Mobile Code

Georg Lukas
Seminar Ubiquitous Computing
Universität Magdeburg

2003-11-16

Zusammenfassung

Auf dem Markt existieren zahlreiche Werkzeuge zur Entwicklung mobilen Codes, die zum Teil unterschiedliche Anforderungen erfüllen und nicht für jedes Szenario alle benötigten Funktionen bieten. Ziel dieser Ausarbeitung ist es, eine Klassifizierung dieser Werkzeuge vorzustellen, und an praktischen Beispielen zu zeigen, wie man diese Klassifizierung anwendet. Insbesondere wird auch auf Paradigmen, Sprachen und Sicherheitsaspekte eingegangen.

Inhaltsverzeichnis

1 Einführung

In den letzten Jahren gab es einen starken Schub in der Entwicklung von Computernetzen und daran gebundener Geräte. Zu nennen sind neben Netzwerkdruckern, DSL-Modems und WLan-AccessPoints auch PDAs, Handys und bald auch zentral gesteuerte Haushaltsgeräte, die mit immer mehr Speicher und schnelleren Prozessoren ausgestattet werden.

Dieser Entwicklungsschub wurde jedoch nicht von einer vergleichbaren wissenschaftlichen Entwicklung im Bereich des Softwaredesigns für vernetzte Systeme begleitet, was zum einen dazu führte, dass die verwendete Terminologie nicht einheitlich ist oder sich überlagert, zum anderen aber, dass die eingesetzte Software nicht optimal an die Verbreitung und die neuen Fähigkeiten der Hardware angepasst ist.

Der Schwerpunkt dieses Artikels ist *Code Mobility* – die Fähigkeit, die Bindung zwischen Code-Fragmenten und dem Ort ihrer Ausführung dynamisch zu verändern. Dabei wird auch auf die Unterschiede und Ähnlichkeiten zu anderen Verfahren eingegangen.

Das weit verbreitete Client-Server-Paradigma sollte hinterfragt werden, wenn Clients immer mehr eigene Ressourcen haben, und die zentralen Server nicht mit der Auslastung skalieren können – hier können die im nächsten Abschnitt vorgestellten Alternativen für Abhilfe sorgen. Danach geht es um die grundlegenden Kriterien von Mobile-Code-Systemen, also Sprachen und Umgebungen, die Code-Mobilität erlauben. Wichtig dabei ist auch der Sicherheitsaspekt bei der Ausführung von nicht vertrauenswürdigem Code, auf den der vierte Abschnitt eingeht. Schließlich wird dann darauf eingegangen, wie diese Kriterien von existierenden Mobile-Code-Systemen umgesetzt werden.

2 Paradigmen

Das Design-Paradigma, welches man für eine Anwendung wählt, hat Einfluss auf alle weiteren Schritte der Entwicklung, daher sollte die Entscheidung für ein bestimmtes Paradigma wohlüberlegt sein. Weiterhin sollte das verwendete Werkzeug an dem Paradigma ausgerichtet sein – es ist zwar auch möglich, mit nicht angepassten Tools Programme nach einem Designmuster zu schreiben, allerdings muss man dabei die musterspezifische Funktionalität selbst implementieren, statt sie vom Entwicklungssystem oder der Programmiersprache übernehmen zu können.

Im Folgenden werden drei Paradigmen vorgestellt, die Code-Mobilität umsetzen, und miteinander sowie mit dem Client-Server-Modell verglichen. Um die Auswahl des passenden Konzepts zu erleichtern, werden hier Methoden vorgestellt, nach denen die Praxistauglichkeit von Paradigmen für eigene Szenarios untersucht werden kann.

Um die Klassifizierung zu erleichtern, werden hier mehrere Begriffe definiert:

- *Komponenten* sind die Grundeinheiten dieser Strukturierung. Sie unterteilen sich in *code components*, die ausführbare Algorithmen enthalten, *resource components* (Daten oder Gerätetreiber) und *computational components*, welche in der Lage sind, Code-Komponenten auszuführen.

- *Sites* sind Orte zum Speichern und Ausführen von Komponenten. Sie besitzen zu diesem Zweck meist eigene computational components. Sites werden durch reale oder virtuelle Maschinen repräsentiert.

