Entwicklung einer abstrakten Speicherkomponente für eine verteilte heterogene dynamische Infrastruktur in Java/CORBA

Torsten Fink, Michael M. Gutzmann, Torsten Wolf, Werner Erhard

Lehrstuhl für Rechnerarchitektur, Universität Jena, D-07740 Jena

Zusammenfassung Da die Entwicklung von Applikationen für verteilte heterogene dynamische Systeme komplexe Probleme birgt, existieren vielfältige Werkzeuge, die eine Infrastruktur zur Unterstützung des Entwicklers bilden. In dieser Arbeit wird eine Speicherkomponente als Bestandteil einer umfassenden Infrastruktur vorgestellt, die eine ortstransparente Datenhaltung mit abstrakten Datentypen ermöglicht. Zur Steigerung der Zugriffsgeschwindigkeit können lokale Kopien transparent erzeugt werden. Um einen schnellen Datentransport in heterogenen Netzwerken zu gewährleisten, wird zur Laufzeit aus einer Menge unterstützter Übertragungsprotokolle ein jeweils geeignetes gewählt.
Diese Speicherkomponente wurde unter Einsatz einer Kombination von Java und CORBA implementiert. Messungen an einem exemplarischen Testsystem offenbaren den auftretenden Organisationsaufwand.

1 Einführung

Die Programmierung verteilter heterogener dynamischer Systeme birgt vielfältige Probleme. Eine Applikation muß in einzelne Komponenten zerlegt werden, sogenannte Applikationskomponenten (AKn), die auf unterschiedlichen Rechnern parallel zusammenarbeiten, welche durch ein oder mehrere, evtl. unterschiedliche Netzwerke miteinander verbunden sind. Neben der Koordination der parallelen Kontrollflüsse ist auch der Zugriff auf die Daten zu organisieren.

Middlewaretools (MWT) stellen eine einfache Infrastruktur dar, um durch Bibliotheken und Hilfsprogramme die Programmierung verteilter Systeme zu vereinfachen. Während die Steuerung der Kontrollflüsse i.a. komfortabel unterstützt wird, beschränkt sich die Unterstützung der Datenflüsse meist auf einfache, direkte Übertragung von Parametern zwischen den AKn. Hier besteht Handlungsbedarf hinsichtlich der Entwicklung einer leistungsfähigeren Speicherkomponente.

Eine Speicherkomponente (SK) einer Infrastruktur, die eine umfassende Unterstützung bieten soll, muß nach unserer Meinung folgende Anforderungen erfüllen:

- *Abstraktion:* Um eine komfortable und fehlersichere Nutzung von Daten zu gewährleisten, sollte der Typ der Daten, also deren besondere Eigenschaften

beim Zugriff Berücksichtigung finden. Fehleranfällig wäre z.B. die Speicherung einer zweidimensionalen Matrix als Vektor von Bytes, so daß die AKn erst lokal eine Umformatierung durchführen müssen. Besser wäre der Zugriff auf einzelne Elemente über deren Indizes mittels eines speziellen Selektors. Dieser Ansatz gleicht dem der abstrakten Datentypen.
- *Erweiterbarkeit:* Da Applikationen i.a. auch neue Datentypen definieren, sollte die SK dynamisch Erweiterungen um neue Datentypen ermöglichen. Dies sichert eine flexible Benutzung der SK.
- *Ortstransparenz:* Für den Zugriff auf Daten sollte kein Wissen bzgl. des Ortes der nahesten Kopien nötig sein. Statt dessen sollte ein eindeutiger Bezeichner benutzt werden.
- *Zugriffseffizienz:* Ein- und Ausgabedaten müssen so gelagert werden, daß den AKn ein schneller Zugriff möglich ist. Aufgrund der Dynamik der hier betrachteten Systeme kann es erforderlich sein, die Daten zur Laufzeit umzuverteilen oder lokale Kopien zu erstellen.

Jedes MWT unterstützt die Möglichkeit, Daten explizit zwischen einzelnen AKn als Nachrichten zu übertragen. MWT's wie PVM oder MPI bieten vielfältige Bibliotheksfunktionen an, um Daten in Form von Nachrichten zu übertragen [1, 2].

