Effizientes RMI für Java

Christian Nester, Michael Philippsen und Bernhard Haumacher

Universität Karlsruhe, Institut für Programmstrukturen und Datenorganisation
Am Fasanengarten 5, 76128 Karlsruhe
http://wwwipd.ira.uka.de/JavaParty/

Zusammenfassung Der entfernte Methodenaufruf gängiger Java-Implementierungen (RMI) ist zu langsam für Hochleistungsanwendungen, da RMI für Weitverkehrskommunikation entworfen wurde, auf einer langsamen Objektserialisierung aufbaut und keine Hochgeschwindigkeitsnetze unterstützt.
Das Papier zeigt ein wesentlich schnelleres RMI mit effizienter Serialisierung in reinem Java ohne native Methodenaufrufe. Es kann auf jeder Plattform anstelle des normalen RMI aktiviert werden. Darüberhinaus ist das neuentworfene RMI auch für Hochgeschwindigkeitsnetze ohne TCP/IP-Protokoll ausgelegt und funktioniert sogar auf Rechnerbündeln mit heterogener Protokollarchitektur. Als Nebenprodukt wird eine Sammlung von RMI-Benchmarks vorgestellt.
Bei durch Ethernet verbundenen PCs spart die verbesserte Serialisierung in Kombination mit dem neuentworfenen RMI im Median 45% (maximal 71%) der Laufzeit eines entfernten Methodenaufrufs ein. Auf unserem ParaStation-Rechnerbündel spart man im Median 85% (maximal 96%) der Zeit. Ein entfernter Methodenaufruf wird damit innerhalb von derzeit 80μs möglich (im Vergleich zu 1450μs im Standardfall).

1 Einleitung

Die Aktivitäten des Java-Grande Forums [6, 15] und Vergleichsstudien [13] belegen, daß ein wachsendes Interesse an Java für Hochleistungsanwendungen besteht, die Parallelrechner als Plattform benötigen. Aber während Java angemessene Mechanismen für internetweite Kommunikation bereitstellt, ist RMI [18] für Umgebungen mit kurzer Latenz und großer Bandbreite (z.B. Bündel von Arbeitsplatzrechnern, IBM SP/2) zu ineffizient.

1.1 Aufschlüsselung der Kosten von RMI

In einem dreistufigen Meßaufbau untersuchen wir die Zeit für einen entfernten Methodenaufruf mit drei verschiedenen Argumenttypen. Als erstes wird der Zeitbedarf eines entfernten Methodenaufrufs `ping(obj)` gemessen, der lediglich das übergebene Objekt wieder zurückliefert. Ein Teil der Zeit wird für die Serialisierung der Argumente und die Netzwerkkommunikation aufgewendet. Diese Zeiten werden separat gemessen, indem erstens ein Objekt des entsprechenden Typs über einen Socket-Objektstrom zwischen zwei Rechnern ausgetauscht wird

Tabelle1. Zeit für `ping(obj)` (μs) über RMI (=100%), Socket-Kommunikation und reine JDK-Serialisierung. Das Argument `obj` besitzt entweder 32 `int` Werte, 4 `int` Werte und 2 `null` Zeiger oder es ist ein balancierter Binärbaum bestehend aus 15 Knoten mit je 4 `int` Werten.

	μs pro Objekt	32 `int`		`4int 2null`		tree(15)		Meßgrundlage
PC	RMI `ping(obj)`	2287		1456		3108		zwei 350MHz Pentium II, Windows NT 4.0, verbunden über Ethernet, vom restl. Netz isoliert, Java 1.2 mit JIT
	`socket(obj)`	1900	83%	1053	72%	2528	81%	
	`serialize(obj)`	840	37%	368	25%	1252	40%	
DEC	RMI `ping(obj)`	7633		4312		14713		8er-Bündel von 500MHz Alphas, verbunden über FastEthernet oder ParaStation, Java 1.1.6 mit JIT
	`socket(obj)`	6728	88%	2927	68%	12494	85%	
	`serialize(obj)`	4332	57%	1724	40%	9582	65%	

und zweitens nur Serialisierung und Deserialisierung des Argumentobjekts ohne Kommunikation durchgeführt wird.

Tabelle 1 zeigt die Resultate; andere Objekttypen verhalten sich ähnlich. Der durch RMI verursachte Mehraufwand ist konstant, so daß bei größeren Objekten Serialisierung und Kommunikation dominieren. Als Daumenregel läßt sich ableiten, daß die Serialisierung mindestens 25% und mit wachsender Objektgröße bis zu 65% der Zeit verschlingt. Der Prozentsatz ist bei langsamen Java-Implementierungen noch größer. RMI verursacht einen Mehraufwand zwischen 0,4 und 2,2ms.

1.2 Gliederung des Beitrags

Wir arbeiten an allen drei Bereichen (Serialisierung, RMI und Netzwerksubsystem), um bestmögliche Geschwindigkeit zu erreichen. Nach einer Diskussion verwandter Arbeiten in Abschnitt 2 zeigt Abschnitt 3 die zentralen Ideen einer verbesserten Serialisierung. Abschnitt 4 bespricht den Entwurf unseres schlanken RMI, daß ausschließlich in Java implementiert und daher voll portabel ist. Abschnitt 5 stellt kurz das auf Myrinet-Hardware basierende ParaStation Netzwerk vor, welches wir im Austauch für Ethernet eingesetzt haben, um die Unabhängigkeit unseres RMI von TCP/IP-basierten Netzwerken zu demonstrieren. Der letzte Abschnitt 6 diskutiert quantitative Ergebnisse.

