1. Motivation

Verteilte Applikationen gewinnen in der heutigen Welt, in der die meisten Rechner in einem Netzwerk miteinander verbunden sind, immer mehr an Bedeutung. Lokale Netzwerke sind i.a. mit dem Internet verbunden und bilden so ein sehr großes verteiltes heterogenes dynamisches System, in dem sehr viele Benutzer über viele verteilte Applikationen miteinander interagieren. Dabei werden unterschiedliche Nutzerschnittstellen unterstützt, wie z.B. die verschiedenen Browser für das WWW oder unterschiedliche Klienten zum Zugriff auf Nachrichtendienste oder auf Systeme für gemeinsamen Dateizugriff. Um in diesen großen Systemen eine akzeptable Effizienz und Zuverlässigkeit gewährleisten zu können, werden komplexe Mechanismen wie Replikation oder automatische Fehlererkennung und -korrektur eingesetzt. Aktuelle verteilte Applikationen besitzen eine hohe Komplexität und es ist davon auszugehen, dass diese in Zukunft weiter wachsen wird.

Zusätzlich bestehen heutzutage selbst zentralisierte Applikationen häufig aus einem erweiterbaren Rahmen, der zur Laufzeit um Komponenten erweitert werden kann, die aus dem Netz geladen werden. Typische Beispiele sind Anzeigeapplikationen für Multimediadaten, wie z.B. QuickTime von Apple, oder Entwicklungsumgebungen wie Netbeans oder jEdit. Häufig lässt sich dieser Rahmen selbst über das Netzwerk installieren und aktualisieren. Sogar die unterste Systemschicht aus der Software-Perspektive, das Betriebssystem, lässt sich auf diese Art verwalten. Das bedeutet, dass auch klassische nichtverteilte Applikationen Eigenschaften verteilter Applikationen übernehmen.

Die Entwicklung verteilter Applikationen weist im Vergleich zu der Entwicklung zentralisierter Applikationen zusätzliche Schwierigkeiten auf. Verteilte Applikationen sind bzgl. ihres Programmcodes groß und müssen geeignet hierarchisch strukturiert werden, damit sie in den Entwicklungs- und Wartungsphasen handhabbar sind. Diese Strukturierung muss zusätzlich berücksichtigen, dass verteilte Applikationen meistens aus lose gekoppelten, zentralisierten Komponenten bestehen.

Diese lose Kopplung ist zum einen in der geringen Bandbreite und der hohen Latenzzeit bei der Nutzung eines Netzwerks begründet. Um eine akzeptable Leistung der Applikation zu erzielen, versuchen Applikationsentwickler daher, die Kommunikation zwischen Rechnergrenzen zu minimieren. Eine weitere Ursache für diese lose Kopplung ist die Heterogenität der Komponenten einer Applikation, die evtl. in unterschiedlichen Programmiersprachen für unterschiedliche Systeme entwickelt wurden. Eine Kommunikation zwischen in diesem Sinne heterogenen Komponenten ist technisch aufwendig.

Objektorientierte Ansätze zur Strukturierung von Applikationen setzen in-

tensiv Vererbungsmechanismen ein, die eine starke Kopplung zwischen den einzelnen Klassen bewirken. Sie bereiten daher Probleme bei dem Entwurf verteilter heterogener Systeme. Architektur-basierte Ansätze erlauben dagegen, eine Applikation mit einer groben Granularität in Komponenten zu zerlegen, die über wenige, genau definierte Schnittstellen miteinander interagieren. So lassen sich lose gekoppelte Systeme adäquat beschreiben.

Um die technischen Probleme zu lösen, die sich aus der Heterogenität und Dynamik eines modernen verteilten Systems ergeben, setzt man bei der Applikationsentwicklung auf höherwertige Dienste auf, die von Middleware-Systemen angeboten werden. Diese Dienste lösen die obigen Probleme oder vereinfachen zumindestens deren Lösung stark.

So bewältigt z.B. die Middleware CORBA das Problem der Heterogenität bzgl. verwendeter Programmiersprachen, indem es standardisierte Abbildungen von einer allgemeinen Schnittstellenbeschreibungssprache in die jeweiligen Programmiersprachen anbietet. Systeme wie Condor oder Globus lösen das Problem der nichtdeterministisch und dynamisch sich ändernde Menge verfügbarer Rechenressourcen, indem sie eine verteilte Infrastruktur anbieten, die das Gesamtsystem überwacht und Informationen über Verfügbarkeit und Qualität der Ressourcen über Anfragedienste zur Verfügung stellt.