- *Interaktionen* sind Ereignisse oder Nachrichten, die zwischen unterschiedlichen Komponenten ausgetauscht werden – man unterscheidet zwischen lokalen und site-übergreifenden Interaktionen, wobei die ersten mit kleinerem Aufwand (und damit kleineren Kosten) verbunden sind.

Um Code ausführen zu können, muss sich die entsprechende Code-Komponente zusammen mit allen benötigten Daten und Ressourcen auf einer Site befinden. Ferner wird dazu eine Rechenkomponente benötigt. Die einzelnen Paradigmen unterscheiden sich darin, wo sich die Komponenten vor Anfang der Ausführung befinden, und wohin das Ergebnis der Ausführung kommuniziert wird.

Die Eignung eines Paradigmas für ein konkretes Problem sollte man daran bemessen, welcher Aufwand nötig ist, um alle zur Berechnung erforderlichen

Komponenten auf die ausführende Site zu bewegen, und dann das Ergebnis an den Empfänger mitzuteilen. Der Aufwand kann über die Interaktionskosten gemessen werden, die wiederum vom Umfang und der Entfernung der zu transportierenden Daten abhängen.

2.1 Client-Server

Das in Computer-Netzen am weitesten verbreitete Paradigma ist das Client-Server-Modell. Obwohl es keine Code-Mobilität anbietet, wird es hier wegen seiner Bedeutung und als Referenz aufgeführt.

Bei diesem Modell befindet sich ein Teil der Datenkomponenten vor der Ausführung auf dem Client, Code- und Rechenkomponente sowie weitere Daten sind dagegen auf dem Server. In einer ersten Interaktion übermittelt der Client seine für die Berechnung erforderlichen Daten an den Server, der dann die Code-Komponente ausführt, und das Ergebnis – wiederum eine Daten-Komponente – an den Client zurückschickt.

Dieses Paradigma ist mit zwei Nachteilen verbunden. Zum einen ist der Server ein Single Point of Failure, so dass bei Ausfall oder Überlastung alle Clients betroffen sind. Zum anderen ist die Schnittstelle zwischen Client und Server starr vorgegeben – ein Client kann nicht davon abweichen, um Interaktionskosten zu sparen, weil er nur eine Teilmenge der Daten braucht, und er kann nicht Informationen erhalten, die bei der Entwicklung der Schnittstelle nicht bedacht wurden. Weiterhin muss das Protokoll auf beiden Seiten einheitlich implementiert sein, Erweiterungen müssen immer beidseitig erfolgen, um genutzt werden zu können.

2.2 Code On Demand

Das Code-On-Demand-Modell (COD) sieht eine zentrale Speicherung des Codes vor. Wenn auf einer Site Daten vorliegen, die bearbeitet werden müssen, so fordert sie die benötigten Code-Komponenten vom zentralen Server an und arbeitet den Algorithmus mit lokalen Daten und Ressourcen auf der lokalen Rechenkomponente ab. Danach kann die Code-Komponente entweder verworfen oder für zukünftige Berechnungen zwischengespeichert werden, wobei man dabei vor der nächsten Ausführung das Vorhandensein einer aktualisierten Version auf dem Server überprüfen muss.

Dieses Modell setzt wie das vorangegangene einen zentralen Server voraus, der auch hier eine Schwachstelle darstellt. Allerdings muss er nur Code-Komponenten ausliefern und braucht weder komplizierte Berechnungen durchzuführen, noch große Mengen an Daten oder Ressourcen vorzuhalten.

COD bietet sich an, wenn auf einer großen Menge an Rechnern häufig modifizierte oder aktualisierte Software laufen soll – damit werden von Hand durchgeführte Updates aller Systeme unnötig. Außerdem lässt sich das Modell einsetzen, um Nutzungsgebühren nach tatsächlicher Beanspruchung zu erheben, z.B. durch einen Application Service Provider.

2.3 Remote Evaluation

Remote Evaluation (REV) ist ein Modell des Software-Designs, bei dem sich die für eine Berechnung erforderlichen Daten und Ressourcen auf einer entfernten Gegenstelle befinden, und die Site, die an dem Ergebnis der Berechnung interessiert ist, die Code-Komponente hat. Diese muss an die entfernte Site geschickt und dort ausgeführt werden, damit das Ergebnis zurück an die Quellsite übermittelt werden kann.

