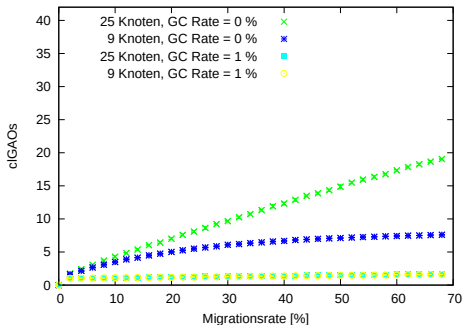


(a) Rot-Schwarz Baum

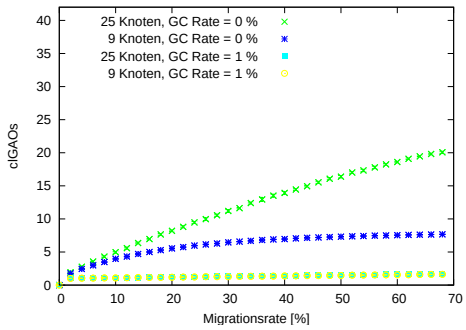


(b) Lineare Liste

Abbildung A.5: Durchschnittliche Anzahl von Weiterleitungen für einen GAO-Zugriff, Migrationsroute auf einen Hop beschränkt

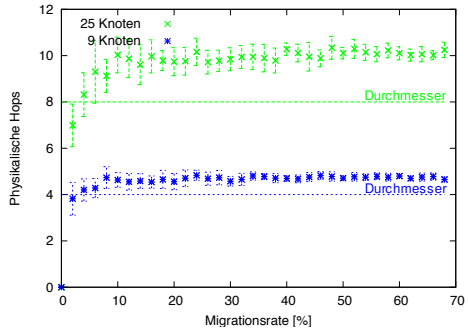


(a) Rot-Scharz Baum

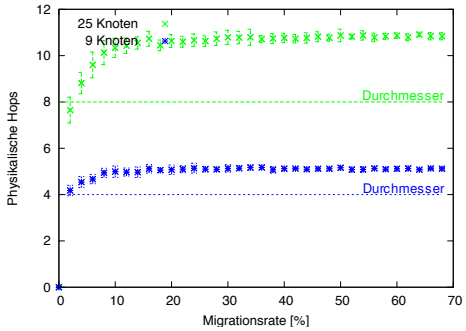


(b) Lineare Liste

Abbildung A.6: Durchschnittliche Anzahl von clGAOs für ein GAO, Migrationsroute auf einen Hop beschränkt

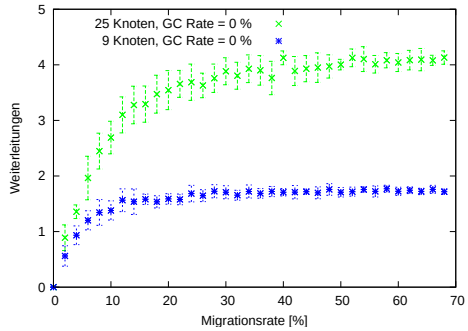


(a) Rot-Schwarz Baum, 0 % GC Rate

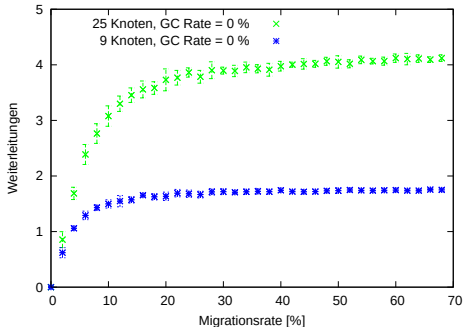


(b) Lineare Liste, 0 % GC Rate

Abbildung A.7: Durchschnittliche Anzahl physikalischer Hops für einen GAO-Zugriff, keine Aktualisierung ausgehender Referenzen

