Interprozesskommunikation in verteilten Systemen

Interprocess Communication In Distributed Systems

Andres Koch
Object Engineering GmbH
Birmensdorferstrasse 32
CH-8142 Uitikon Waldegg
Tel +41 (0) 44 400 47 00
Fax +41 (0) 44 400 47 07
email: info@objeng.ch

Erschienen in "Offene Systeme" (1993) Band 2/ Nr. 4, November 93, S. 201-209
Springer Verlag International Heidelberg.

UNIX war eines der Betriebssysteme, welches man als Pionier und als Vorreiter der Prozessorientiertheit und der dazu notwendigen Interprozesskommunikation bezeichnen kann. Die 'Pipe' (xad) ist jedem UNIX-Benutzer auf Shell-Ebene ein Begriff, welche sich dann bis auf die tiefe System-Call-Schicht von UNIX durchzieht (pipe()). Dieser einfache und dadurch mächtige Mechanismus war am Durchsetzungs-Erfolg von UNIX nicht ganz unschuldig. Doch um den Anforderungen von offenen und vorallem verteilten Systemen gerecht zu werden, konnte die Entwicklung auch in UNIX nicht bei der Pipe halt machen. Vorallem mit UNIX V wurden neue und erweiterte Mechanismen wie die 'Message Queue', 'Shared Memory', 'Named Pipe' (FIFO) angeboten. Weitere Mechanismen wie Remote Procedure Call (RPC), Sockets, Message Passing sowie höhere Abstraktionen werden von unterschiedlichen Herstellern angeboten. Dass für offene Systeme im Bereich Interprozesskommunikation ein Standard notwendig ist, scheint offensichtlich und dieser müsste bestimmt auf OSF/DCE aufgebaut sein, um in Zukunft bestehen zu können. Einen Überblick und einen Einblick in dieses Gebiet bietet der nachfolgende Artikel

1 Einleitung

Die allgemeine Tendenz der heutigen Informations-Technologie in Richtung offene und verteilte Systeme scheint uns eine richtige Aufbruchstimmung zu bescheren. Client/Server als neue Applikations-Architektur ist in aller Munde und darf auch als einen guten, relativ einfachen Einstieg in verteilte Systeme gewertet werden.

Die Offenheit von Applikationen für verteilte Systeme hängt aber stark von der Gesamtarchitektur und darin von der Kommunikation ab, worüber die Applikationen zusammenarbeiten können. Die Gesamtarchitektur wird stark durch das 'Distributed Computing Environment' (DCE) von OSF (Open Software Foundation) ([9]) geprägt und es ist absehbar, dass sie sich auch wirklich als Standard in der Praxis durchsetzt.

Auf Basis 'Transport Service Provider' (vgl Abb.1) auf der Kommunikationseite liegen ebenfalls Standards vor, die verbreitet und akzeptiert sind (TCP/IP, SNA (IBM), NetBios (IBM), IPX/SPX (Novell)) (vgl.[6]).

Was aber läuft auf höherer Ebene (Data Exchange Facility), wo Programme (z.B. Client-Programm und Server-Programm) miteinander Daten austauschen müssen?.

Solange verschiedene Programme ihre Informationen über Dateien oder über eine gemeinsame Datenbank

austauschen, wird ganz normal über die entsprechenden APIs (Application Programming Interface) gearbeitet. Ganz abgesehen von der eingeschränkten Durchsatzrate und der Komplexität der Synchronisation, hat dieses Verfahren spätestens seine Grenzen, wenn sich die Programme auf verschiedenen Rechner befinden.