2 Verwandte Arbeiten

Thiruvathukal et al. [16] experimentierten mit expliziten Versenderoutinen, allerdings können wir zeigen, daß nur durch enge Zusammenarbeit mit der Pufferverwaltung erhebliche Verbesserungen erreicht werden.

Manta [17] ermöglicht einen effizienten entfernten Methodenaufruf (35μs für einen entfernten Null-Aufruf) allerdings nicht durch ein effizientes RMI-Paket, sondern indem eine Untermenge von Java direkt in nativen Code für ein PC-Bündel übersetzt wird. Zur Objektserialisierung werden explizite Versenderoutinen erzeugt, die die Kenntnis des Speicherlayouts der Objekte ausnutzen. Es

ist unklar, welche Geschwindigkeit Manta bei der Serialisierung allgemeiner Objektstrukturen wie z.B. Graphen erzielt. Unsere Arbeit basiert hingegen ganz auf Java, und kann leicht auf jeder Plattform benutzt werden.

Breg et al. [2] haben eine Teilmenge von RMI auf das Nexus-Laufzeitsystem portiert. Unser Entwurf kann leichter auf andere Zielarchitekturen angepaßt werden und erreicht bessere Geschwindigkeiten.

Horb [5] und Voyager [11] sind alternative Technologien für verteilte Objekte in Java, die sich im Gegensatz zu unserer Arbeit weder einfach gegen RMI austauschen lassen, noch für Hochleistungsrechnen ausgelegt sind.

Es gibt weitere Ansätze zur Verwendung von Java auf Rechnerbündeln, bei denen aber Objektserialisierung keine Rolle spielt. *Java/DSM* [20] implementiert eine JVM aufbauend auf Treadmarks [8]. Dort ist keine explizite Kommunikation nötig, da alles transparent vom unterliegenden DSM erledigt wird. Es liegen uns keine Angaben über die Geschwindigkeit vor.

Ein dazu orthogonaler Ansatz vermeidet Objektserialisierung durch Objekt-Caching [9]. Objekte, die nicht verschickt werden, ziehen auch keinen Serialisierungsaufwand nach sich.

Es sind uns keine anderen Benchmarksammlungen für RMI bekannt, obwohl die meisten der obengenannten Gruppen quantitative, aber nicht direkt miteinander vergleichbare Ergebnisse veröffentlicht haben.

3 Effiziente Serialisierung

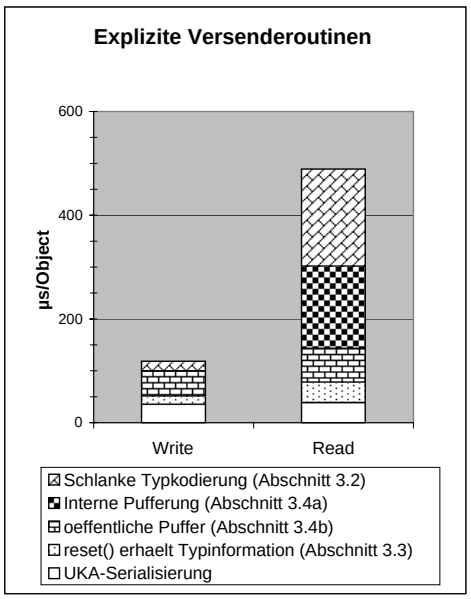
3.1 Grundlagen der Objektserialisierung

Als Kopie übergebene Objekte und primitive Typen werden durch die Serialisierung in eine Repräsentation als Byte-Feld überführt. Diese Kodierung wird auf der Empfängerseite in eine Kopie des Objektgraphen zurückverwandelt. Dabei können sogar zyklische Strukturen kopiert werden, da mehrfache Referenzen auf dasselbe Objekt mittels einer Hashtabelle aufgelöst werden. Bei jedem neuen Methodenaufruf muß diese Tabelle gelöscht werden, da sich der Zustand der übertragenen Objekte geändert haben könnte.

Die Serialisierung wird mittels dynamischer Typintrospektion für alle Objektklassen von einer Bibliotheksklasse erledigt. Der Programmierer kann jedoch eigene Serialisierungsmethoden schreiben (`writeObject` oder `writeExternal`), falls spezifischere Operationen notwendig sind oder eine bessere Geschwindigkeit erzielt werden soll. Manta erzeugt solche Routinen im Zuge der nativen Übersetzung. Auch wir arbeiten an einem Werkzeug, das effiziente Versenderoutinen für existierende Klassen generiert.

Im Rest des Papiers meint der Ausdruck „Serialisierung“ das Schreiben und Lesen von Byte-Repräsentationen im allgemeinen. Die im JDK erhältliche Version der Serialisierung soll „JDK-Serialisierung“ und unsere verbesserte Version soll „UKA-Serialisierung“ heißen.

Die Abschnitte 3.2 bis 3.4 diskutieren die gemusterten Bereiche der Balken in Abbildung 1. Die UKA-Serialisierung kann alle diese Zeitanteile einsparen. Die



Die kompletten Balken geben die von der JDK-Serialisierung benötigte Zeit an, um ein Objekt mit 32 `int` Werten unter Zuhilfenahme von expliziten Versenderoutinen zu schreiben bzw. zu lesen. Unsere Umgebung bestand aus JDK 1.2beta3 mit eingeschaltetem JIT auf einer Sun Ultra Sparc IIi unter Solaris 2.6. (Ähnliche Ergenbisse haben wir auf PC und DEC mit anderen JDK-Versionen gemessen.) Die JDK-Serialisierung bietet zwei verschiedene Protokollversionen an. Obwohl die Version 2 die Standardeinstellung ist, benutzt RMI die etwas schnellere Protokollversion 1. Unsere Vergleichsmessungen benutzen ebenfalls Version 1. Die gemusterten Teile stehen für die Zeitanteile, die durch die einzelnen Optimierungen eingespart wurden. Auf diese Optimierungen wird in den Abschnitten 3.2 bis 3.4 näher eingegangen. Durch Verwendung aller dieser Optimierungen bleibt nur noch die Zeit übrig, die durch die kleinen weißen Kästen ganz unten angedeutet wird. Ähnliche Ergebnisse erhält man auch für andere Objekttypen, siehe Tabelle 2.