Auch dieses Verfahren hat Ähnlichkeiten zu Client-Server, die Datenspeicherung und Verarbeitung läuft zentral ab, und kann eine hohe Belastung des Servers mit sich bringen. Allerdings bietet das Modell dafür flexiblere Möglichkeiten, mit den Daten umzugehen, so dass effizient formulierte Code-Komponenten weniger Last verursachen und die Netzanbindung weniger beanspruchen, da nur die wirklich relevanten Daten zurückgeliefert werden müssen.

2.4 Mobile Agents

Der Begriff *mobile agent* oder *mobiler Agent* ist in unterschiedlichen Forschungsbereichen mit abweichenden Bedeutungen belegt, so wird er z.B. in der KI-Forschung genutzt, um selbständige „Intelligenzen" zu beschreiben.

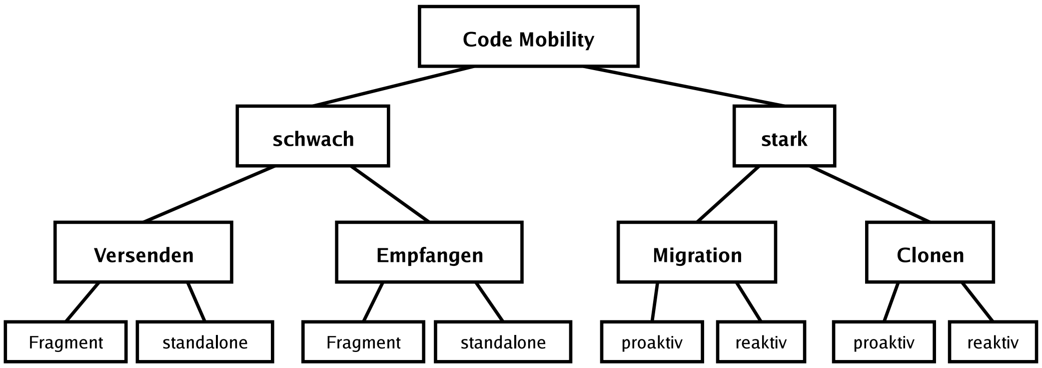
In diesem Dokument ist ein mobiler Agent (MA) eine Code-Komponente zusammen mit eventuellen Daten-Komponenten, die auf eine entfernte Site übertragen und dort ausgeführt werden können, um dort vorhandene Ressourcen zu nutzen und die eigenen Daten zu verändern oder Interaktionen durchzuführen.

Dieses Paradigma macht es möglich, einen MA auf einer fremden Site einzusetzen, damit dieser unmittelbar auf Ereignisse auf dieser Site reagieren kann. Der Versender des MA kann dabei seine Site abschalten oder die Verbindung zum Netz trennen, der Agent kehrt dann erst mit den Ergebnissen zurück, wenn die Verbindung wieder aufgebaut wurde. Alternativ kann er aber auch Ereignisse sofort an seinen Urheber signalisieren.

3 Mobile Code Systems

Mobile Code-Systeme (MCS) sind Sprachen und Umgebungen, die Mechanismen zur Unterstützung von Code-Mobilität anbieten. Solche Systeme müssen Schnittstellen für die oben erwähnten Elemente – Komponenten, Interaktionen und Sites – bereitstellen, bzw. diese implementieren. Dazu gehört eine Abstraktion von den Netzwerkprotokollen, die Möglichkeit für den Code, die aktuelle Ausführungsposition zu erfragen oder sogar zu ändern, die Verwaltung von Speicher und Ressourcen, einschliesslich der Möglichkeit, Ressourcen zwischen Komponenten zu teilen. Ausserdem sollte das MCS von der tatsächlich vorhandenen Hardware abstrahieren, damit der mobile Code auf unterschiedlichen Plattformen laufen kann.

Man unterscheidet zwischen *schwacher* und *starker Code-Mobilität*:



- *schwache Code-Mobilität* bedeutet, dass nur statische Daten und Code übertragen werden können, und der Code nach der Übertragung ausgeführt wird. Dabei wird unterschieden, ob die Aktion von der versendenden oder von der empfangenden Site initiiert wurde, und ob Code-Fragmente oder selbständige Anwendungen übertragen werden.