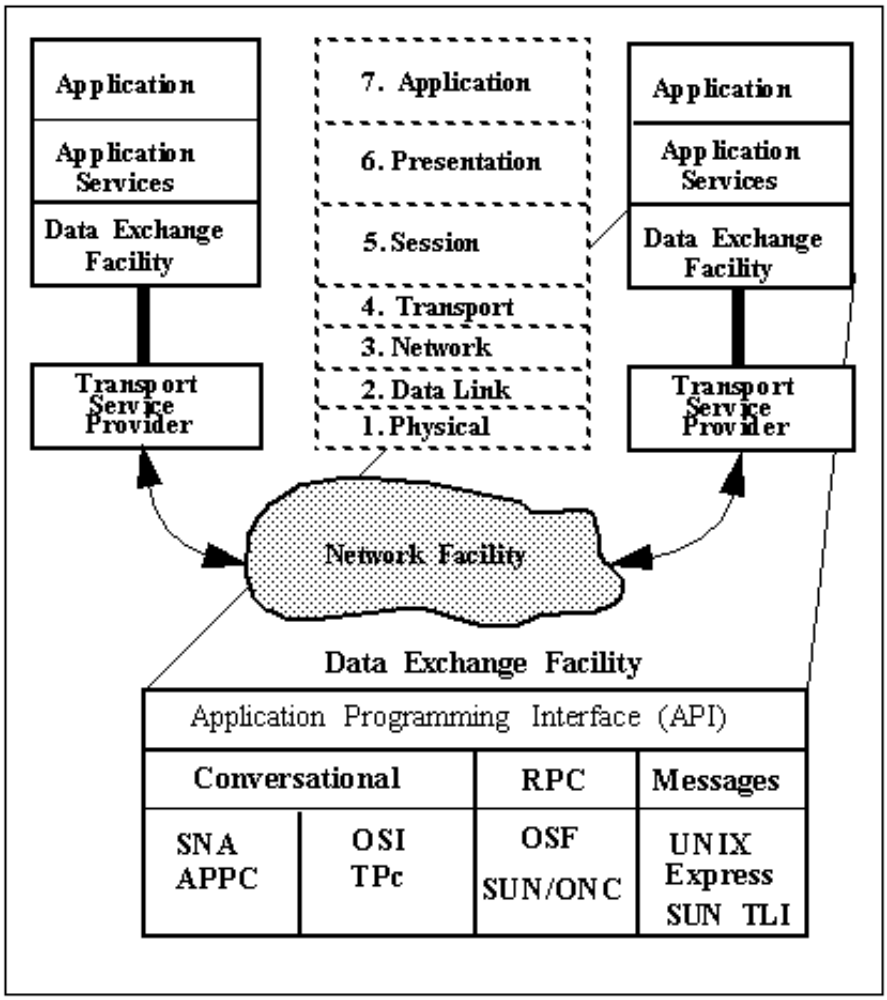
Werden zwischen gleichzeitig laufenden Programmen (Prozesse, Tasks) Informationen ausgetauscht, so wird die Kommunikation möglichst direkt zwischen den Prozessen etabliert. Hier spricht man dann von Interprozess-Kommunikation oder kurz IPC (Interprocess Communication).

Befinden sich die kommunizierenden Prozesse auf dem gleichen Rechner können die Informationen via Speichertransfer mit Hilfe des Betriebssystems ausgetauscht werden. Sofern die Rechnergrenzen aber überschritten werden, muss auf die Kommunikations-Protokolle und -Kanäle zurückgegriffen werden, welche die Rechner verbindet.

Diese Mechanismen sollten natürlich so abstrakt sein, dass der Anwender (Programmierer) keinen Unterschied sieht, ob sich die Prozesse nun auf dem gleichen oder verschiedenen Rechner befinden.

Innerhalb einer Client/Server-Architektur ist das Kommunikations-Protokoll ziemlich starr. Der Client sendet eine Anfrage (Request) und der Server antwortet mit einer oder mehreren Antworten (Reply). Dieses Request-Reply-Schema hat auch auf die Interprozesskommunikation einen Einfluss. Es gibt einzelne IPC-Mechanismen, die für diesen Ablauf sehr geeignet sind (z.B. RPC). Dabei muss man sich aber bewusst sein, dass die Client/Server-Architektur automatisch eine hierarchische Struktur der laufenden Prozesse ergibt, nämlich Client zu Server und evtl. Server zu Superserver etc. Werden zukünftig die Peer-to-Peer-Architektur für Applikationen verwendet, dann sind nicht mehr alle IPC-Mechanismen gleichermassen geeignet. Dies soll im folgenden etwas näher erklärt werden.

Abb. 1: Interprozesskommunikation im OSI-7-Schichtenmodell

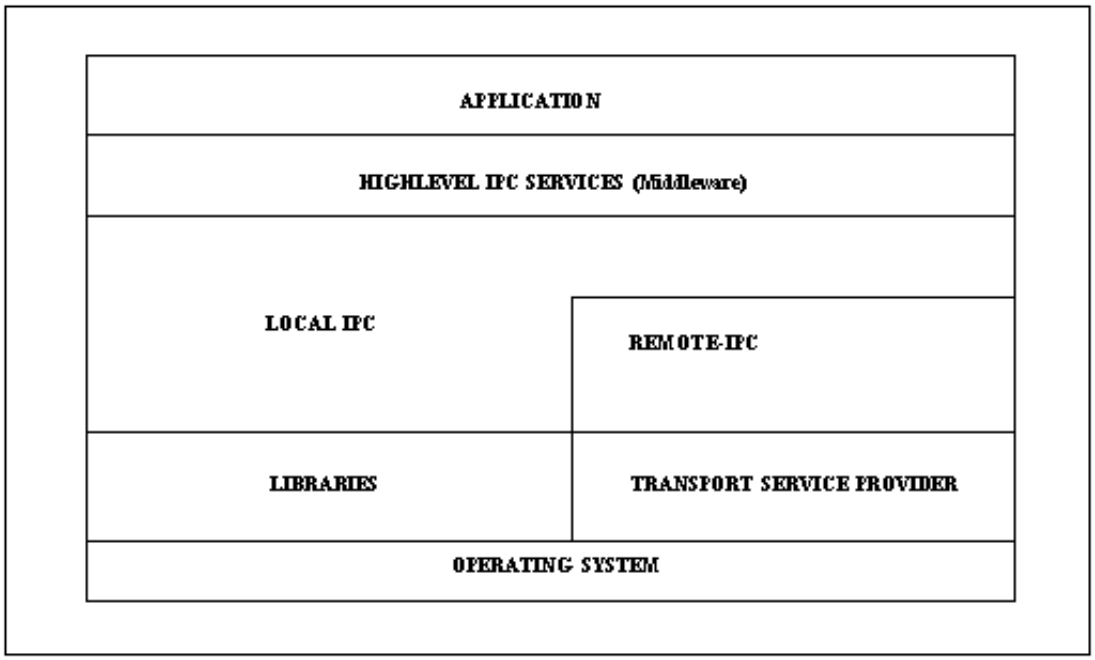


2 IPC-Schichteneinteilung

Wir unterscheiden bei der Interprozesskommunikation folgende Schichten, die zusammengefasst wieder die 'Data Exchange Facility' ausmachen:

- Local-IPC
- Remote-IPC
- Highlevel-IPC-Services

Abb. 2: IPC-Schichtenmodell



Die Local-IPC ist prozessorgebunden und nur innerhalb ines Rechners möglich. Dazu IPC gehören praktisch alle bisherigen IPC-Mechanismen von UNIX, nämlich Pipe, Shared Memory und Message Queues. Die Kommunikation, man könnte auch von einem IPC-Adressbereich sprechen, ist nur innerhalb eines Rechnersystems unterstützt. Entsprechend dieser Einschränkung können natürlich einfachere und schnellere Schnittstellen realisiert werden. Local-IPC genügt in der Regel auch für Prozesse, die auf einem als Server dienenden Rechner laufen. Also zum Beispiel Datenbank-Server-Prozess und seine Threads (leichtgewichtige Prozesse, die auf den gleichen Adressraum wie der Prozess, dem sie zugeordnet sind, zugreifen können).

Bei der Remote-IPC, handelt es sich um Mechanismen, welche sowohl innerhalb des gleichen Prozessors, wie auch zwischen Programmen auf verschiedenen mit Netzwerk (typisch LAN) gekoppelten Prozessoren eingesetzt werden. Hierzu gibt es einerseit die sogenannten Remote Procedure Calls (RPC), die Sockets vom BSD-Unix sowie neuerdings das TLI (Transport Layer Interface) von Sun.

Bei den Highlevel-IPC-Services handelt es sich um sogenannte Middleware, also Software-Pakete, welche als defacto-normierte Schicht zwischen Applikations-Programm und einem darunterliegenden noch nicht ganz normierten System-Plattform befindet. Middleware kann auch die Aufgabe haben, ein gegenüber Applikations-Programmen normiertes, altbewährtes Kommunikations-Protokoll auf eine neuere darunterliegende Architektur anzupassen. Ein typisches Beispiel ist das CICS (Customer Information Control System, (IBM)), welches heute auf verschiedenen Nicht-Host-Plattformen verfügbar ist (AIX, HP-UX, OS/2, ua.). Damit kann auf höherer Ebene ein etabliertes Kommunikations-Protokoll verwendet werden, um das auf dem Host verfügbare CICS zu bedienen. Andere Produkte sind ToolTalk ([5]) von Sun oder der Broadcast Message Server von HP, welche zur Kopplung und Kommunikation zwischen verschiedenen Tools (typisch in CASE und SW-Entwicklungsumgebungen) dienen.

Auf einer ähnlichen Stufe befindet sich die Interprozess-Kommunikations-Bibliothek Express von ParaSoft Corp. ([10]) welches das Austauschen von Meldungen aus C- und Fortran-Programmen unterstützt.

Mit fortschreitender Standardisierung und Entwicklung von DCE-kompatiblen Produkten, werden zum Teil gewisse Middleware-Produkte verschwinden, während sich andere eben der DCE-Umgebung angepassen werden. Altbewährtes ist in der Regel häufig eingesetzt und dadurch trotz neuerer Technologien, nicht so schnell ersetzbar.

Im folgenden wollen wir uns hauptsächlich mit den beiden ersten Schichten befassen, also sowohl mit Local-

IPC wie mit Remote-IPC. Dabei wird uns der Unterschied nur am Rand interessieren, da für verteilte Applikationen, sich der Entwickler wie der Anwender nicht darum kümmern möchte, ob nun prozessorgebunden oder -ungebunden kommuniziert wird. Er soll sich ein Modell vorstellen, in dem mehrere Prozesse auf irgendwelchen Maschinen ablaufen und sich gegenseitig Informationen mittels IPC zustellen können.

3 IPC-Topologien und Protokolle

Es gibt drei Hauptprotokolle, die für IPC massgebend sind:

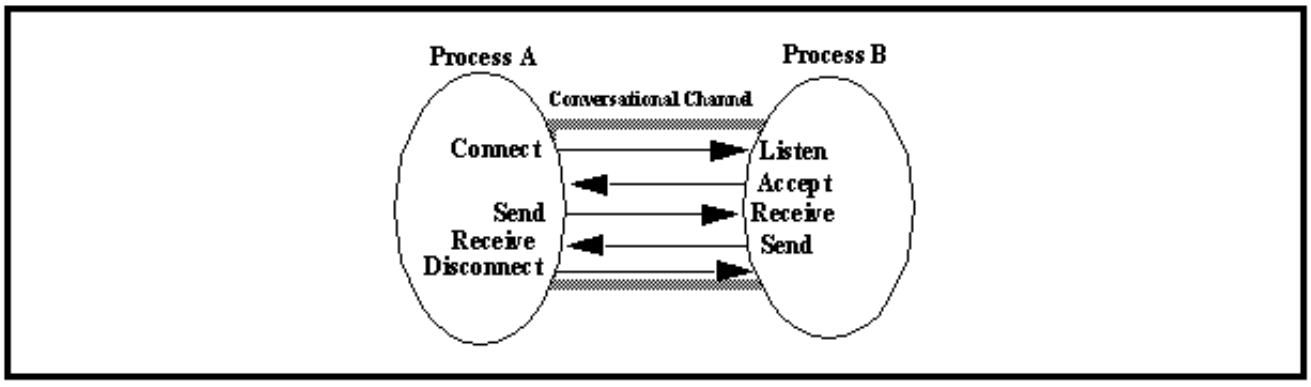
- Conversational Mode
- Remote Procedure Call (RPC)
- Message Passing (Message Queues)

3.1 Conversational Mode

Der Conversational Mode (s.Abb.3) stellt eine virtuelle Verbindung zwischen zwei Prozessen dar. Man kann dies ähnlich einem Telefongespräch vorstellen, bei dem der Anrufer zuerst den Empfänger anwählen muss, dieser muss den Anruf beantworten (Telefon abnehmen) und sich bereit erklären mit dem Anrufer zu kommunizieren. Darauf folgt das "Ping-Pong" von Datenaustausch, abwechslungsweise zwischen den beiden Teilnehmer.

Während dieses Protokoll zwar durch die fixe etablierte Verbindung recht eingeschränkt und synchron ist, hat es den Vorteil für grössere Datenmengen effizienter zu sein, da die Verbindung nicht jedesmal neu etabliert werden muss, wie dies bei den andern beiden, RPC und Message Passing, der Fall ist. In diese Kategorie fallen IBMs APPC (Advanced Programm to Programm Communication), Sockets in BSD sowie die verbindungsorientierte Verwendung von TLI (Sun).

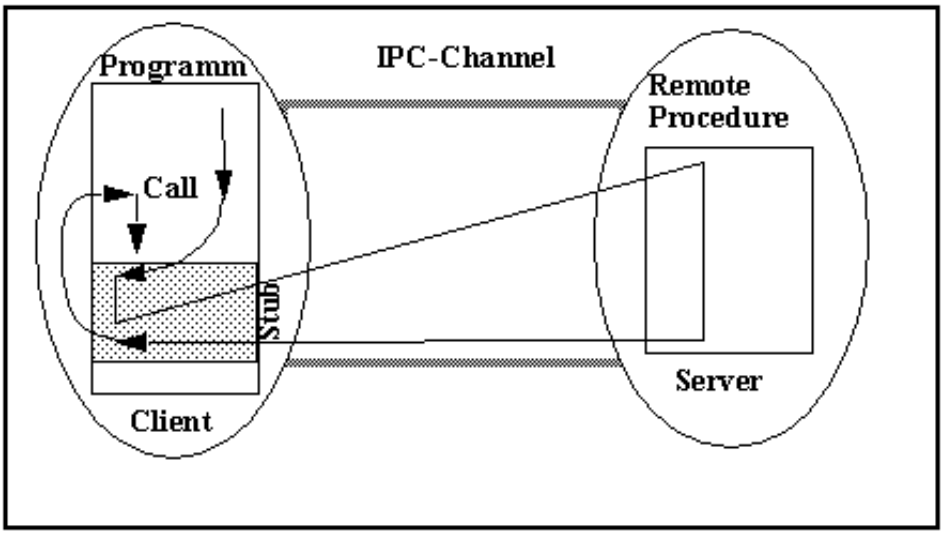
Abb. 3: Conversational IPC



3.2 Remote Procedure Call (RPC)

Das Verfahren der Remote Procedure Calls (RPC) ([4]) ist heute für verteilte Applikationen und vorallem für Client/Server-Anwendungen das häufigst gebrauchte Mittel. Im Programm sieht es einem normalen Funktionsaufruf sehr ähnlich. In der Ausführung hingegen werden die Funktionsparameter in eine Meldung verpackt und an den evtl. entfernten Server geschickt. Dieser packt sie aus und ruft damit die bei ihm lokale Funktion (Procedure) auf. Das Resultat dieser Funktion wird wiederum in eine Meldung verpackt und an den aufrufenden Prozess zurückgeschickt, dort ausgepackt und als Funktionswert (oder veränderte Parameter) zurückgegeben. Der aufrufende Prozess hat also eine entfernte Prozedur aufgerufen woraus der 'Remote Procedure Call' seinen Namen erhielt (Abb.4)

Abb. 4: Remote Procedure Call.