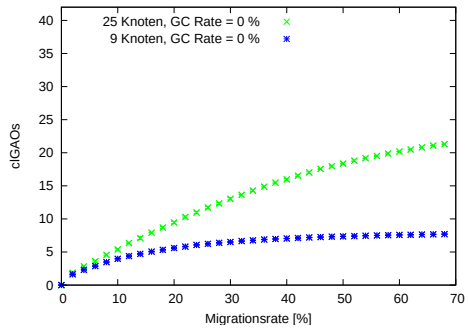


(a) Rot-Schwarz Baum

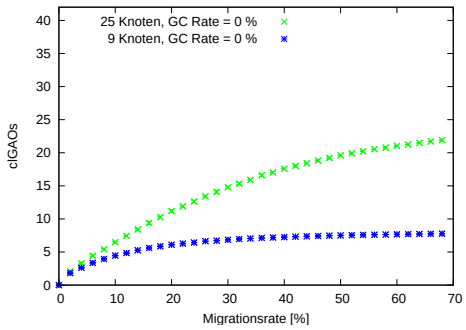


(b) Lineare Liste

Abbildung A.8: Durchschnittliche Anzahl von Weiterleitungen für einen GAO-Zugriff, keine Aktualisierung ausgehender Referenzen



(a) Rot-Scharz Baum



(b) Lineare Liste

Abbildung A.9: Durchschnittliche Anzahl von clGAOs für ein GAO, keine Aktualisierung ausgehender Referenzen

A.2 Details zur Simulationsausführung

Die Durchführung der Evaluation basiert auf der Ausführung von vielen Simulationen. Für die Simulationen werden die in Kapitel 8.2 beschriebenen Parameter mit Hilfe eine Perl Scripts (siehe [6]) variiert, die zugehörigen Konfigurationsdateien erzeugt und die Simulationen in Verbindung mit dem OMNet++ Simulator gestartet. Im weiteren Verlauf wird darauf eingegangen, wie der Quellcode kompiliert wird und anschließend die Simulationen gestartet werden.

Zunächst müssen alle notwendigen Ressourcen aus dem SVN-Repository lokal gespeichert werden.

```
$ cd /home/user/
$ mkdir workspace
$ cd workspace/
$ svn checkout https://moon.ira.uka.de/subversion/main/ \
                        branches/projects/ssr-gao-addressing/ ./
```

Durch den `svn checkout` werden im Verzeichnis `/home/user/workspace/` drei Unterverzeichnisse angelegt. Das Unterverzeichnis `ssr-core/` beinhaltet die Implementierung des SSR Protokolls, die um das GCAGC Verfahren und die Komponenten zur Simulation der ACVM erweitert wurde. Darüber hinaus existiert das Verzeichnis `easyOmMac/`, das den OMNet++ Simulator und die zugehörigen Komponenten zur Anbindung der SSR Implementierung enthält. Im dritten Verzeichnis `evaluation/` befinden sich alle Ressourcen, die zur Evaluation notwendig sind.

Bevor die Simulationen gestartet werden können, muss sowohl für das SSR Protokoll, als auch den OMNet++ Simulator der Quellcode kompiliert werden.

```
$ cd ssr-core/
$ make -C debug-linux
$ cd ../easyOmMac/
$ ./bootstrap.sh debug-linux cmdenv
$ cd ..
```

Nach erfolgreichem Abschluss der Kompilierung des Quellcodes ist es erforderlich, das Script `runEvaluation.pl` an den lokalen Installationspfad anzupassen. Dazu muss die Variable `$BASEDIR` entsprechend gesetzt werden. Abbildung A.10 zeigt das Script in gekürzter Form, das die mit Bezug auf die obige Ablaufbeschreibung geändert Variable enthält. Das Script benötigt als Aufrufparameter den Pfad zu einer Datei. Diese Datei muss die Debug-Ausgabe der ACVM beinhalten, die bei der Ausführung eines Testprogramms durch die ACVM entsteht. Während der Laufzeit des Scripts wird diese Datei geparst und daraus die Trace Files für den jeweiligen Simulationslauf generiert. Schließlich werden die Simulationen mit Hilfe des Scripts zum Beispiel für die einfach verkettete Liste folgendermaßen gestartet:

```
$ cd evaluation/
$ perl runEvaluation.pl acvmTestRunsOutput/LinearList100Test.txt
```