Abbildung1. Serialisierungszeitanteile bei expliziten Versenderoutinen.

von der JDK-Serialisierung benötigten Zeiten, repräsentiert durch die komplette Säule, schrumpfen dabei auf die Anteile zusammen, die durch die kleinen weißen Kästen ganz unten angedeutet sind. Einzelheiten der UKA-Serialisierung und Vorschläge für weitere Verbesserungen findet man in [4].

3.2 Schlanke Typkodierung

Obwohl aus der Methodendeklaration die Parametertypen eines entfernten Methodenaufrufs statisch bekannt sind, muß beim Aufruf dennoch Typinformation übertragen werden, da zur Laufzeit beliebige Untertypen des statischen Parametertyps als Argument übergeben werden können. Allerdings ist für parallele Java-Programme auf Rechnerbündeln und DMPs eine äußerst schlanke Typinformation ausreichend, da keine Persistenz benötigt wird, die über viele Java-Versionen hinweg reichen muß, und alle Knoten Zugriff auf die selben Klassendateien über ein gemeinsames Dateisystem besitzen.

Die UKA-Serialisierung überträgt nur den vollen Klassen- und Paketnamen. Es sind aber noch kürzere Kodierungen denkbar. Die schlanke Typkodierung beschleunigt die Geschwindigkeit der Serialisierung deutlich, siehe Abbildung 1. Es zeichnet sich ab, daß Sun unsere Idee schlanker Typkodierung aufgreifen und in der nächsten JDK-Version optional zur Verfügung stellen wird.

3.3 Zwei Reset-Varianten

Um beim entfernten Methodenaufruf Kopiersemantik für die Argumente zu erreichen, muß jeder entfernte Methodenaufruf mit einer leeren Tabelle zur Zyklenerkennung beginnen. Nur so werden Objekte, die bereits in einem früheren Aufruf übertragen wurden, erneut mit ihrem aktuellen Zustand übertragen.[1] Die derzeitige RMI-Implementierung legt dazu für jeden entfernten Methodenaufruf ein neues Serialisierungsobjekt an. Alternativ könnte die `reset()`-Methode des Serialisierungsobjekts aufgerufen werden. Beide Ansätze haben den Nachteil, daß mit dem Löschen der bereits übertragenen Objekte auch alle Information über schon übertragene *Typen* gelöscht wird.

Die UKA-Serialisierung bietet daher eine neue Variante der `reset()`-Methode an, die zwar die Tabelle der bereits übertragenen Objekte löscht, die *Typ*information aber unverändert läßt. Die gepunkteten Bereiche in Abbildung 1 zeigen die dadurch eingesparte Zeit.

3.4 Verbesserte Pufferung

a) Interne statt externer Pufferung. Auf der Empfängerseite führt die JDK-Serialisierung keine eigene Pufferung durch sondern vertraut auf den unterliegenden `BufferedStream`. Dessen Pufferung ist allgemein und kann kein Wissen über die Länge der Byte-Repräsentation übertragener Objekte nutzen.

Die UKA-Serialisierung puffert selbst und kann Wissen über das Leitungsformat ausnutzen. Sofern möglich, werden immer ganze Objekte auf einmal gelesen und dadurch unnötige Abfragen auf Unter- oder Überlauf vermieden. Die mit 3.4a markierten Bereiche in Abbildung 1 zeigen den Effekt interner Pufferung.

b) Öffentliche statt privater Puffer. Die externe Pufferung erzwingt das Beschreiben der Puffer über den Umweg von Methodenaufrufen. Da die UKA-Serialisierung ihren eigenen Puffer implementiert, kann den expliziten Versenderoutinen direkter Zugriff darauf gewährt und eine Schnittstelle zum Puffermanagement angeboten werden. An dieser Stelle tauschen wir Modularität und Sicherheit des ursprünglichen Entwurfs gegen eine Geschwindigkeitssteigerrung ein. Die 3.4b markierten Bereiche zeigen die durch direktes Beschreiben der Puffer gewonnene Zeit.

3.5 Quantitative Verbesserungen

Tabelle 2 zeigt den Effekt der UKA-Serialisierung bei verschiedenen Objekttypen. Zur Serialisierung und Deserialisierung eines Objekts mit 32 `int` Werten werden anstatt 66+354=420μs auf einem PC (2166μs auf einer DEC) mit der UKA-Serialisierung nur noch 5+15=20μs (156μs) benötigt. Die Einsparungen bei tiefen Strukturen sind etwas geringer (ca. 80%), weil der zur Zyklenerkennung notwendige Aufwand nicht reduziert werden kann.

[1] Objekt-Caching kann häufige Neuübertragung u.U. vermeiden, siehe Abschnitt 2.

Tabelle2. Verbesserungen bei verschiedenen Objekttypen analog Tabelle 1.