Auf den ersten Blick, scheint dieser Mechanismus sowohl einfach im Konzept wie auch einfach in der Bedienung. In der Praxis zeigen sich aber folgende Grenzen:

- Die Definitionen von RPCs in Programmen müssen mit einer speziellen Zwischensprache (IDL) und dem entsprechenden Präprozessor übersetzt werden. Dieser Vorgang erfordert etwas mehr Spezial- und Systemkenntnisse als üblich.
- Der aufrufende Prozess (Thread) blockiert bis der aufgerufene Prozess die Funktion ausgeführt hat. Ein Weiterarbeiten des Client-Prozesses ist nur möglich, wenn er für den Aufruf einen eigenen Thread generiert, was dann die RPCs nicht mehr als nur Prozeduraufrufe erscheinen lässt.

In der Regel werden solche RPCs in Applikationsbibliotheken verpackt und von Systemspezialisten entworfen und programmiert. Der Anwendungs-Entwickler ruft dann wirklich nur eine Funktion auf, welche ihrerseits dann die RPC-Mechanismen aktivieren.

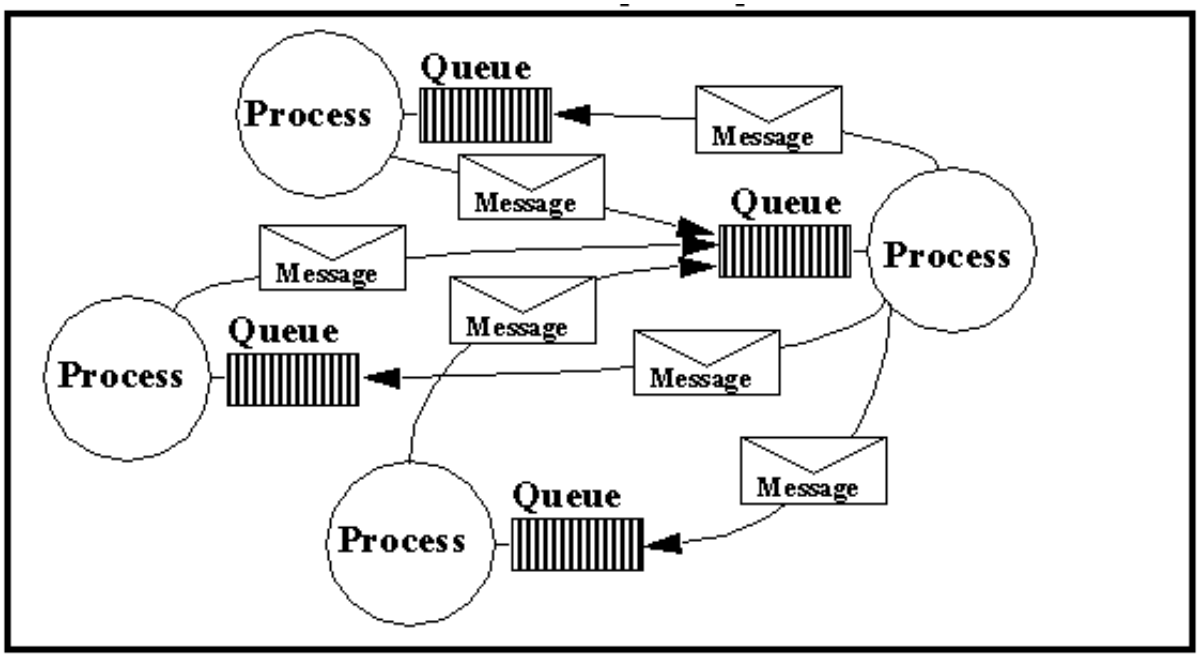
Jede Remote Procedure erhält eine spezielle Nummer (eindeutig), über die sie im System den Standort der Implementation der Prozedur findet. Die Verwaltung dieser Nummern muss natürlich so gemacht werden, dass im Adressbereich des Netzwerkes in dem RPC verwendet werden, diese Nummern eindeutig sind. In OSF/RPC wird dies vom UUID-Generator (Universal Unique Identifier) unterstützt. Ein weiteres Problem, das wir teilweise auch bei Programmbibliotheken kennen, ist die Version. Wenn ein Client-Programm via RPC eine Prozedure eines Servers aktiviert, dessen Programm aber inzwischen geändert wurde, dann muss der Server entscheiden, ob er eine ältere Version eines RPCs noch unterstützen kann oder nicht. Das zeigt auch, dass in einem verteilten System das Administrieren von Programmversionen wesentlich aufwendiger wird, wohl geplant und durchdacht sein muss, als dies bereits bei zentralen Applikationen der Fall war. Eine weitere Aufgabe, die bei allen drei Mechanismen zu lösen ist, bildet die unterschiedliche Datenrepräsentation auf verschiedenen Rechnerarchitekturen. Der Entwickler und RPC-Benutzer, weiss ja nicht auf welchem Rechner die Server-Applikation läuft. Es ist deshalb die Aufgabe der Interface-Sprache und eines Runtime-Services diese Datenkonversion korrekt vorzunehmen. Das Konzept ist so, dass jeder sendende Rechner seine Daten auf ein standardisiertes Netzwerk-Format (z.B. XDR = External Data Representation (SUN), NDR = Network Data Representation (OSF)) formatiert. Der empfangende Rechner wandelt dann die Daten wieder auf seine Datendarstellung zurück. Wieso muss man dies überhaupt machen? Als einfaches Beispiel nehmen wir die Datenrepräsentation einer kurzen Integerzahl auf einem Intel 80486 und auf einem Motorola 68000-xx. Auf dem Intelprozessor wird im Speicher zuerst Bit 0-7 dann Bit 8-15 abgespeichert, während beim 68000 zuerst Bit 8-15 dann 0-7 abgespeichert wird. Wenn man ohne Datenkonversion eine kurze Integerzahl mit dem Wert 1 von einem 68000 auf einen 80486 schickt, repräsentiert diese Integerzahl

dort den Wert 256, ein "kleiner" Unterschied.