Während der Ausführung des Scripts wird das Verzeichnis `evaluation/results` erstellt. Dieses Verzeichnis enthält nach Beendigung der Simulationen für den jeweiligen Aufrufparameter ein Unterverzeichnis. In diesem Unterverzeichnis sind einerseits die Konfigurationsdateien, die für die Ausführung der Simulationen notwendig sind, und andererseits die Ergebnisse der Simulationen abgespeichert. Die Trennung der Dateien erfolgt über die beiden Verzeichnisse `inputFiles/` und `outputFiles/`.

```
#!/usr/bin/perl

########################################################################
##
## Update the following variable for your local configuration
##
$BASEDIR = "/home/user/workspace";
########################################################################

$OMNETDIR = "$BASEDIR/easyOmMac";
$OMNET = "$OMNETDIR/easyOmMac";
$SCENARIODIR = "$BASEDIR/easyOmMac/scenarios";
$EVALDIR = "$BASEDIR/evaluation";
$RESULTDIR = "$BASEDIR/evaluation/results";
$BLOBPARSER = "$BASEDIR/evaluation/blobOutputParser.pl";

$LIBSSRPATH = "$BASEDIR/ssr-core/debug-linux";

#
# check for argument (= acvmTestRunsOutput)
#
if ($#ARGV == 0)
{
  #
  # check whether given file exists
  # and create necessary directories
  #
  {
    ...
  } # END

  # get filename without leading path
  $filename = substr($ARGV[0], rindex($ARGV[0], "/")+1);
  $statfilename = substr($filename, 0, -4) . ".stat";

  $outputdir = $RESULTDIR . "/" . $filename;
  $inputFileName = $outputdir . "/" . $filename;
  $inputFilesDir = $outputdir . "/inputFiles";
  $outputFilesDir = $outputdir . "/outputFiles";
```

```
  #
  # create the $outputDir and
  # create the subdirectories
  # $inputFilesDir and $outputFilesDir
  #
  {
    ...
  } # END

  # copy input file to results dir
  `cp $ARGV[0] $outputdir`;

  # create ned files
  if (!createNedFiles($inputFilesDir, $inputFileName, $outputFilesDir))
  {
    print "error: creating ned files\n";

    exit;
  } # END if

  # run simulations
  runSimulations($inputFilesDir, $outputFilesDir);

  print "runEvaluation finished\n";

  exit;

} # END if
else
{
  # no file argument given
  print "error: no file to run given\n";

  exit;

} # END if else
```

```
# subroutine createNedFiles uses 3 parameters
# $_[0] => directory where the ned files should be created
# $_[1] => the name of the file which is the source for the ned files
# $_[2] => full path name to file which should be parsed in
           order to create ned files

sub createNedFiles
{

  $filesDir = $_[0];
  $fileToParse = $_[1];
  $outputFilesDir = $_[2];

  #
  # check whether $fileToParse,
  # $filesDir and $outputFilesDir exists
  #
  {
    ...
  } # END

  # copy input file once more
  `cp $fileToParse $filesDir`;

  $fileToParse = $filesDir . "/" .
                 substr($fileToParse, rindex($fileToParse, "/")+1);


  ################################################
  # for every input parameter create ned file ... #
  ################################################

  # garbage collections
  @GCs = (0.0, 0.01, 0.02);

  # grid sizes
  @GS = (9, 25, 45);

  for ($size = 0; $size < @GS; $size++)
  {
    # set grid size
    $gridSize = $GS[$size];
```

```
    for ($run = 0; $run <= 9; $run++)
    {
      for ($migRate = 0.0; $migRate <= 0.7; $migRate += 0.02)
      {
        for ($gcRate = 0; $gcRate < @GCs; $gcRate++)
        {
          # create a name for files reflecting the parameters
          $fileFix = "m=" . $migRate . "g=" . $GCs[$gcRate] .
                     "s=" . $gridSize . "r=" . $run;

          # rename input file in order to parse it
          `mv $fileToParse $filesDir"/"$fileFix`;

          # update input file name
          $fileToParse = $filesDir . "/" . $fileFix;
          $statFileToParse = $outputFilesDir . "/" .
                             $fileFix . ".stat";

          # parse the input file for the given parameters
          print "creating $fileToParse\n";
          print `perl $BLOBPARSER $fileToParse
                      -m $migRate -g $GCs[$gcRate] -s $gridSize`;

        } # END gcRate
      } # END migRate
    } # END run
  } # END size

  # finally delete copied input file
  `rm $fileToParse`;

  return 1;

} # END createNedFiles
```

```
# subroutine runSimulations uses 2 parameters
# $_[0] => directory where the ned files are located
# $_[1] => directory where the simulation results should be copied to

sub runSimulations
{
  $inputFileDir = $_[0];
  $outputFileDir = $_[1];

  #
  # check whether $inputFileDir and
  # $outputFileDir exists
  #
  {
    ...
  } # END

  $numSims = 0;
  $numSimAborts = 0;

  # look for *.ned files within the input file directory
  $tmp = $inputFileDir . "/*.ned";
  @nedFiles = `ls $tmp`;

  # change to omnet dir
  chdir $OMNETDIR;

  # set environment variable
  $ENV{'LD_LIBRARY_PATH'} = $LIBSSRPATH;

  ################################################
  # for every ned file run simulation ...        #
  ################################################

  while (@nedFiles != 0)
  {
    $currentFile = pop(@nedFiles);
    chomp($currentFile);

    $fileName = substr($currentFile, rindex($currentFile, "/")+1);
    $statFileName = substr($fileName, 0, -4) . ".stat";
```

```
    print "running simulation with $currentFile\n";

    # copy ned file to simulation directory
    `cp $currentFile $SCENARIODIR`;

    # run simulation
    if (`$OMNET -f $SCENARIODIR/$fileName 2>&1 1>/dev/null`)
    {
      # error durring simulation
      print "error durring simulation run\n";

      # count errors
      $numSimAborts++;

    } # END if
    else
    {
      # successfully executed simulation
      $numSims++;

    } # END if else

    # copy statfile from input directory to output directory
    if (-e $inputFileDir . "/" . $statFileName)
    {
      `mv $inputFileDir/$statFileName $outputFileDir`;
    } # END if

    # delete file
    `rm $SCENARIODIR/$fileName`;

  } # END while

  print "finished running $numSims simulations " .
        "with $numSimAborts errors\n";

  return 1;

} # END runSimulations
```