Systeme, die auf dem Paradigma der entfernten Prozeduraufrufe basieren, wie z.B. SunRPC oder DCE [3, 4], erlauben es, Daten als Parameter in Prozeduraufrufen zu verschicken. Beide Ansätze zwingen den Applikationsprogrammierer allerdings dazu, die Verwaltung der Daten in bezug auf optimale Lagerung und Entlastung des Hauptspeichers durch partielles Auslagern selbst zu organisieren.

DCE bietet zusätzlich ein verteiltes Filesystem an mit ähnlichen Fähigkeiten wie AFS oder WebNFS [5, 6]. Dieser Ansatz ermöglicht durch dynamisches Caching eine hohe *Zugriffseffizienz* und durch eine abstrakte Verzeichnisstruktur *Ortstransparenz*. Allerdings ist der Grad der verfügbaren *Abstraktion* und *Erweiterbarkeit* gering, da alle Datentypen als Datei, also als geordnete Liste von Bytes, gespeichert werden müssen. Auch wenn Bibliotheken zur Verfügung stehen, die es erlauben, strukturierte Datentypen direkt als Datei zu laden oder zu speichern, so benötigt eine AK doch genaues Wissen über den internen Aufbau der Daten. Abstrakte Datentypen, die nur über bestimmte Methoden manipuliert werden können, oder aktive dynamische Daten sind so nicht möglich.

In dieser Arbeit stellen wir eine SK für eine verteilte heterogene dynamische Infrastruktur vor, welche sich an den oben formulierten Anforderungen orientiert. Sie ist Teil einer umfassenden abstrakten Infrastruktur, die mächtige Automatismen für eine effiziente Nutzung vorhandener Ressourcen enthält [7].

Daten werden in dieser SK als aktive, mit einem Bezeichner versehene Objekte, deren Methoden die Eigenschaften des Datentyps reflektieren, in das System integriert. Um eine hohe Zugriffseffizienz zu gewährleisten, werden die Daten, sofern nötig, kopiert. Hierbei werden unterschiedliche Übertragungsprotokolle, bzw. unterschiedliche Netzwerkleistungen berücksichtigt. Zur Implementierung dieser Komponente benutzten wir Java in Verbindung mit CORBA (Common Object Request Broker Architecture) [8].

Diese Arbeit gliedert sich wie folgt: In Abschnitt 2 beschreiben wir den Aufbau und die Funktionsweise der SK. Abschnitt 3 enthält experimentelle Ergebnisse zur Evaluierung des Organisationsaufwands. In Abschnitt 4 fassen wir unsere Erfahrungen zusammen, die wir bei dem kombinierten Einsatz von Java und CORBA gewonnen haben. Wir schließen mit Ergebnissen und Ausblick in Abschnitt 5.

2 Beschreibung der Speicherkomponente

Im folgenden werden die einzelnen Elemente der in dieser Arbeit entwickelten Speicherkomponente (SK) vorgestellt. Abbildung 1 verdeutlicht einige Aspekte der inneren Struktur der SK durch ein statisches Klassendiagramm, welches mittels der Unified Modeling Language (UML) entworfen wurde. Für eine detailliertere Beschreibung von UML wird auf [9, 10] verwiesen.