Der Einsatz solcher Middleware vereinfacht die Applikationsentwicklung so stark, dass man postulieren kann, dass die Entwicklung von Applikationen schon ab mittlerer Größe ohne Einsatz von Middleware praktisch nicht sinnvoll ist. Trotzdem ist auch bei ihrem Einsatz ein Wissen um die prinzipielle Funktionsweise der Middleware und ein der Mächtigkeit der Middleware entsprechender Einarbeitungsaufwand notwendig, um diese möglichst fehlerfrei und effizient zu benutzen. Hinzukommend gibt es sehr viele Middleware-Systeme, die alle für unterschiedliche Applikationsbereiche und für unterschiedliche Zielsysteme entwickelt wurden. Um für eine konkrete Anwendung die geeignete Middleware auszuwählen, bedarf es einer hohen technologischen Kompetenz für die Evaluation der Alternativen.

Zusammenfassend läßt sich sagen, dass immer mehr Applikationen entweder verteilt sind oder zumindestens bei der Installation und Wartung entfernte Ressourcen einsetzen. Die Werkzeuge für die Applikationsentwicklung müssen dieser Tatsache Rechnung tragen. Aufgrund der vielfältigen Probleme bei der Entwicklung verteilter Applikationen, die sich mit der Vielfältigkeit vorhandener unterstützender Middleware kombiniert, ist es aktuell unklar, welche Eigenschaften ein Entwicklungswerkzeug besitzen sollte, um eine größtmögliche Unterstützung anzubieten.

Verteilungstransparente Applikationsentwicklung

Ein Ansatz besteht darin, eine größtmögliche Transparenz zu gewährleisten. Der Entwickler braucht sich dabei insbesondere nicht um die Verteilung der einzelnen Applikationskomponenten auf die heterogenen Ressourcen zu kümmern. Neben einem minimalen Entwicklungsaufwand verspricht dieser Ansatz auch eine gute Leistung der Applikation in einer dynamischen Umgebung. Hierfür kann ein Laufzeitsystem für die Applikation abhängig von der

aktuellen Lastverteilung eine optimale Verteilung bestimmen. Ändert sich die Last im System kann das Laufzeitsystem die Applikation im laufenden Betrieb umkonfigurieren.

Dieser vielversprechende Ansatz birgt allerdings einige schwerwiegende Probleme, die dazu geführt haben, dass solche Systeme trotz eines hohen Forschungs- und Entwicklungsaufwand nur für kleine Anwendungsbereiche zur Verfügung stehen. In dem bekannten Papier [WWWK94] ist eine umfassende Argumentation zu finden, warum Verteilungstransparenz prinzipiell vermieden werden sollte. Die aus der Sicht des Autors dieser Arbeit wichtigsten Argumente sind die folgenden:

- Für eine optimale automatische Verteilung der Applikation benötigt das Laufzeitsystem Information über den Rechenaufwand in den einzelnen Applikationsteilen. Dieser lässt sich schon aufgrund des Halteproblems in der Klasse der ν-rekursiven Funktionen nicht automatisch bestimmen. Dieses Problem lässt sich mit Annotationen und Messungen vorheriger Programmläufe umgehen. Das System Coign nutzt z.B. das Komponentensystem COM und Messungen von Laufzeiten und Kommunikationsverhalten, um eine automatische Verteilung durchzuführen [HS98]. Beide Ansätze stellen aber nur Abschätzungen dar, deren Genauigkeit und damit die Qualität der automatischen Verteilung nicht garantiert werden können.

- In einem verteilten System können einzelne Subsysteme ausfallen, während in einem zentralisieren System i.a. nur das ganze System ausfallen kann. Da die Fehlerwahrscheinlichkeit in einem offenen verteilten System zusätzlich sehr hoch ist, muss eine verteilte Applikation in der Lage sein, Teilausfälle zu tolerieren. In einem transparenten Entwicklungssystem muss allerdings das Laufzeitsystem selbstständig die Fehlerbehandlung durchführen und kann hierfür nur das Applikationswissen einsetzen, das sich durch eine automatische Analyse gewinnen läßt.

  Ein Ansatz besteht darin, den gesamten Applikationszustand in regelmäßigen Abständen zu speichern (Checkpointing). Kommt es zu einem Teilausfall wird die gesamte Applikation terminiert und von dem zuletzt gespeicherten Zustand wieder neu gestartet. Dieses ist besonders einfach, wenn die Applikationsarchitektur auf einem Entwurfsmuster mit einer regelmäßigen zentralen Synchronisation basiert. So unterstützt das System MW z.B. nur das Farm-Muster, welches eine regelmäßige Speicherung des Systemzustands bei dem Master-Prozess auf natürliche Weise unterstützt [GKLY00]. Besitzt die Applikation keine Kommunikationsstruktur mit eine regelmäßigen Synchronisation ist dieser Ansatz wesentlich schwieriger. Erstens sind Nachrichten zu berücksichtigen, die gerade übertragen werden, bzw. noch nicht von Prozessen angenommen worden. Zweitens ist das Wiederaufsetzen der Applikation komplexer. Der Zustand von jedem Prozess der Applikation muss regelmäßig gespeichert werden. Beim Wiederaufsetzen der Applikation kann nicht einfach die jeweils letzte Speicherung benutzt werden, son-

dern es muss gewährleistet sein, dass keine kausale Abhängigkeit zwischen den Zuständen existiert [NX95].