Abbildung A.10: Perl Script zur automatisierten Ausführung der Simulationen

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] E.H.L. Aarts, J.M. Rabaey, and W. Weber. *Ambient intelligence*. Springer, 2005.

[2] Y. Aridor, M. Factor, and A. Teperman. cJVM: a Single System Image of a JVM on a Cluster. In *International Conference on Parallel Processing*, volume 411, 1999.

[3] H. Balakrishnan, M.F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking up data in P2P systems. *Communications of the ACM*, 46(2):43–48, 2003.

[4] B. Bruegge and A.H. Dutoit. *Object-Oriented Software Engineering: Using UML, Patterns and Java*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 2003.

[5] R. Buyya, T. Cortes, and H. Jin. Single System Image. *International Journal of High Performance Computing Applications*, 15(2):124–135, 2001.

[6] T. Christiansen and N. Torkington. *Perl Cookbook*. O'Reilly, 2nd edition, 2003.

[7] M. Friedewald, O.D. Costa, Y. Punie, P. Alahuhta, and S. Heinonen. Perspectives of ambient intelligence in the home environment. *Telematics and Informatics*, 22(3):221–238, 2005.

[8] T. Fuhrmann. AmbiComp. `http://ambicomp.net/`, Stand: 21.10.2008.

[9] T. Fuhrmann. Scalable routing for networked sensors and actuators. *Proceedings of the Second Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, pages 240–251, 2005.

[10] T. Fuhrmann. Scalable Routing in Sensor Actuator Networks with Churn. *Sensor and Ad Hoc Communications and Networks, 2006. SECON'06. 2006 3rd Annual IEEE Communications Society on*, 1, 2006.