	μs pro Objekt	32 `int`		4`int` 2`null`		tree(15)		Meßgrundlage
		w	r	w	r	w	r	
PC	JDK-Serialisierung	66	354	31	153	178	448	siehe Tab. 1
	UKA-Serialisierung	5	15	3	11	41	107	
Verbesserung %		**92**	**96**	**90**	**93**	**77**	**76**	
DEC	JDK-Serialisierung	700	1466	271	591	1643	3148	siehe Tab. 1
	UKA-Serialisierung	54	102	32	71	216	397	
Verbesserung %		**92**	**93**	**88**	**88**	**87**	**87**	

4 KaRMI: Effizientes RMI

Unter dem Namen KaRMI haben wir das RMI von JDK 1.2 neu entworfen und neu implementiert. Die Idee dabei war, ein schlankes, schnelles Rahmenwerk bereitzustellen, das durch Spezialmodule zu ergänzen ist. Dies können sowohl optimierte Komponenten mit voller RMI-Funktionalität, als auch solche sein, die Teile der RMI-Funktionalität gegen eine Geschwindigkeitssteigerrung eintauschen. Ebenso sind Komponenten möglich, die spezielle Kommunikationshardware unterstützen oder angepaßte Speicherbereiniger nutzen.

Die folgenden Abschnitte bieten einen Überblick über KaRMI und die Gründe für die Geschwindigkeitssteigerung; sie diskutieren die Unterstützung für Kommunikationshardware ohne TCP/IP-Protokoll und den in KaRMI verfolgten Ansatz alternativer verteilter Speicherbereinigungsalgorithmen.

4.1 Saubere Schnittstellen zwischen Entwurfsschichten

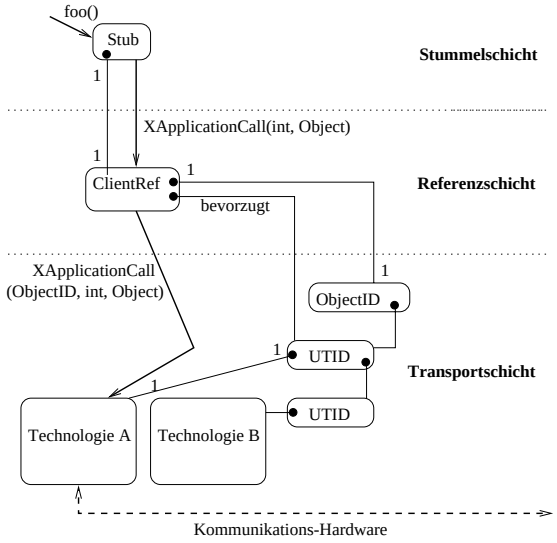
Wie beim Entwurf des offiziellen RMI gibt es bei KaRMI drei Schichten (Stellvertreter/Skelett-, Referenz- und Transportschicht).[2] Im Gegensatz zur offiziellen Version bietet KaRMI klar dokumentierte Schnittstellen zwischen den Schichten, was zwei entscheidende Vorteile hat: Erstens einen Geschwindigkeitsvorteil, weil KaRMI bei einem entfernten Methodenaufruf mit lediglich zwei Methodenaufrufen an den Schnittstellen auskommt und keine temporären Objekte erzeugt. Zweitens können weitere Implementationen für Referenz- und Transportschicht einfach hinzugefügt werden (siehe Abschnitt 4.3).

Der Kardinalfehler des RMI-Designs war das Offenlegen der Socket-Schnittstelle für die Anwendung. Dadurch wird z.B. das Exportieren von Objekten an festgelegten Ports möglich. Wenn Sokets aber für die Anwendung sichtbar sind,

[2] Für die nicht mit der Architektur von RMI vertrauten Leser: Die *stub/skeleton Schicht* überführt einen spezifischen Methodenaufruf an die generische Aufrufschnittstelle der Referenzschicht. Auf der Seite des Gerufenen erfolgt der Rückaufruf an die Anwendung. Die *Referenzschicht* ist für die Aufrufsemantik zuständig. Hier wird unterschieden, ob es sich um einen einfachen oder einen replizierten Server handelt. Die *Transportschicht* realisiert die eigentliche Netzwerkkommunikation auf unterster Ebene.

muß jede RMI-Implementierung Sockets in der Transportschicht verwenden – auch dann, wenn das unterliegende Netzwerk Sockets nicht gut unterstützt. In diesem Fall müßte in der Transportschicht der RMI-Implementierung das TCP/IP-Protokoll nachgebildet werden, was für Hochleistungsanwendungen nicht nur unnötig, sondern auch zu langsam ist. Um Hochgeschwindigkeitsnetze auszunutzen, besteht der einzige Ausweg darin, die Socket-Funktionalität aus dem RMI-Design herauszunehmen.

4.2 Geschwindigkeitsverbesserungen



Die drei Schichten von KaRMI: Ein entfernter Methodenaufruf wird vom Objektstellvertreter zum entsprechenden Referenzobjekt (`ClientRef`) weitergerreicht. Das Referenzobjekt besitzt einen Identifikator `ObjectID`, der selbst auf eine Kette von `UTID`s zeigt, wobei jeder `UTID`-Identifikator auf das entsprechende Technologieobjekt verweist, welches für die Kommunikation mit der Netzwerkhardware zuständig ist.

Abbildung2. Entfernter Methodenaufruf in KaRMI

Abbildung 2 zeigt die Aufrufseite eines entfernten Methodenaufrufs. Ein am Stellvertreter ankommender Methodenaufruf wird durch einen Aufruf von `XApplicationCall(int, Object)` zum Referenzobjekt (`ClientRef`) weitergeleitet. Normalerweise ist dabei das erste Argument die Methodennummer, das zweite die Argumentliste des Aufrufs; andere Semantiken sind denkbar. Indem mehrere Versionen von `XApplicationCall()` mit unterschiedlichen Rückgabetypen (verschlüsselt im `X` des Methodennamens) angeboten werden, vermeidet KaRMI im Gegensatz zur offiziellen RMI-Version, die nur `Object` als Rückgabetyp kennt, teures Verpacken primitiver Werte. Auf der Empfängerseite wird die entsprechende Version von `XdoApplicationCall(..)` aufgerufen.