3.3 Message Passing (MP)

Das Prinzip des Message Passing (s. ~~Abb. 5~~) kann stark mit unserer Briefpost verglichen werden. Das Programm verpackt seine Daten, setzt Absender und Empfängeradresse drauf und übergibt das "Paket" an einen "Message Passing Service". Dieser sucht sich den "Briefkasten" (Message Queue) des Empfängers aus und legt das "Paket" dort ab. Das Empfängerprogramm holt sich bei günstiger Gelegenheit dieses Paket ab und verarbeitet es. Eventuell sendet es daraufhin eine Antwort an den Empfänger zurück.

Abb. 5: Message Passing



Wir unterscheiden zwischen zwei verschiedenen Methoden von Message Passing:

- Synchrones Message Passing
- Asynchrones Message Passing

Beim synchronen Vorgang, müssen Empfänger- wie Senderprozess sich zu einem bestimmten Zeitpunkt treffen (vgl. Rendezvous in ADA), zu welchem sie die Daten austauschen. Dieser Vorgang hat den Vorteil, dass keine Speicherprobleme durch einen schnellen Sender und einem langsameren Empfänger auftreten. Als Nachteil muss man werten, dass natürlich Deadlocks (Blockaden) auftreten können, weil zwei Prozesse (evtl. über einen Dritten) Daten vom andern erwarten. Ebenfalls kann der Sender oder Empfänger nicht weiterarbeiten, wenn der Partner nicht bereit ist (ähnlich wie bei RPC ohne zusätzliche Threads). Das synchrone Message Passing in Kombination mit Timeouts wird häufig in Realtime- und Prozesssteuerungs-Systeme eingesetzt, weil damit das Verhalten besser im voraus zu beurteilen ist, besser getestet werden kann und auch stabiler ist. Ein synchron arbeitendes System benötigt weniger dynamisch allozierten Speicherplatz als eines mit asynchronem Message-Passing.

Beim asynchronen Vorgang, kann der Senderprozess weiterarbeiten, nachdem er die Meldung in die Message Queue (Briefkasten) des Empfängers gelegt hat. Dies bedingt natürlich, dass darin genügend Speicherplatz alloziert werden kann, um die Meldung aufzunehmen. Der Empfänger kann entweder auf die Meldung warten (Blocked Receive) oder wird beim Eintreffen der Meldung mit einem Event (Signal) darauf aufmerksam gemacht, dass eine Meldung im Briefkasten vorhanden ist. Die asynchrone Form entspricht mehr unserer Denkweise und wie unsere Organisationen operieren. Um nämlich geeignete Algorithmen für verteilte

Applikationen zu finden, kann man sich an unseren Organisations-Techniken orientieren. Die humanen Organisation sind exellente Beispiele für verteilte und echtparallel operierende Systeme.

Für die entsprechende Applikation sollte die geeignete Form gewählt werden. Viele Bibliotheken und Pakete lassen sich für beide Verfahren einfach konfigurieren oder parameterisieren. Ebenfalls kann man mit dem asynchronen Verfahren auch ein synchrones Verfahren simulieren, indem der Sender auf eine Reply-Meldung wartet, welche ihm bestätigt, dass der Empfänger die Meldung entgegen genommen hat. Damit wären wir wieder bei dem Request-Reply-Schema, das auch bei RPC dem Entwickler aufgezwungen ist.

Ebenfalls gibt es Mischformen von Sende- und Empfangs-Operationen. Man unterscheidet zwischen:

Modus / Operation	**Senden**	**Empfangen**
Blockierend	der Sender wartet bis die Meldung abgelegt (z.B. wenn kein Speicher vorhanden ist) ist, und der Empfänger diese abgeholt hat.	der Empfänger wartet bis eine Meldung verfügbar ist.
Unblockierend	der Sender kann unmittelbar arbeiten, nachdem er die Meldung dem System übergeben hat.	der Empfänger initialisiert den Buffer in dem er eine Meldung empfangen will und arbeitet dann weiter. Beim Eintreffen der Meldung wird diese in den reservierten Buffer abgelegt und dem Empfangsprozess wird signalisiert dass eine Meldung eingetroffen ist.

Wie bei allen verteilten Systemen, spielt die Adressierung auch beim Message Passing eine zentrale Rolle. Hier die geeignete Form zu finden bedarf noch einiges an Standardisierung. Trotzdem ist es angebracht auf Basis von symbolischen Adressen zu operieren, welche dann via eine Übersetzung - z.B. via Directory Service - in eine eindeutige Adresse im Netzwerk umgewandelt werden kann.