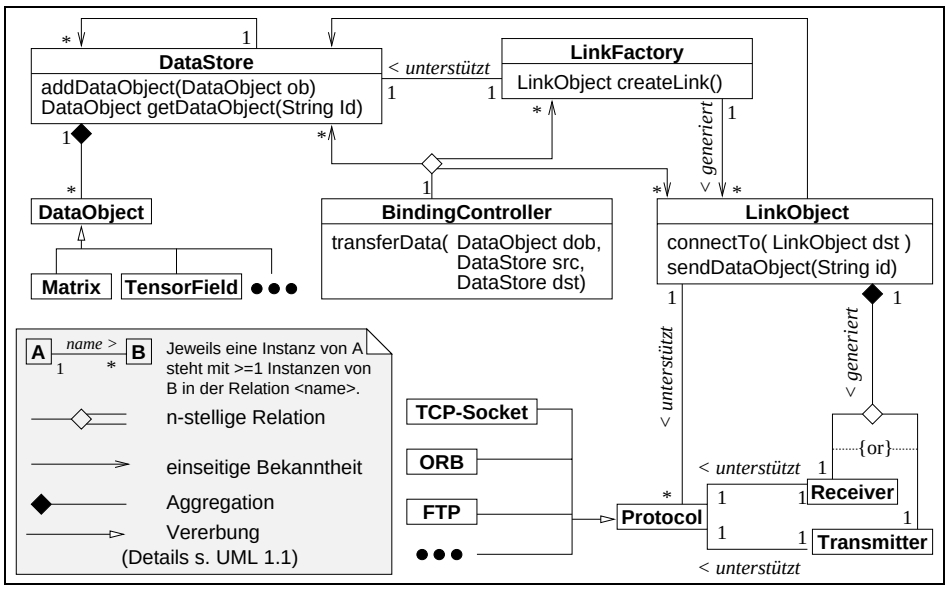

Abbildung1. Statisches Klassendiagramm in UML-Notation

Für jeden speziellen Datentyp wird eine entsprechende Klasse von `DataObject` (DO) abgeleitet, welche entsprechende Zugriffs- und Modifikationsoperatoren als Methoden bereitstellt. In Abbildung 1 sind exemplarisch die Typen `Matrix` und `TensorField` abgeleitet.

Datenobjekte werden in Objekten der Klasse `DataStore` (DS) gespeichert. DS-Objekte verwalten physikalischen Speicher, wie Hauptspeicher oder Festplattenplatz. Die einzelnen DS-Objekte bilden ein für den Applikationsprogrammie-

rer transparentes Netzwerk innerhalb des verteilten Systems. Um auf ein DO-Objekt $\mathcal{D}$ zugreifen zu können, benötigt eine AK nur ein beliebiges DS-Objekt $\mathcal{S}$ und den Bezeichner von $\mathcal{D}$. Mit der Methode `getDataObject` erhält die AK eine CORBA-Referenz auf die bzgl. $\mathcal{S}$ nächstliegende Kopie von $\mathcal{D}$.

Da die Geschwindigkeit des Zugriffs auf die Daten abhängig von dem Ort des entsprechenden DO-Objekts ist, können zur Optimierung DO-Objekte kopiert werden. Hierbei sollte gewährleistet sein, daß sich der zusätzliche Kopieraufwand durch die erhöhte Zugriffsgeschwindigkeit auszahlt. Dies ist Aufgabe der Applikation.

Die Konsistenz der Daten wird von den DO-Objekten selber gewährleistet. Hierbei werden Strategien eingesetzt, die sich an den jeweiligen Datentypen orientieren. Zur effizienten Datenübertragung einzelner DO-Objekte muß die vorhandene, meist heterogene und dynamische Netzwerkstruktur ausgenutzt werden. Das bedeutet einerseits, daß unterschiedliche Protokolle (TCP, ATM etc.) unterstützt werden müssen, andererseits ist auch die aktuelle Auslastung zu berücksichtigen.

Zur Unterstützung unterschiedlicher Protokolle beim Transfer von DO-Objekten dienen die Klassen `BindingController` (BC), `LinkFactory` (LF) und `LinkObject` (LO). Um ein DO-Objekt $\mathcal{D}$ von einem DS-Objekt $\mathcal{S}_1$ zu einem anderen DS-Objekt $\mathcal{S}_2$ zu übertragen, wird die Methode `transferData` des BC-Objekts $\mathcal{C}$, von dem nur eines im verteilten System existieren darf, mit entsprechenden Parametern aktiviert. Da Objekte selten kopiert werden, ist der durch diesen zentralen Ansatz entstehende Flaschenhals i.a. akzeptabel.

Jedem DS-Objekt ist ein LF-Objekt zugeordnet, das in der Lage ist, LO-Objekte zu produzieren. Ein LO-Objekt verbindet immer zwei DS-Objekte und unterstützt mehrere Protokolle. Soll nun $\mathcal{D}$ übertragen werden und sind $\mathcal{S}_1$ und $\mathcal{S}_2$ noch nicht durch zwei LO-Objekte $\mathcal{L}_1$, $\mathcal{L}_2$ verbunden, läßt $\mathcal{C}$ die den $\mathcal{S}_1$ und $\mathcal{S}_2$ zugeordneten LF-Objekte zwei LO-Objekte produzieren. Diese wählen ein unterstütztes und zu diesem Zeitpunkt optimales Protokoll aus und können nun $\mathcal{D}$ übertragen.