Das Referenzobjekt ist für die Adressierung des entfernten Objekts zuständig und speichert dazu eine oder bei Replikation mehrere `ObjectID`s. Wie weiter unten genauer ausgeführt wird, ruft das Referenzobjekt die Methode

`XApplicationCall(ObjectID, int, Object)` eines geeigneten Technologieobjekts für die Netzwerkkommunikation auf. Einzelheiten der Adressierung und der Technologieobjekte werden in Abschnitt 4.3 erklärt.

Wo KaRMI für jeden entfernten Methodenaufruf ein einzelnes Objekt anlegt, erzeugt RMI ungefähr 25 Objekte, plus je ein Objekt pro Argument und Rückgabewert. Unsere verbesserte Implementierung vermeidet die häufige und langsame Objekt-Allokation und führt daher zu einer höheren Geschwindigkeit. Ferner realisiert KaRMI folgende Verbesserungen:

• RMI benutzt teure Aufrufe nativen Codes und den teuren Mechanismus zur dynamischen Typintrospektion, um mit primitiven Typen zurechtzukommen.[3] Pro entferntem Methodenaufruf werden fünf, pro Argument und Rückgabewert je zwei native Methoden aufgerufen. KaRMI benutzt hingegen native Methodenaufrufe lediglich zur Kommunikation mit dem Gerätetreiber.

• Im Gegensatz zu RMI erkennt KaRMI entfernte Objekte in derselben JVM und kappt die Aufrufkette beim Zugriff darauf. Natürlich müssen Argumente trotzdem kopiert werden, um die Semantik eines entfernten Aufrufs zu erhalten. Allerdings spart man dadurch einen Kontextwechsel und die Verwendung des „loopback"-Gerätetreibers.

• Die Architekten von RMI benutzen Hashtabellen auch dort, wo andere Datenstrukturen schneller sein könnten oder sie ganz vermeidbar gewesen wären. Obwohl das Löschen von Hashtabellen zeitaufwendig ist, löscht RMI Hashtabellen sogar nach der letzten Verwendung.

4.3 Technologieobjekte kapseln Netzwerkhardware

KaRMI unterstützt im Gegensatz zu RMI Netzwerke ohne TCP/IP-Protokoll und erlaubt die Verwendung mehrerer Netzwerkzugänge auf demselben Knoten. Zu diesem Zweck führt KaRMI den Begriff der Netzwerktechnologie ein. Bei der Initialisierung wird für jede auf einem Knoten `A` verfügbare Netzwerkhardware ein Technologieobjekt `T` angelegt. Dessen `UTID`-Objekt enthält alle Informationen, die ein anderer Knoten braucht, um `A` über `T` zu erreichen. Z.B. existieren auf einem Knoten, der sowohl über Ethernet- als auch über ParaStation-Hardware verfügt, zwei Technologieobjekte, die beide die Schnittstelle `XApplicationCall(...)` der Transportschicht implementieren. Abhängig vom zu erreichenden Objekt wählt die Referenzschicht das beste Technologieobjekt aus.

Zum Zeitpunkt des Bekanntwerdens eines Objekts auf einem entfernten Knoten werden die `UTID`-Identifikatoren für jede nutzbare Technologie mittels Serialisierung übergeben. Besitzen zwei Knoten eine gemeinsame Technologie, wird diese zur Kommunikation benutzt, ansonsten werden Brückenobjekte an Technologiegrenzen eingesetzt.

Zusätzlich zur Ethernet-Technologie haben wir eine optimierte Technologie für die ParaStation-Hardware implementiert (siehe Abschnitt 5). Die Optimierungen nutzen aus, daß ParaStation-Pakete immer korrekt und immer in der rich-

[3] Der Aus- und Wiedereintritt nach Java über JNI ist im derzeitigen JDK aufwendig.

tigen Reihenfolge zustellt werden. Knoten in unserem Rechnerbündel legen zwei Technologieobjekte an, eines für Ethernet und eines für ParaStation. Das zweite wird benutzt, wenn Kommunikation innerhalb des Rechnerbündels stattfindet, das erstere, um Knoten außerhalb anzusprechen, die nicht mit ParaStation-Hardware ausgestattet sind.

4.4 Einsteckbare verteilte Speicherbereiniger

Verteilte Speicherbereinigung ist schwierig, weil Pakete in verteilten Systemen evtl. vervielfacht, verspätet oder gar nicht ankommen oder ganze Knoten ausfallen. Die Speicherbereinigung in RMI ist für Weitverkehrsnetzwerke mit all diesen Problemen ausgelegt. In einem engverknüpften Rechnerbündel kann auf Zusatznachrichten zur Ausfallsicherheit zugunsten von besserer Geschwindigkeit verzichtet werden.

Da es für ausfallsichere Netzwerke effizientere Speicherbereinigungsalgorithmen gibt [14], bietet KaRMI die Möglichkeit, diese über eine saubere Schnittstelle ins System einzustecken. Es kann für jede Technologie ein anderer Speicherbereiniger verwendet werden. Die oben erwähnten an Technologiegrenzen eingesetzten Brückenobjekte bewirken eine korrekte Zusammenarbeit unterschiedlicher Speicherbereiniger.

4.5 Einschränkungen

Nach einer Änderung am `BOOTCLASSPATH` und der Neuerzeugung von Stellvertreter- und Skelett-Klassen kann ein existierendes Programm von KaRMIs Geschwindigkeitsverbesserungen ohne Neuübersetzung profitieren.