Obwohl RPC sehr stark verbreitet ist und als dominant wirkt, zeichnet sich ein Trend ab, dass in Zukunft Programme vermehrt auf der Basis von Message Passing arbeiten werden. Während RPC sehr geeignet für die Client/Server-Architektur ist, eignet sich Message Passing für zukünftig stark verteilte Systeme mit Peer-to-Peer-Kommunikation, worin jeder mit jedem kommunizieren kann. Für den Anwendungsprogrammierer zeigt sich die Kommunikation zu andern Prozessen auch transparenter, als dies bei RPC der Fall ist.

Wenden wir uns kurz der objektorientierte Programmierung zu, so hören wir oft die Aussage: "Objekt A schickt Objekt B eine Meldung", wenn es darum geht, dass Objekt A eine Methode (Operation) eines anderen Objektes aufruft. Das lässt einerseits den naheliegenden Schluss zu, dass Meldungen für verteilte Prozesse wie eine Methode für Objekte zu betrachten sind. Diese Annahme erhärtet sich, wenn man dan Entwurf von verteilten Systemen mit einer objektorientierten Design-Methode (wie z.B. Booch [3]) macht und sieht, dass die einzelnen Prozesse nichts anderes als parallel laufende Objekte sind, die verschiedene Operationen (Methoden) zur Verfügung stellen und die Aufträge via Meldungen empfangen (Methode aktivieren). Dieses Vorgehen hat der Autor bereits vor Jahren anhand einer meldungsorientierten Programmiersprache gezeigt ([8]). Vergessen wir dabei auf der RPC-Seite nicht, dass natürlich auch RPC ein direkter Weg ist, Operationen von verteilten Objekten (oder Prozessen) zu aktivieren (vgl. [12]).

Wenn man wieder auf die DCE-Architektur von OSF schaut, worin RPC seinen festen Platz hat, wäre

Message Passing im Bereich "Other Fundamental Services" basierend auf RPC zu positionieren. Damit wäre der unterliegende Datenaustausch durch OSF/RPC standardisiert und darauf das Message-Passing aufgebaut, das für eine einfache, transparente Programmierung in verteilten Systemen verwendet werden kann.

4 IPC-Grundmechanismen in UNIX V

Wie erwähnt bietet UNIX System V einige Local-IPC-Mechanismen standardmässig an, welche heute hauptsächlich für die Kommunikation zwischen Programmen eines abgeschlossenen Applikations-System verwendet werden aber noch wenig für den Austausch zwischen Applikationen verschiedener Hersteller. Nachfolgend sind diese Mechanismen kurz charakterisiert.

4.1 Pipe

Wie einleitend erwähnt, ist der einfache aber mächtige Pipe-Mechanismus so alt wie UNIX und hat UNIX zu seiner Flexibilität verholfen. Es handelt sich dabei um einen Buffer, in welchen der produzierende Prozess hineinschreibt und der konsumierende Prozess die Daten herausliest. Der Pipe-Systemaufruf kann zwischen zwei Prozessen verwendet werden, wenn entweder beide vom gleichen Prozess abstammen (Parent) oder einer vom andern Prozess abgespaltet (fork (2)) wurde. Der Mechanismus welcher anfangs als Basis-Systemaufruf (pipe(2)) geführt wurde, wird heute durch eine zusätzliche Version ergänzt, die sich etwas konfortabler gestaltet. Der C-Aufruf:

```
pfid = popen ("ls -l", "R");
```

eröffnet einen neuen Shell-Prozess, in welchem der Befehl "ls -l" (Verzeichnis anzeigen) ausgeführt wird und die Ausgabe kann darauf mit den gängigen FILE-Streambefehlen (fget, fscanf, fgets, fprintf etc.) wie wenn die Daten von einem File kämen, bearbeitet werden.

4.2 Named Pipe (FIFO)

Die Einschränkung der einfachen Pipe, ist beim FIFO oder der "Named Pipe" aufgehoben. Voneinander unabhängige Prozesse können miteinander bidirektional kommunizieren, indem beide auf die Pipe zugreifen, die im Filesystem zum Beispiel mit dem Befehl '/etc/mknod pipexy p' kreiert wurde. Die Daten werden mit den System-Calls write(2) und read(2) ins FIFO geschrieben resp. herausgelesen. Diese Operationen gelten als unteilbar, das heisst, auch wenn der Prozess unterbrochen wird, wird der übergebene Datenblock ganzheitlich hineingeschrieben. Named Pipes gehören heute in jedes modernere Betriebssystem, sind auch z.B. in OS/2 oder NT vorhanden und erlauben darum eine einfache und teilweise portierbare Interprozess-Kommunikation.

4.3 Message-Queues

Die oben bereits ausführlich beschriebene IPC-Topologie, das Message-Passing, kann unter UNIX mit Message-Queues für lokale IPC implementiert werden.