- *starke Code-Mobilität* dagegen erlaubt die Migration von laufenden Prozessen. Läuft der Prozess nach seiner Migration auch auf der Quell-Site weiter, so spricht man von *Clonen*. Weiterhin kann eine Migration *proaktiv*, vom Mobile-Code selbst, oder *reaktiv*, also durch Einwirkung von außen, eingeleitet werden.

Die Übertragung von laufendem Code auf ein anderes System erfordert ein ausgeklügeltes Ressourcen-Management, welches Bindungen zwischen dem Code und Ressourcen nach der Migration neu aufbaut oder der Anwendung signalisiert, dass sie ungültig geworden sind. Die Entscheidung kann aufgrund von zwei Kriterien gefällt werden, das erste ist, welche Bindung die Anwendung zur Ressource eingegangen ist:

- *über Typ* – nur die Art der Ressource spielt eine Rolle, das System kann stillschweigend diese Ressource durch eine gleichartige auf dem Zielsystem besser erreichbare ersetzen. Solche Ressourcen können Temporärspeicher oder site-lokale Einstellungen repräsentieren.

- *über Wert* – der Inhalt der Ressource ist relevant, sie kann bei der Migration auf das Zielsystem kopiert werden. Auf diese Weise lassen sich statische Daten „mitnehmen".

- *über Namen* – die Ressource ist eindeutig gekennzeichnet, nach der Migration muss – sofern überhaupt möglich – genau diese Ressource wieder angesprochen werden können. Über diese Form der Bindung könnte ein Mobile Agent z.B. ein Fenster auf seinem „Heimsystem" geöffnet halten und nach seinen Migrationen Status-Updates in diesem Fenster ausgeben.

Das andere Kriterium beim Wiederaufbau einer Bindung ist die „Bewegungsfreiheit" der Ressource, die von ihrer Art und ihren Eigenschaften abhängt:

- *not transferrable* - eine solche Ressource kann ihre Site nicht verlassen, da sie physikalisch daran gebunden ist. Peripheriegeräte wie Drucker oder Monitor sind nicht transferierbar.

- *fixed, transferrable* - diese Ressourcen können zwar auf eine andere Site kopiert, jedoch nicht vom Ausgangssystem gelöscht werden.

- *free, transferrable* - solche Ressourcen lassen sich frei übertragen.

Dem System stehen unterschiedliche Verfahrensweisen zur Verfügung, wie mit einer Ressourcenbindung bei der Migration verfahren werden kann:

- *verwerfen* ist die einfachste Art, die Bindung wird einfach für ungültig erklärt.

- als *Netzbindung* wiederherstellen, also die ursprüngliche Ressource für das migrierte Programm transparent weiter bereitstellen.

- *re-binding*, Binden an eine lokale Ressource des gleichen Typs.

- *duplizieren*, Erstellen einer Kopie auf der Zielsite.

- *verschieben*, Duplizieren und Löschung des Originals.

Die folgende Tabelle fasst zusammen, welche Verfahrensweisen in Abhängigkeit von Ressource und Art der Bindung verwendet werden können:

	Art der Bindung		
Ressourcenart	über Typ	über Wert	über Namen
not transferrable	re-binding (Netzbindung)	Netzbindung	Netzbindung
fixed, transferrable	re-binding (duplizieren, Netzbindung)	duplizieren (Netzbindung)	Netzbindung
free, transferrable	re-binding (duplizieren, verschieben, Netzbindung)	duplizieren (verschieben, Netzbindung)	verschieben (Netzbindung)

Die eben vorgestellten Merkmale werden nicht von allen MCS implementiert, meist schränken die Sprachen die Flexibilität der Ressourcen-Anbindung ein. Daher ist der Umfang, in dem obige Eigenschaften in einer Sprache vorhanden sind, ein weiteres Kriterium, an dem die Sprache ausgewählt werden sollte.

4 Sicherheit bei Mobile Code

Die Sicherheit ist ein sehr wichtiger Aspekt für Mobile-Code-Umgebungen, da unterschiedliche Code-Komponenten, die auf der selben Site laufen, sich nicht gegenseitig stören dürfen, selbst wenn ein Teil des Codes aus nicht zuverlässigen Quellen stammt. Man unterscheidet zwei wichtige Aspekte der Sicherheit:

Unter *safety* versteht man die Sicherheit vor Fehlern im Code oder in der Umgebung. Safety sollte auf Betriebssystemebene, in der Programmiersprache, im dazugehörigen Binärcode, und zur Laufzeit sichergestellt werden.