Dennoch muß man bei der Verwendung von KaRMI einige Einschränkungen hinnehmen. Die gravierendste ist wohl, daß KaRMI nicht mit Code zurechtkommt, der Port-Nummern oder die SocketFactory verwendet. Logischerweise kann KaRMI auch nicht mit Code benutzt werden, der undokumentierte RMI-Klassen verwendet (wie z.B. beim San Francisco Projekt der IBM). Weitere Einschränkungen der gegenwärtigen Implementierung sind in [7] diskutiert.

5 ParaStation-Netzwerk

ParaStation [19] ist eine Kommunikationstechnologie, um handelsübliche Arbeitsstationen zu einem Supercomputer zu kombinieren. ParaStation basiert auf Myrinet-Hardware [1] und skaliert bis zu 4096 Knoten. Die *user-level* Kommunikation von ParaStation rettet die kurzen Latenzzeiten der Hardware in die Anwendung, indem das Betriebsystem aus dem Kommunikationspfad herausgenommen wird. Trotzdem bleibt der volle Schutz einer Mehrbenutzerumgebung erhalten.

6 Benchmark-Sammlung und Ergebnisse

6.1 Benchmark-Sammlung

Für die quantitative Beurteilung von KaRMI haben wir eine Sammlung von RMI-Benchmarks zusammengestellt: Verschiedene Kern-Benchmarks und einige kleine Anwendungen. Die Anwendungen benutzen dabei entfernte Methodenaufrufe häufiger, als für die Lösung des Problems notwendig wäre und sind daher oft langsamer als sequentielle Lösungen. Andererseits testen sie häufig vorkommende Kommunikationsmuster oder messen die Geschwindigkeit von RMI im Zusammenspiel mit dem Prozessorzuteilungsverfahren oder der Synchronisierung. Wir behaupten keineswegs, daß die Benchmarks eine repräsentative Sammlung von RMI-Anwendungen sind, dennoch dienen sie als guter Startpunkt für die Bewertung der Geschwindigkeit der grundlegenden RMI-Funktionalität. Die Sammlung ist unter [7] öffentlich zugänglich.

Die Programme werden für jeden Parametersatz mehrfach ausgeführt, um die Auswirkungen von endlicher Zeitauflösung, von Laufzeitschwankungen durch Cache-Effekte oder JIT-Warmlaufphasen und von sonstigen Ausreißern durch Betriebsystemunterbrechungen zu minimieren.

Für jedes Programm der Sammlung wird der Parameter `obj` durch eine simple Fabrik-Klasse erzeugt. Momentan stehen Fabrik-Klassen für die folgenden Typen zur Verfügung:

`null`	null-Zeiger
`byte[`n`]` `int[`n`]`	Feld mit n Elementen
`float[`n`]`	(unsere Sammlung benutzt n=50,200,500,2000,5000,20000)
4 `int` 32 `int`	ein Objekt mit 4 oder 32 `int` Werten
`tree(`n`)`	ein balancierter Baum aus Objekten mit 4 `int` Werten und insgesammt n Knoten (unsere Sammlung benutzt n=15)

a) Kern-Benchmarks mit zwei beteiligten Rechnern
- `void ping()`
- `void ping(int, int)`
- `void ping(int, int, float, float)`
- `void ping(obj)` and `obj ping(obj)`
- `void pingpong(obj)` and `obj pingpong(obj)`

Im Gegensatz zum einfachen `ping` findet bei `pingpong` auf der entfernten Seite ein Rückaufruf statt, bevor beide Aufrufe zurückkehren.

b) Kern-Benchmarks für Serverlast bei konkurrierenden Aufrufen
- `obj star(obj)`

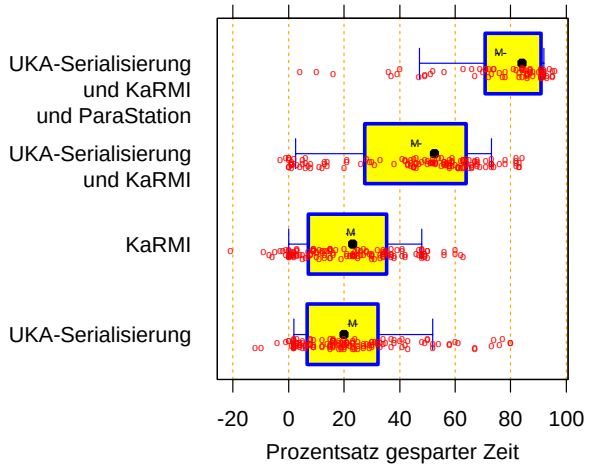
Alle Clienten warten an einer Barriere, bevor sie alle gleichzeitig eine bestimmte Methode desselben Servers aufrufen.

c) Kleine Anwendungen
- Hamming's Problem [3]. Gegeben ist eine Reihe Primzahlen a, b, c, ... in aufsteigender Reihenfolge ohne Duplikate (in unserer Implementierung jede zweite

Primzahl). Gesucht sind, ebenfalls in aufsteigender Reihenfolge und ohne Duplikate, alle Zahlen der Form $a^i \cdot b^j \cdot c^k \ldots \leq n$.
• Erzeugung von Paraffin-Strukturformeln [3]. Gegeben eine Zahl n, liefere zu jedem $i \leq n$ duplikatfrei alle chemischen Strukturformeln für Paraffinmoleküle (C_iH_{2i+2}) und deren Isomere.
• SOR successive overrelaxation, ein iterativer Algorithmus zur Lösung von Laplace Differentialgleichungen auf einem 2D-Gitter. In jeder Iteration wird der Wert eines Gitterpunktes augrund der Werte seiner vier Nachbarpunkte aktualisiert. Die RMI-Implementierung stammt von Maassen [10].