Dieser Ansatz erfüllt die in Abschnitt 1 aufgestellten Anforderungen:

- Durch Definition einer eigenen Objektklasse für jeden Datentyp mit entsprechenden Zugriffsmethoden wird sowohl *Abstraktion* als auch *Erweiterbarkeit* gewährleistet.
- Der Zugriff auf die Daten über einen Bezeichner ermöglicht *Ortstransparenz*.
- Das Anlegen lokaler Kopien und das Ausnutzen geeigneter Übertragungsprotokolle erhöht die *Zugriffseffizienz*.

Offen ist die Frage, wie gut diese Anforderungen erfüllt sind. Diese Frage wird hier nur teilweise beantwortet und ist Gegenstand aktueller und zukünftiger Forschung.

3 Experimentelle Leistungsmessungen

Eine Faustregel besagt, je höher die Komplexität eines Systems ist, desto höher ist der induzierte Organisationsaufwand und desto niedriger damit dessen Ef-

fizienz. Um den Organisationsaufwand abzuschätzen, der bei der Nutzung des Netzwerkes durch die Speicherkomponente auftritt, haben wir Messungen durchgeführt. Hierbei wurden Daten unterschiedlicher Größe innerhalb verschiedener Umgebungen verschickt und die Übertragungszeiten gemessen.

Um das Verhalten für kleine Datengrößen zu beobachten, wurden Daten in den Größen $\{2^i \mid 0 \leq i \leq 10\}$ verschickt. Zur Evaluierung des Verhaltens bei großen Datenmengen wurden die Größen $\{i * \frac{(2^{20}-2^{10})}{10} + 2^{10} | 1 \leq i \leq 10\}$ gewählt. Zur Abschätzung des Organisationsaufwands wurden die folgenden Umgebungen betrachtet:

- Die Daten wurden als Parameter für Methoden in Java mittels CORBA verschickt. Hierbei wird die hier vorgestellte Speicherkomponente nicht benutzt. (Diese Umgebung wird im folgenden mit *Java/CORBA* referenziert)
- Die Daten werden zwischen zwei DS-Objekten verschickt, die durch LO-Objekte verbunden sind. Als Protokoll wird TCP eingesetzt. (*SK-TCP*)
- Die Daten werden zwischen zwei DS-Objekten verschickt, die durch LO-Objekte verbunden sind. Als Protokoll werden ähnlich wie bei Java/CORBA die Daten als Parameter mit CORBA verschickt. (*SK-CORBA*)

Für jede Paketgröße wurden 100 Messungen auf einem 10 MBit Ethernet unter Benutzung von Sun-Ultra 1 Workstations (143 MHz, 64 MB Ram) durchgeführt. Um Verzerrungen aufgrund von Schwankungen in der Netzwerkleistung durch parallel laufende Anwendungen zu dämpfen, wurden nur die besseren 50-Perzentil betrachtet und von diesen der Mittelwert berechnet. Abbildung 2 zeigt die hierbei erzielten Ergebnisse. In der rechten, linearen Darstellung, welche die Ergebnisse für große Datenmengen zeigt, ist zusätzlich die theoretisch minimale Übertragungszeit beim Einsatz eines 10 MBit[1] Netzwerks aufgetragen.

Bei den kleinen Datengrößen besteht die Übertragungszeit hauptsächlich aus dem Organisationsaufwand des Übertragungsprotokolls. Dies gilt sowohl für die Übertragung in Java/CORBA, als auch für die Nutzung der SK. Wie zu erkennen ist, liegt diese Aufsetzzeit für die SK bei ungefähr 75 ms, während die direkte Nutzung von Java/CORBA nur 4 ms benötigt.

Ab einer Datengröße von 200 KB bietet SK-TCP eine höhere Leistung als die direkte Benutzung von CORBA. Bei 1 MB wird eine um etwa 15% höhere Übertragungsleistung erzielt. Dies ist vor allem deshalb eine deutliche Steigerung, da die Übertragung von CORBA selbst direkt auf TCP aufsetzt. Bei Unterstützung spezieller, leitungsorientierter Protokolle, wie z.B. ATM, die nicht von CORBA eingesetzt werden, sind weitere Leistungssteigerungen zu erwarten.

Die direkte Benutzung von CORBA stellt eine obere Leistungsgrenze für die Benutzung von CORBA innerhalb der SK dar. Wie die Messungen zeigen, ist der zusätzliche Organisationsaufwand aber schon bei einer Datengröße ab 500 KB praktisch irrelevant.