Dagegen wird unter *security* die Sicherheit gegenüber Angriffen verstanden. Security erfordert ein funktionierendes Safety-Konzept, da bei einem Angriff sonst Lücken im System ausgenutzt werden können, geht aber noch weiter, da man auch übermäßige Ressourcen-Nutzung und andere Attacken ausschliessen muss. Nach [3] wird security in vier Ziele unterteilt, die ein MCS einhalten muss:

- *Vertraulichkeit* - Schutz vor dem Ausspionieren von Daten
- *Integrität* - Schutz vor Veränderung
- *Verfügbarkeit* - Sicherstellung des Betriebs
- *Authentizität* - Echtheit der Daten und des Urhebers

Die Sicherheit hängt von allen Schichten eines Systems ab, sie ist eine Art Kette, die ein Angreifer an der schwächsten Stelle aufbrechen kann. Daher muss sie auf jeder Ebene beachtet werden.

Auf der *Kommunikationsebene* bei vernetzten Systemen kann, um safety zu gewährleisten, ein gegenüber Fehlern robustes Protokoll eingesetzt werden. Security spielt eine Rolle, wenn Datenverbindungen über nicht vertrauenswürdige Sites gehen, wo Abhören oder Manipulation möglich ist. Durch Verwendung kryptographischer Verfahren zur Authentifizierung der Gegenstellen und zur Verschlüsselung der Daten (z.B. IPSec oder SSL), kann soetwas ausgeschlossen werden. Es ist jedoch nicht möglich, Verfügbarkeit zu garantieren, wenn man nicht alle Sites unter Kontrolle hat.

Das *Betriebssystem* muss auch einen wesentlichen Beitrag zur Sicherheit leisten. Durch die Verwendung (und Verwaltung) von Hardware-Speicherschutz können einzelne Prozesse voneinander isoliert werden, so dass sie nur über die Betriebssystem-Schnittstellen miteinander kommunizieren können und gegenseitige Störungen nicht möglich sind. Die Schnittstellen sind allerdings vom verwendeten Betriebssystem abhängig, was plattformübergreifende Entwicklung erschwert. Ausserdem ist Hardware-Speicherschutz auf vielen portablen Geräten wie Mobiltelefonen und PDAs, die ein wichtiges Ziel von Mobile-Code-Anwendungen sind, nicht vorhanden.

In diesem Fall kann eine *abstrakte Maschine* eingesetzt werden, die den unsicheren Code interpretiert, unzulässige Zugriffe abfängt, und damit einen Speicherschutz in Software bietet. Eine solche Maschine bietet weiterhin eine einheitliche Schnittstelle für Programme, die nicht von der verwendeten Plattform oder Architektur abhängig ist, und damit gute Voraussetzungen für Mobile Code bietet. Sie benötigt aber einen Zwischencode, der von ihr interpretiert wird, was mit deutlich höherem Aufwand verbunden ist, als direkte Ausführung durch die CPU.

Die Sicherheit kann auch auf Ebene der *Programmiersprache* verbessert werden. Dazu muss die Sprache unsichere Elemente wie Zeigerarithmetik und Speichermanagement aus den Händen des Programmierers nehmen, und weitere Überprüfungen wie z.B. von Array-Grenzen vorsehen. Wenn die Sicherheitsüberprüfung nur zur Compile-Zeit stattfindet, darf der erzeugte Binärcode durch spätere Manipulationen nicht für andere Anwendungen gefährlich werden. Ausserdem sollten Sprachkonstrukte mit Zugriff zur lokalen Site nicht uneingeschränkt nutzbar sein, und die Verwendung von Ressourcen sollte sich durch die Site begrenzen lassen.

Diese Sicherheitskriterien, die für die Ausführung fremden Codes besonders wichtig sind, sind ein weiteres Kriterium zur Auswahl der richtigen Programmiersprache, und der dazugehörigen Laufzeitumgebung.