6.2 Ergebnisse

Wir haben vier Software- und drei Hardware-Konfigurationen untersucht. *Software:* reines RMI, RMI mit UKA-Serialisierung, KaRMI mit JDK-Serialisierung und KaRMI mit UKA-Serialisierung. Bei jedem Durchlauf wurden 64 Programme vermessen (jeder Kern-Benchmark mit jedem Objekttyp und die kleinen Anwendungen). *Hardware:* Die in Tabelle 1 erwähnten Plattformen, wobei die Alphas sowohl mit Ethernet als auch mit ParaStation-Netzwerk vermessen wurden.



Die unteren drei „Zeilen“ zeigen je $2 \cdot 64$ Meßergebnisse (Ethernet auf PC und Fast-Ethernet auf Alpha). Die unterste Zeile zeigt die erzielte Verbesserung von RMI mit UKA-Serialisierung. Die zweite Zeile zeigt die Verbesserung, die KaRMI zusammen mit der JDK-Serialisierung bewirkt. Die darüberliegende Zeile zeigt die kombinierte Wirkung. Die oberste Zeile demonstriert das Verhalten, wenn zusätzlich zur UKA-Serialisierung und KaRMI noch das ParaStation-Netz verwendet wird (64 Meßergebnisse).

Abbildung3. Prozentsatz eingesparter Zeit durch UKA-Serialisierung, KaRMI und ParaStation-Netzwerk.

Jede gemessene Zeit ist durch einen kleinen Kreis repräsentiert, welcher den Prozentsatz eingesparter Zeit gegenüber Standard-RMI angibt. M steht für den Mittelwert und der dicke Punkt für den Median der Messungen. Die Rechtecke umfassen die mittlere Hälfte aller Messungen. So spart man mit ParaStation in der Hälfte der Messungen zwischen 70 und 90% der Laufzeit ein; bei einem Viertel sogar über 90%. Die H-Linien geben das 0,1 und 0,9 Quantil der Messungen

an, so daß die kleinsten zehn Prozent der Meßwerte links des H liegen. Ein keiner Auzug der Meßdaten ist in Tabelle 3 abgedruckt.

Tabelle3. Auszug aus den Meßdaten. Sowohl einzeln als auch in Kombination sparen UKA-Serialisierung und KaRMI Laufzeit ein. Durch das ParaStation-Netzwerk ergeben sich weitere Verbesserungen.

	PC gemäß Tabelle 1			DEC gemäß Tabelle 1		
Benchmark (μs)	RMI	KaRMI	UKA + KaRMI	RMI	UKA + KaRMI	UKA + KaRMI + ParaStation
void ping()	745	385 (48%)	360 (52%)	1451	511 (65%)	117 (92%)
void ping(2 int)	731	619 (15%)	398 (46%)	1473	793 (46%)	194 (87%)
obj ping(obj)						
• 32 int	2287	1935 (15%)	674 (71%)	7633	1232 (84%)	328 (96%)
• 4 int, 2 null	1456	1104 (24%)	464 (68%)	4312	1123 (74%)	279 (94%)
• tree(15)	3108	2708 (13%)	1311 (58%)	14713	2485 (83%)	1338 (91%)
• float[50]	1462	1095 (25%)	859 (41%)	2649	1264 (52%)	483 (82%)
• float[5000]	37113	37123 (0%)	37203 (0%)	16954	12590 (26%)	8664 (49%)
paraffins (2PE)	19013	18350 (3%)	7121 (53%)	56870	15580 (73%)	19600 (66%)
paraffins (8PE)				42290	9450 (78%)	13860 (67%)

Abbildung 3 zeigt, daß die UKA-Serialisierung und KaRMI in fast allen Fällen die Geschwindigkeit erhöhen, sowohl wenn sie einzeln benutzt werden, als auch in Kombination. So kann ohne spezielle Kommunikationshardware ein Median von 45% der Laufzeit eingespart werden; in manchen Fällen sind Verbesserungen bis zu 71% möglich. Mit ParaStartion wird im Median 85% und maximal 96% der Zeit eingespart. Nur mit ParaStation ist es möglich einen entfernten Methodenaufruf innerhalb von 117μs auszuführen. (Mit einem neueren JIT von DEC, der allerdings noch zu fehlerhaft für eine durchgängige Verwendung ist, gelangen uns sogar 80μs.)

Jede Zeile in Abbildung 3 enthält Kreise nahe null. Diese stehen für Kern-Benchmarks mit Feld-Parametern. Bei der Übertragung großer Datenmengen ist die Kommunikationszeit der dominierende Faktor, der jegliche Geschwindigkeitssteigerungen im RMI und in der Serialisierung verdeckt (vergleich Tabelle 3 und Abbildung 4). Auf den PCs (links in Abbildung 4) geht die Verbesserung für große Felder gegen null; doch wegen der höheren Bandbreite sieht man auf ParaStation immer noch Verbesserungen von ca. 40%.

Interessanterweise spart die Implementierung für Fast-Ethernet bei der Paraffin-Anwendung mehr Zeit, als die ParaStation-Implementierung. Das hängt mit Prozessorzuteilungsproblemen in der ParaStation-Bibliothek zusammen, wo durch aktives Warten an der Kommunikationshardware andere Kontrollfäden am Weiterarbeiten gehindert werden.