Zusammenfassend kann man sagen, daß der hier vorgestellte allgemeine Ansatz für größere Datenmengen eine signifikante Steigerung der Übertragungslei-

[1] Hierbei wurde davon ausgegangen, daß ein 10 MBit Netz eine Datentransferrate von $\frac{10^6}{8} \frac{Byte}{s}$ besitzt.

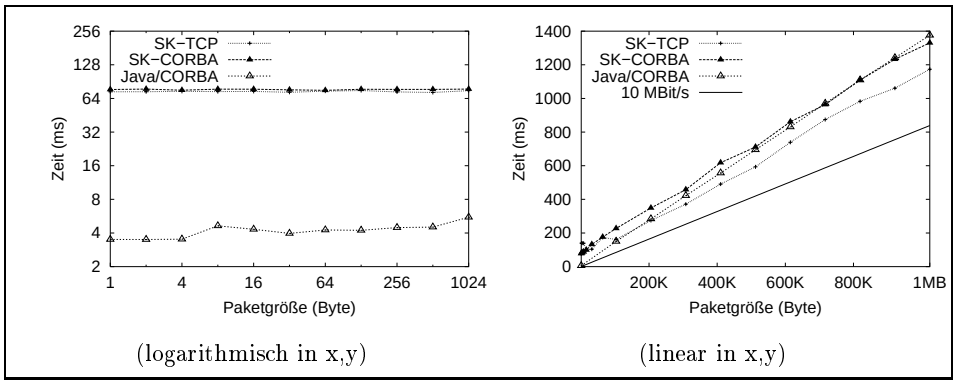

Abbildung2. Messungen der Übertragungszeiten für unterschiedliche Datenmengen zur Abschätzung des Organisationsaufwands

stung ermöglicht. Die eingesetzten Werkzeuge Java und CORBA beeinträchtigen allerdings die Leistung deutlich, so daß eine um ungefähr 30% unter dem Optimum liegende Übertragungsgeschwindigkeit erzielt wird. Frühere Messungen haben gezeigt, daß die Ursache hier bei Java zu suchen ist, da das gleiche CORBA-System mit C++ deutlich bessere Übertragungsleistungen lieferte [11].

4 Erfahrungen mit Java/CORBA

Zur Implementierung der Speicherkomponente wurde das Java-Development-Kit von SUN in der Version 1.1.5 und das CORBA System ORBacus von Object-Oriented-Concepts [12] eingesetzt. Diese Kombination stellte eine stabile und trotz fehlender graphischer Benutzeroberfläche ausreichend komfortable Entwicklungsplattform dar.

Die Übertragung der Datenobjekte war mit Hilfe des Serialisierungskonzepts von Java einfach zu implementieren. Auch die Unterstützung nebenläufiger Objekte durch Threads erwies sich als angenehm. Für eine industrielle Nutzung des Systems sind weitere Sicherheitsanforderungen an die Datenobjekte zu stellen, die ausführbaren Code enthalten. Hierfür bietet sich das Sandkasten-Prinzip von Java an.

Die Einschränkung von Java auf Einfachvererbung stellt allerdings in Kombination mit CORBA ein Problem dar. Jedes Java-Objekt, das über CORBA dem verteilten System Methoden anbietet, muß von einem automatisch generierten CORBA-Objekt abgeleitet werden. Die hier vorgestellten Klassen bilden aber eine eigene Hierarchie und sind alle von einer Basisklasse abgeleitet. Diese Hierarchie muß durch das Interface-Konzept in Java modelliert werden. Dadurch führt, auch bei Verwendung von Hilfsklassen, jede Erweiterung der Basisklasse zu Änderungen an allen anderen Klassen, die manuell durchgeführt werden müssen.

Das Fehlen eines Destruktors in Java erleichtert zwar prinzipiell die Programmierung führt aber in diesem Projekt mehrmals dazu, daß das gesamte Programm genauestens nach Verweisen abgesucht werden mußte, damit der Garbage-Collector den Speicherplatz eines bestimmten Objekts wieder frei gab. Da einige Objekte der hier vorgestellten Speicherkomponente für die gesamte Laufzeit der Infrastruktur aktiv sind, ist die Verwaltung des Speichers besonders wichtig. Desweiteren wurde die Suche durch den automatisch von CORBA generierten Programmcode erschwert.