5 Programmiersprachen

Dieses Kapitel wendet die oben dargestellten Klassifizierungsverfahren für Mobile-Code-Systeme beispielhaft auf drei Programmiersprachen an, besonders wird auf die Sicherheit der Sprachen eingegangen. Weitere Sprachen lassen sich nach dem selben Muster untersuchen und einordnen.

5.1 JAVA

JAVA[4] ist eine von Sun entwickelte objektorientierte Sprache. Die primären Ziele bei der Entwicklung waren ein klares Objektmodell, eine für viele Entwickler verständliche C-ähnliche Syntax und Plattformunabhängigkeit, die durch die Verwendung von binärem Zwischencode (Bytecode) sichergestellt wurde.

Der Bytecode einer JAVA-Anwendung wird durch die JVM (Java Virtual Machine) interpretiert, die gleichzeitig eine Abkapselung von der ausführenden Site darstellt, also eine Sandbox. Es existieren JVMs für zahlreiche Plattformen, auch für mobile Geräte wie Handys oder PDAs.

Das Objektmodell von JAVA ist klassenbasiert, und die Standardumgebung bringt viele Klassen für Arbeit mit Ressourcen und für Benutzer-Interaktionen mit. Weitere Klassen können zur Laufzeit automatisch oder explizit vom laufenden Code angefordert werden. Sie werden dann durch den *class loader*, der Teil jeder JVM ist, nachgeladen.

Die Bekanntheit von JAVA ist auf die leichte Integration in Webseiten zurückzuführen. Dabei werden JAVA-Applets in ein HTML-Dokument eingebettet, der dafür notwendige Code wird vom Webserver geladen und lokal in der Browser-JVM gestartet. Dies ist eine Form schwacher Code-Mobilität, bei der Code-Fragmente empfangen und ausgeführt werden. Dadurch, dass dabei keine Bindungen zu dem den Code beherbergenden Server erhalten bleiben, entfällt die Notwendigkeit einer komplexen verteilten Ressourcen-Verwaltung.

Es existiert jedoch mit der JAVA Aglets API[5] eine von IBM entwickelte Erweiterung, die proaktive Code-Migration und Ressourcenübertragung durch duplizieren erlaubt, sich aber auch nur auf schwache Mobilität beschränkt.

Zum Safetykonzept von JAVA gehört, dass auf unsichere Elemente in der Sprache weitgehend verzichtet wurde. Es gibt keine Zeigeroperationen und unsichere Typumwandlungen, unions und der von C bekannte Präprozessor wurden ebenfalls nicht implementiert. Das Speichermanagement wurde dem Programmierer komplett aus der Hand genommen und Zugriffe auf Felder und Strings müssen immer von der JVM auf ihre Zulässigkeit überprüft werden.

Fehlerzustände werden durch Exceptions behandelt, die den Code-Ablauf im Fehlerfall unterbrechen und an dem entsprechenden Exception Handler fortsetzen – das erleichtert dem Programmierer den Umgang mit Laufzeitproblemen, und schützt gleichzeitig davor, dass in unvorhergesehenen Fehlersituationen Code ausgeführt wird, der das Problem nicht berücksichtigt.

Zugriffe auf Ressourcen der lokalen Site dürfen nur durch bestimmte Klassen vorgenommen werden, die zur Laufzeitumgebung gehören und nicht ableitbar sind, und damit weder erweitert noch manipuliert werden können. Die Rechte, die diesen Klassen zustehen, werden vom *SecurityManager* kontrolliert, der vom Benutzer konfiguriert werden kann.

Obwohl JAVAs Sicherheitskonzept vielversprechend ist, und von der virtuellen Maschine bis zum Sprachdesign greift, ist es hauptsächlich auf die Ausführung von Applets ausgerichtet und macht deshalb Unterschiede zwischen über das Netz und von Festplatte geladenen Klassen. Außerdem sieht es einige Dinge, wie die Beschränkung des Arbeitsspeichers oder der Prozessorzeit nicht zu.

5.2 Telescript

Die Sprache Telescript[6] wurde 1996 von General Magic mit dem Ziel entwickelt, virtuelle Marktplätze realisieren zu können. Sie ist wie JAVA klassenbasiert und objektorientiert, ihr Schwerpunkt liegt jedoch auf Kommunikation.