Bei wenigen Messungen beobachtet man eine Verlangsamung bei alleiniger Nutzung von entweder der UKA-Serialisierung oder KaRMI. Für die UKA-

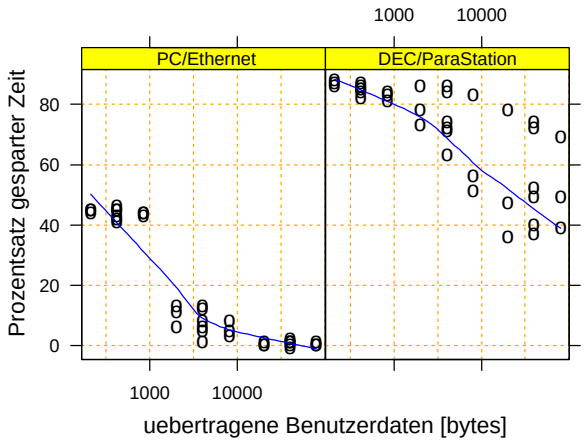


Abbildung4. Der Geschwindigkeitsgewinn durch Kombination von UKA-Serialisierung und KaRMI sinkt mit wachsenden Feldgrößen.

Serialisierung liegt das daran, daß die JDK-Implementierung eine schnellere native Methode zur Erzeugung von uninitialisierten Feldern benutzt, wohingegen in der UKA-Serialisierung der Standard-Konstruktor aufgerufen wird.[4] Die Verlangsamung bei KaRMI beruht darauf, daß die Stellvertreter bei JDK 1.1.6 primitive Parametertypen etwas effizienter handhaben können. KaRMI wird aber deutlich gegenüber einer 1.2-Implementierung gewinnen, sobald eine solche für Digital Unix verfügbar sein wird.

Danksagungen

Wir möchten uns bei Lutz Prechelt bedanken, der uns bei der statistischen Auswertung der Meßergebnisse zur Seite stand. Matthias Gimbel war duldsamer Beta-Tester immer neuer Versionen. Das Java Grande Forum und Siamak Hassanzadeh von Sun Microsystems unterstützten die Diskussion von Unzulänglichkeiten der JDK-Serialisierung und des RMI finanziell.

Literatur

1. N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, 1995.
2. F. Breg, S. Diwan, J. Villacis, J. Balasubramanian, E. Akman, and D. Gannon. Java RMI performance and object model interoperability: Experiments with Java/HPC++. *Concurrency: Practice and Experience*, 10(11–13):941–956, 1998.

[4] Wir werden dieses Problem durch Aufruf derselben nativen Methode beheben.

3. J.T. Feo, editor. *A Comparative Study of Parallel Programming Languages: The Salishan Problems*. Elsevier Science Publishers, Holland, 1992.
4. B. Haumacher and M.Philippsen. More efficient object serialization. In *Parallel and Distributed Processing*, number 1586 in Lecture Notes in Computer Science, pages 718–732, Puerto Rico, April 12 1999. Springer Verlag.
5. S. Hirano, Y. Yasu, and H. Igarashi. Performance evaluation of popular distributed object technologies for Java. *Concurrency: Practice and Experience*, 10(11–13):927–940, 1998.
6. Java Grande Forum. http://www.javagrande.org.
7. JavaParty. http://wwwipd.ira.uka.de/JavaParty/.
8. P. Keleher, A.L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. 1994 Winter USENIX Conf.*, pages 115–131, January 1994.
9. V. Krishnaswamy, D. Walther, S. Bhola, E. Bommaiah, G. Riley, B. Topol, and M. Ahamad. Efficient implementations of Java Remote Method Invocation (RMI). In *Proc. of the 4th USENIX Conf. on Object-Oriented Technologies and Systems (COOTS'98)*, 1998.
10. J. Maassen and R.v. Nieuwpoort. Fast parallel Java. Master's thesis, Dept. of Computer Science, Vrije Universiteit, Amsterdam, August 1998.
11. ObjectSpace. Voyager. http://www.objectspace.com.
12. OMG. *Objects by Value Specification*, January 1998.
13. M. Philippsen, M. Jacob, and M. Karrenbach. Fallstudie: Parallele Realisierung geophysikalischer Basisalgorithmen in Java. *Informatik—Forschung und Entwicklung*, 13(2):72–78, 1998.
14. D. Plainfossé and M. Shapiro. A survey of distributed garbage collection techniques. In *Intl. Workshop on Memory Management*, Kinross, Scotland, UK, September 1995.
15. G.K. Thiruvathukal, F. Breg, R. Boisvert, J. Darcy, G.C. Fox, D. Gannon, S. Hassanzadeh, J. Moreira, M. Philippsen, R. Pozo, and M. Snir (editors). Java Grande Forum Report: Making Java work for high-end computing. In *Supercomputing'98*, Orlando, Florida, November 7–13, 1998. panel handout.
16. G.K. Thiruvathukal, L.S. Thomas, and A.T. Korczynski. Reflective remote method invocation. *Concurrency: Practice and Experience*, 10(11–13):911–926, 1998.
17. R. Veldema, R.v. Nieuwport, J. Maassen, H.E. Bal, and A. Plaat. Efficient remote method invocation. Techn. Rep. IR-450, Vrije Universiteit, Amsterdam, 1998.
18. J. Waldo. Remote procedure calls and Java Remote Method Invocation. *IEEE Concurrency*, 6(3):5–7, 1998.
19. T.M. Warschko, J.M. Blum, and W.F. Tichy. ParaStation: Efficient parallel computing by clustering workstations: Design and evaluation. *Journal of Systems Architecture*, 44(3-4):241–260, 1997.
20. Weimin Yu and A. Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice and Experience*, 9(11):1213–1224, November 1997.