Ein in verteilten Systemen häufig eingesetzter Ansatz zur Erhöhung der Zuverlässigkeit besteht darin, einen fehlgeschlagenen entfernten Aufruf solange zu wiederholen, bis dieser gelingt. Eine derartige Funktionalität kann von dem Übersetzer automatisch und transparent für den Applikationsentwickler der Applikation hinzugefügt werden. Wenn ein entfernter Aufruf aber nicht atomar istund Seiteneffekte auslöst, können fehlgeschlagene Aufrufe zu fehlerhaften Ergebnissen der Applikation führen.

Letztendlich sind daher Transaktions-ähnliche Mechanismen nötig, um die Korrektheit der Applikation zu gewährleisten. In [RW02] wird ein System vorgestellt, das die Programmiersprache Java um explizite Transaktionsmechanismen für zentralisierte Applikationen erweitert. Aufgrund des dafür zusätzlichen Verwaltungsaufwands ergaben sich Verlangsamungen des Programms um mindestens einen Faktor von sechs. Eine Ausweitung dieses Ansatzes auf verteilte Applikationen würde sicherlich zu einem noch höheren zusätzlichen Aufwand führen.

Zusätzlich zu diesem signifikanten zusätzlichen Aufwand ist ein automatisches System i.a. nicht in der Lage, dem Anwender eine verständliche Fehlermeldung evtl. mit einem Hinweis für eine Fehlerbehebung zu geben. Hierfür müsste es ein Verständnis für den aktuellen Arbeitskontext des Besitzers besitzen. Da dem Applikationsentwickler die Verteilung verborgen bleibt, ist es auch ihm nicht möglich, eine adäquate Fehlermeldung oder gar eine Fehlerbehandlung zu implementieren.

- Es gibt Anwendungen, bei denen der Applikationsentwickler Kontrolle über die Verteilung der Applikation haben muss. So kann z.B. die Applikation eine bestimmte Altsoftware einsetzen, die spezielle Fähigkeiten eines bestimmten Systems einsetzt und nur auf diesem lauffähig ist. Oder eine Applikation besitzt grafische Benutzerschnittstellen, die auf den Rechnern der entsprechenden Anwender zu platzieren sind. In beiden Fällen muss der Entwickler in der Lage sein, die Verteilungstransparenz zu durchbrechen.

Manuelle Entwicklung

Der umgekehrte Ansatz besteht darin, nur Middleware-Systeme einzusetzen, die keine höherwertigen Dienste anbieten, und diese mit einem Entwicklungswerkzeug verknüpfen, das keine zusätzlichen Dienste hinzufügt. Dieser Ansatz ist für alle Anwendungsbereiche brauchbar und bietet Potenzial für eine hohe Effizienz der Anwendungen. Er ist aber mit einem hohen Entwicklungsaufwand für den Applikationsentwickler verbunden. Er muss alle Optimierungen, die für ein akzeptables Laufzeitverhalten notwendig sind, zuerst erkennen und dann geeignet umsetzen. Zusätzlich erzwingt jede Änderung des Zielsystems, wie z.B. die Integration oder das Entfernen von Rechen- und Netzwerkressourcen, eine händische Anpassung der Applikation.

Applikationsbereichsabhängige Entwicklungswerkzeuge

Es gibt Anwendungsbereiche, für die man leistungsfähige, von den technischen Details der Implementierung abstrahierende Entwicklungswerkzeuge einsetzen kann. So können z.B. rechenintensive numerische Algorithmen in einer Programmiersprache ohne explizite Konstrukte für Verteilungsanweisungen entwickelt und automatisch in einer dynamischen Umgebung verteilt und ausgeführt werden. Auch für Anwendungen, in denen der Entwickler die Verteilung vornehmen muss, lassen sich Werkzeuge entwickeln, die ihn dabei unterstützen. So bietet z.B. das Verteilungssystem Pangaea ein grafisches Entwicklungswerkzeug an, mit dem ein Anwendungsentwickler eine Applikation komfortabel in Komponenten zerlegen und auf die gewünschten Zielrechner verteilen kann [Spi00].