Die oben beschriebenen Messungen bescheinigen Java eine vergleichsweise schlechte Übertragungsleistung. Ob diese akzeptabel ist, muß für die jeweilige Applikation überprüft werden.

Zusammenfassend ist zu sagen, daß Java mächtige Konzepte anbietet, welche die Implementierung des hier vorgestellten Systems in kurzer Zeit ermöglicht. Auch wenn Detailprobleme stören, existieren zur Zeit wohl nur wenige vergleichbare Entwicklungsumgebungen. Die mit Java erzielbare Leistung ist für unseren prototypischen Aufbau einer Infrastruktur ausreichend.

5 Ergebnisse und Ausblick

Wir haben eine Speicherkomponente als Teil einer umfassenden Infrastruktur zur Unterstützung der Entwicklung von Applikationen für verteilte heterogene dynamische Systeme objektorientiert konzipiert und mit Java und CORBA prototypisch implementiert. Diese Speicherkomponente erfüllt die gestellten Anforderungen nach einer *Abstraktion* der Daten, *Erweiterbarkeit*, *Ortstransparenz* und *Zugriffseffizienz*. Daten können intern repliziert und umverteilt werden. Hierfür werden aus mehreren Netzwerkprotokollen dynamisch geeignete ausgewählt.

Um den zur Realisierung der Anforderungen zusätzlich notwendigen Organisationsaufwand abzuschätzen, wurden Übertragungszeiten für Daten unterschiedlicher Größe einmal direkt mit Java/CORBA und einmal unter Benutzung der Speicherkomponente in einer Testumgebung bestehend aus SUN-Workstations verbunden durch ein 10 MBit Ethernet gemessen. Dabei ergab sich eine Aufsetzzeit von ungefähr 75 ms. Es zeigte sich, daß ab 200 KB der Einsatz der Speicherkomponente zu höheren Übertragungsraten als die direkte Nutzung von CORBA führte.

Als nächsten Schritt werden wir den Einsatz der Speicherkomponente innerhalb von Applikationen evaluieren. Schwerpunkte dabei sind effiziente Datenlagerung, Datentypen und Konsistenzprotokolle.

Literatur

1. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek und V. Sunderam. *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994. URL: `http://www.netlib.org/pvm3/book/pvm-book.html`.

2. M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker und J. Dongarra. *MPI: the complete reference*. MIT Press, Cambridge, MA, USA, 1996.
3. Sun Microsystems. RPC: Remote Procedure Call Protocol Specification Version 2; RFC1058. *Internet Request for Comments*, (1057), Juni 1988.
4. W. Rosenberry, D. Kenney und G. Fisher. *OSF Distributed Computing Environment: Understanding DCE*. O'Reilly and Associates, 1993.
5. A. Spector und M. Kazar. Wide Area File Service and the AFS Experimental System. *Unix Review*, 7(3), 1989.
6. B. Callaghan. RFC 2055: WebNFS Server Specification, Oktober 1996. URL: `ftp://ftp.internic.net/rfc/rfc2055.txt`.
7. T. Fink, M. M. Gutzmann und S. Kindermann. A Dynamic Metacomputing Framework based on Distributed Objects. *Berichte zur Rechnerarchitektur*, 3(7), Dezember 1997. Universität Jena, ISSN 0949-3042. URL: `http://www2.informatik.uni-jena.de/FG-PVS/Documents/TBe-97-BR-3:7.html`.
8. J. Siegel. *CORBA Fundamentals and Programming*. Jon Wiley & Sons, 1996.
9. P.-A. Muller. *Instant UML*. Wrox Press, 1997.
10. R. S. Corporation. UML-Notation Guide, 1997. URL: `http://www.rational.com/uml/`.
11. T. Fink, C. Rahn, M. M. Gutzmann, O. Preusche und W. Erhard. Comparing the Performance of MPI, PVM, and CORBA on Ethernet LANs. *Berichte zur Rechnerarchitektur*, 3(4), November 1997. Universität Jena, ISSN 0949-3042. URL: `http://www2.informatik.uni-jena.de/FG-PVS/Documents/TBe-97-BR-3:4.html`.
12. ORBacus Home Page. URL: `http://www.ooc.com/ob.html`.