Die Sprache setzt starke proaktive Code Mobility um, und verwendet dafür eine portable Zwischensprache namens *Low Telescript*. Objekte in dieser Zwischensprache (*agents*) können zwischen Sites (die sich jeweils aus einer *Engine* und ineinander verschachtelbaren *Places* zusammensetzen) bewegt werden. Dafür gibt es die Befehle *go* und *send*, mit denen ein Agent proaktiv auf ein anderes Place migrieren oder sich dorthin klonen kann. Dabei werden die von dem Agenten benutzten Ressourcen zusammen mit ihm migriert, und die Bindungen auf der lokalen Site verworfen.

Das auf Metaphern der realen Welt basierte Konzept wird auch bei der Abrechnung der Ressourcennutzung durchgesetzt: jede Ressource und jeder Agent haben einen Besitzer, der die Kosten der Benutzung zu tragen hat. Damit ein Agent auf ein anderes Place migrieren kann, benötigt er zuvor ein *Ticket*, in dem das Ziel und eventuell die Route spezifiziert sind.

Ein Place kann entscheiden, ob es einen ankommenden Agenten akzeptiert oder ablehnt. Im positiven Fall setzen sich die dem Prozess erlaubten Aktionen aus dem Mengen-Durchschnitt der *Permit*s der engine, aller verschachtelten Places und von dem Agenten selbst. Diese regeln seine maximale Verweilzeit und Speicheraufnahme, aber auch ob der Agent weiterwandern, Unterprozesse erstellen und die Permits anderer Prozesse beeinflußen darf.

Damit auch andere Einflüsse ausgeschlossen werden, läuft jeder Prozess in einem eigenen Kontext in der engine, so dass alle Aktionen zwischen unterschiedlichen Prozessen nur über eine einheitliche Schnittstelle der engine stattfinden. Zusätzlich stellt die engine sicher, dass Objekte durch einen Absturz oder Neustart nicht gefährdet werden.

Das Sicherheitskonzept von Telescript ist von Anfang an in das Sprachdesign eingearbeitet worden. Es beachtet sowohl safety als auch security - ein Bug in einem Objekt kann keine anderen Objekte beeinflussen, und Angriffe auf fremde Objekte oder die Systemressourcen sind durch die einheitliche Kommunikationsschnittstelle und durch die Ressourcenbegrenzung kaum möglich. Allerdings ist ein solches Sicherheitskonzept für eine Sprache, deren Entwicklungsziel die Darstellung von Marktplätzen ist, in denen mit realem Geld umgegangen wird, auch zwingend notwendig.

5.3 Safe-TCL

Im Gegensatz zu den oben vorgestellten Sprachen ist Safe-TCL[7] eine (prozedurale) Scriptsprache, d.h. es existiert weder ein Zwischencode noch eine Binärform. Safe-TCL wurde als MIME-Erweiterung vorgesehen, um E-Mails mit aktiven Inhalten versehen zu können, und stellt eine Abwandlung von TCL mit eingeschränktem Funktionsumfang dar.

Da die Verbreitung von Safe-TCL-Code mit E-Mails stattfindet, existieren in der Sprache keine Elemente zur Steuerung der Code-Mobilität. Es ist lediglich über die zusätzlichen MIME-Angaben möglich, zu bestimmen, ob das Script beim Versenden, beim Empfangen oder beim Ansehen der E-Mail ausgeführt werden soll. Bei der alternativen Verbreitungsform über Webseiten, wo die Interpretation im Browser stattfindet, fallen diese drei Phasen zusammen.

Die Syntax von Safe-TCL entspricht der von TCL, entsprechend existieren auch keine Zeiger mit unsicheren Operationen, und der einzige vorhandene Datentyp ist String. Numerische Operationen sind mit einer Konvertierung zu Zahlen und wieder zurück zu Strings verbunden, und auch Programmcode wird mit Strings repräsentiert. Das macht die Variablenverwaltung einfacher und unanfälliger gegen Fehler, erhöht aber immens die Verarbeitungszeit. Variablen können dabei immer entweder global oder in der aktuellen Funktion lokal sein.

Der Funktionsumfang von TCL wurde für Safe-TCL angepasst, um nicht vertrauenswürdigen Code ausführen zu können. Dabei wurden alle Funktionen von TCL auf ihre Eignung überprüft und entweder eliminiert oder durch weniger generische ersetzt. So gibt es statt vollem Dateizugriff nur einen Mechanismus um Konfigurationsdaten zu schreiben und zu lesen, der ähnliche Funktion wie die Cookie-Speicherung in Webbrowsern bietet.