[11] T. Fuhrmann. Performance of Scalable Source Routing in Hybrid MANETs. *Fourth Annual Conference on Wireless on Demand Network Systems and Services, WONS'07*, pages 122–129, 2007.

[12] T. Fuhrmann, P. Di, K. Kutzner, and C. Cramer. Pushing Chord into the Underlay: Scalable Routing for Hybrid MANETs. Technical report, Fakultät für Informatik, Universität Karlsruhe (TH), Deutschland, 21 Juni 2006.

[13] H. Gränicher. *Messung beendet-was nun?* Hochschulverl. AG an der ETH Zürich, 1994.

[14] L.J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Foundations of Computer Science, 1978., 19th Annual Symposium on*, pages 8–21, 1978.

[15] B. Haumacher, T. Moschny, J. Reuter, and W. F. Tichy. Transparent distributed threads for java. In *Proceedings of the 5th International Workshop on Java for Parallel and Distributed Computing in conjunction with the International Parallel and Distributed Processing Symposium (IPDPS 2003)*, April 2003.

[16] D.B. Johnson and D.A. Maltz. Dynamic source routing in ad hoc wireless networks. *Mobile Computing*, 353(153-181):152, 1996.

[17] A. Kiening. Accessing remote objects in a distributed embedded Java VM, 15 May 2008.

[18] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.

[19] H. Liu, T. Roeder, K. Walsh, R. Barr, and E.G. Sirer. Design and implementation of a single system image operating system for ad hoc networks. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 149–162. ACM Press New York, NY, USA, 2005.

[20] S. Microsystems. Java Remote Method Invocation Specification. 1997.

[21] S. Microsystems. RPC: Remote Procedure Call Protocol Specification. 2004.

[22] S. Oualline. *Practical C++ Programming*. O'Reilly, 2003.

[23] C.E. Perkins and E. Royer. Ad hoc On-demand Distance Vector (AODV). In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, 1999.

[24] M. Philippsen and M. Zenger. JavaParty - Transparent Remote Objects in Java. *Concurrency Practice and Experience*, 9(11):1225–1242, 1997.

[25] D. Plainfosse and M. Shapiro. A Survey of Distributed Garbage Collection Techniques. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 211–211, 1995.

[26] G. Saake and K.U. Sattler. *Algorithmen und Datenstrukturen*. dpunkt-Verlag, 2006. 3., überarbeitete Auflage.

[27] B. Saballus, J. Eickhold, and T. Fuhrmann. Towards a Distributed Java VM in Sensor Networks using Scalable Source Routing. In *6. Fachgespraech Sensornetzwerke der GI/ITG Fachgruppe "Kommunikation und Verteilte Systeme"*, pages 47–50, Aachen, Germany, 2007.

[28] B. Saballus, J. Eickhold, and T. Fuhrmann. Global Accessible Objects (GAOs) in the Ambicomp Distributed Java Virtual Machine. In *Proceedings of the Second International Conference on Sensor Technologies and Applications (SENSORCOMM 2008)*, Cap Esterel, France, August 25 – 31, 2008. IEEE Computer Society.

[29] N. Shadbolt. From the Editor in Chief: Ambient Intelligence. *IEEE Intelligent Systems*, 18(4):2–3, 2003.

[30] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[31] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *Proceedings of the 2001 SIGCOMM conference*, 31(4):149–160, 2001.

[32] T.Fuhrmann. Ambicomp Übersicht. `http://i30www.ira.uka.de/p2p/ambicomp/downloads/Datenblatt_AmbiComp.pdf`, Stand: 13.11.2008.

[33] A. Varga et al. The OMNeT++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference (ESM'2001)*, 2001.

[34] P.R. Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*, volume 637. Springer, 1992.

[35] W. Zhu, W. Fang, C.L. Wang, and F.C.M. Lau. A New Transparent Java Thread Migration System Using Just-in-Time Recompilation. In *Proceedings of The 16th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 766–771.

[36] W. Zhu, C.L. Wang, and F.C.M. Lau. JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support. In *IEEE Fourth International Conference on Cluster Computing*, 2002.