Dieser Ansatz ist aber nur dann möglich, wenn eine Applikation komplett im Quellcode vorliegt und möglichst nur eine einzige Programmiersprache verwendet wird, die sich zudem auch gut analysieren lässt. Werden dagegen schwer analysierbare Programmiersprachen, wie z.B. C und damit auch C++, in Kombination verwendet oder kommt Altsoftware zum Einsatz, die nur in binärer Form vorliegt und praktisch nicht analysiert und angepasst werden kann, dann ist dieser komfortable Ansatz, eine Applikation halbautomatisch zu verteilen, nicht durchführbar.

Kombinierter Einsatz spezialisierter Entwicklungswerkzeuge

Ein in diesem Kontext erfolgversprechender Ansatz besteht darin, für unterschiedliche Problembereiche spezialisierte Entwicklungswerkzeuge kombiniert einzusetzen. Die zentralisierten Komponenten einer Applikation werden in einer prinzipiell beliebigen Programmiersprache entwickelt, die sich für die bestimmte Funktionalität der Komponente besonders gut eignet. Die einzelnen Komponenten werden dann mit einer Middleware zu einer verteilten Applikation verbunden. Wie sie miteinander verbunden werden, wird aber nicht in den Komponenten selber implementiert. Statt dessen wird eine spezielle Koordinationssprache eingesetzt, die erstens für die Komposition von Komponenten geeignete Programmierkonstrukte enthält und zweitens die eingesetzte Middleware unterstützt.

Mit diesem auf drei Schichten basierenden Ansatz lassen sich spezialisierte Entwicklungswerkzeuge miteinander zu einer umfassenden Entwicklungsumgebung für verteilte Applikationen verbinden. Da für die Erstellung zentralisierter Komponenten leistungsfähige Systeme existieren und dies ebenfalls für die Middleware-Systeme gilt, konzentriert sich diese Arbeit im folgenden auf die Problematik der obersten Schicht, der Koordinationssprache. Da mittlerweile, wie eingangs erwähnt, in praktisch allen Applikationsklassen Verteiltheit benötigt wird, ist nicht davon auszugehen, dass sich eine bestimmte feste Koordinationssprache als Grundlage für alle verteilten Applikationen eignen kann. Statt dessen wird eine offene Koordinationssprache benötigt, die durch Erweiterungen an die unterschiedlichen Anforderungen der jeweiligen Applikation angepasst werden kann.

Ein modular erweiterbarer Rahmen für Entwicklungswerkzeuge

Als ein Schritt in Richtung einer solchen Koordinationssprache wird in dieser Arbeit ein erweiterbarer Rahmen (*Framework*) vorgestellt, der mit sogenannten Erweiterungsmodulen an unterschiedliche Middleware-Systeme und an unterschiedliche Anwendungsbereiche adaptiert werden kann. Wenn der Rahmen mit den gewünschten Erweiterungsmodulen gefüllt ist, stellt er eine Koordinationssprache zur Verfügung, die der Entwickler nutzen kann, um seine verteilte Applikation zu erstellen.

Dieser Rahmen wird anhand mehrerer Fallstudien für rechenintensive Applikationen und für verteilte Informations- und Kontrollsysteme eingesetzt. Für diese beiden Applikationsbereiche werden spezielle Erweiterungsmodule vorgestellt, die eine einfache und schnelle Applikationsentwicklung ermöglichen. Zusätzlich zu den Erweiterungsmodulen sind auch verteilte Infrastrukturen notwendig, die den Applikationen eine Laufzeitumgebung bereitstellen. Es wird gezeigt, dass die Effizienz der erstellten Applikationen trotz der komfortablen Entwicklungsumgebung stets ausreichend gut ist.

Anhand des Entwurfs der Erweiterungsmodule wird demonstriert, wie es möglich ist, trotz der Nutzung spezifischer Charakteristika unterschiedlicher Zielsysteme abstrakte Entwicklungsmodule zu definieren, die allgemeine Eigenschaften verteilter Applikationen unterstützen. Dieser Ansatz ermöglicht es z.B., den allgemeinen Rahmen mit wenig Aufwand um die Unterstützung von CORBA zu erweitern, indem man auf ein allgemeines Erweiterungsmodul für verteilte Informations- und Kontrollsysteme zurückgreift. Zusätzlich erleichtert er die Entwicklung von heterogenen Applikationen, welche verschiedene Zielsysteme gleichzeitig benutzen.

In dem folgenden Kapitel 2 werden zunächst die Anforderungen konkretisiert, die an einen derartigen Rahmen für eine Koordinationssprache zu stellen sind. Bezüglich dieser Anforderungen wird daraufhin der aktuelle Stand der Forschung dargestellt. Dann wird in Kapitel 4 in knapper Form der wissenschaftliche Beitrag dieser Arbeit formuliert. In diesem Kapitel wird dann auch die gesamte restliche Gliederung dieser Arbeit vorgestellt.