Um eine Abgrenzung zwischen Safe- und vollständigem TCL zu gewährleisten, existieren für beide Varianten eigene Interpreter, wobei der Safe-TCL-Interpreter nur sichere Funktionen implementiert. Dadurch ist dieser kompakt und lässt sich leichter in eigene Systeme einbinden.

6 Zusammenfassung

Zuerst hat diese Arbeit dargelegt, dass Code-Mobilität durch die fortschreitende Entwicklung digitaler Netzwerke eine Daseinsberechtigung hat und eine höhere Aufmerksamkeit seitens der Wissenschaft verdient.

Dann wurden Methoden vorgestellt, nach denen sich Paradigmen für Mobile Code unterteilen lassen, und wie man ihre Eignung für die eigene Problemstellung einschätzen kann. Als typische Paradigmen wurden „Client-Server", „Remote Evaluation", „Code on Demand" und „Mobile Agents" vorgestellt.

Der nächste Abschnitt befasste sich mit Gliederungsmöglichkeiten für Mobile Code Systeme, also den Code-Mobilität unterstützenden Sprachen und ihren Umgebungen. Da wurden die Migrationsmöglichkeiten von Code und die dabei in Frage kommenden Verfahren zum Ressourcenmanagement untersucht.

Im vierten Kapitel wurden die bei Mobile Code besonders wichtigen Sicherheitsaspekte untersucht. Es erfolgte eine Unterteilung in die vier Schichten Kommunikation, Betriebssystem/Hardware, abstrakte Maschine und Programmiersprache, für die jeweils auf safety und security, also Sicherheit vor Fehlern und vor bewußten Angriffen, eingegangen wurde.

Im letzten Abschnitt wurden dann die zuvor gezeigten Verfahren der Klassifizierung auf die als Beispiel ausgewählten Programmiersprachen JAVA, Telescript und Safe-TCL angewandt. Dabei wurden die unterstützten Migrationsformen von Code und Daten, aber auch die Aspekte der Sicherheit auf den unterschiedlichen Schichten betrachtet.

Wenn man die beste Umgebung zur Implementierung einer Problemlösung sucht, sollte man sich zunächst für das sinnvollste Paradigma zur Umsetzung entscheiden. Diese Entscheidung kann erleichtert werden, wenn man die Lösung nach unterschiedlichen Paradigmen modelliert und die anfallenden Interaktionskosten betrachtet.

Daraufhin sollte man aus den möglichen Mobile Code Systemen das auswählen, welches das passende Paradigma am besten umsetzt. Dabei gilt zu beachten, dass man nicht zwingend auf eine vollständig vorhandene Implementierung aller Möglichkeiten angewiesen ist, sondern diese in einem gewissen Rahmen auch selbständig nachgerüstet werden können. Dafür sollten dann aber Betrachtungen der gebotenen und der erforderlichen Sicherheit in die Entscheidung einfließen.

Die letztendliche Implementierung kann durch eine geschickte Auswahl von Paradigma und Umgebung erleichtert werden, ganz entfallen wird sie aber nicht.

Literatur

[1] A. Fugetta, G.P. Picco, G. Vigna„Understandig Code Mobility", IEEE Transactions On Software Engineering, Vol. 24, No. 5, May 1998.

[2] T. Thorn, „Programming Languages for Mobile Code", ACM Computing Surveys, Vol. 29, No. 3, Sep 1997.

[3] D. Russell, G.T. Gangemi Sr., „Computer Security Basics", 1991, O'Reilly & Associates Inc., Sebastopol, CA.

[4] Sun Microsystems, „The Java Language: An Overview", Technical Report, 1994, Sun Microsystems.

[5] http://www.trl.ibm.com/aglets/

[6] J.E. White, „Telescript Technology: Mobile Agents", Software Agents, J. Bradshaw, ed. AAAI Press/MIT Press, 1996.

[7] N. Borenstein, „EMail with A Mind of Its Own: The Safe-Tcl Language for Enabled Mail", technical report, First Virtual Holdings, Inc. 1994.