Ein Prozess kann eine solche Message-Queue - ähnlich einem File - kreieren. Anstelle des Filenamens verwendet man eine 32-Bit-Zahl, der sogenannte Key (vom Typ key_t=long). Andere Prozesse können sich dann ebenfalls einen Zugriff auf die Message-Queue verschaffen, indem sie den gleichen Key bei der Eröffnung (msgget(2)) angeben. Die Queues können wie Files für den Eingentümer, die Gruppe oder für alle andern geschützt werden. Typisch wäre, dass der Eigentümer-Prozess lesen und schreiben darf, während die andern Prozesse nur hineinschreiben dürfen. Das Lesen und Schreiben erfolgt mit den Befehlen msgsnd resp. msgrcv. Durch die Angabe der entsprechenden Parameter kann man wählen zwischen blockierendem und

unblockierendem Senden resp. Empfangen. Eine gewisse Einschränkung bei Message-Queues sind ihre beschränkte Grösse. Obwohl bei der Kernel-Generierung parameterisierbar, liegt die Grösse der Queues typisch bei 2 KB oder 8 KB, in Ausnahmefällen 64 KB. Der Sender muss also damit fertig werden, wenn er auf der Empfängerseite einen langsameren Prozess hat, der seine Message-Queue nicht schnell genug abarbeiten mag.

Eine Meldung hat ein sehr einfaches Format. Eine long-Variable bestimmt den Typ der Meldung, gefolgt von einer Sequenz von Bytes, die Daten. Der Meldungstyp kann beim Empfangen zur Prioritätssteuerung verwendet werden. Man kann dabei angeben, dass man nur einen bestimmten Typ von Meldung empfangen will oder alle Meldungen, die einen kleineren oder gleichen Typ haben wie der angegebene Parameter. Message-Queues sind wesentlich einfacher anzuwenden wie RPCs. Dabei muss man aber beachten, dass was man bei RPCs via der Interface Definitions Sprache (IDL) macht, nämlich die Datentypen und Strukturen der Parameter zu spezifizieren, beim Message-Passing (mit Message-Queues) noch von Hand macht, nämlich die Struktur und die Typen zu spezifizieren, welche in einer Meldung mitgeschickt werden. Verwendet man dabei eine objektorientierte Sprache wie z.B. C++, kann dies durch Ableitung von einer Basis-Meldung aber wiederum sehr einfach gemacht werden.

Eine schöne Eigenheit der Message-Queues (generell bei Message Passing) ist, dass sich der Anwender nicht um Synchronisation zu kümmern braucht. Bei den UNIX-Message-Queues übernimmt das System die Synchronisation, das heisst eine Meldung wird immer als ganzes in die Queue des Empfängers geschrieben und nicht etwa in Teilen. Dadurch können Meldungen auch zur Synchronisation auf höherer Ebene verwendet werden.

In der Praxis bewähren sich die Message-Queues sehr gut. Für komplexere Systeme muss man aber einige zusätzliche Funktionen hinzufügen, damit, verpackt in eine Bibliothek, der Anwender einen zuverlässigen Message-Passing-Dienst zur Verfügung hat.

4.4 Shared Memory

Sollen mehrere Prozesse schnell und häufig auf grosse Datenbestände zugreifen können, die in einem System evtl. laufend ändern, wird IPC mittels Message-Queues oder Named Pipes bald zu langsam. Die Lösung dazu heisst Shared Memory, also gemeinsamer Speicherbereich für zwei oder mehrere Prozesse, welche sonst nicht den gleichen Adressraum haben. Beim Shared Memory, fordert ein Prozess ein Speichersegment, spezifiziert durch Key (wie oben bei Message-Queues beschrieben) und Grösse vom Betriebssystem an. Andere Prozesse können sich nun mit dem gleichen Key auch Zugriff auf diesen Speicherbereich verschaffen. Via eine Referenz können alle Prozesse in diesen Speicherbereich sowohl schreiben wie auch davon lesen, je nachdem wie die Zugriffsberechtigung vom kreierenden Prozess gesetzt wurde. Diese Zugriffe finden in der Regel mit hoher Durchsatzrate statt, da ja von den Prozessen direkt, fast wie auf den eigenen Adressraum zugegriffen wird. Dabei darf man aber einen Aspekt, den der Synchronisation, nicht vergessen. Im Gegensatz zu Message-Queues sind Shared-Memory-Segmente nicht automatisch synchronisiert. Dazu setzt man am Besten entweder Semaphoren (ebenfalls unter UNIX System V verfügbar) oder Meldungen ein. Ansonsten kann die Verwendung von Shared-Memory sogar als einfacher als Message-Queues eingestuft werden.

Wer die Eigenheiten der UNIX System V IPC in der Praxis kennen lernen will, der ist gut beraten, wenn er den Text "Parallele Prozess unter UNIX" ([7]) zur Hand hat, in dem UNIX-IPC u.a. sehr gut, mit Beispielen beschrieben wird.

4.5 Implementationshinweise

Ein Problem, stellt sich sowohl bei der Verwendung von Message-Queues, Shared-Memory wie bei Semaphoren, die ihre Identifikation alle über den erwähnten 32-Bit-Key erhalten.

Sofern man sich an ein bestimmtes Verteilschema oder an einen Verteilalgorithmus hält, wie die Keys unter den Applikationen zugeteilt werden, besteht kein eigentliches Problem. Wenn aber Applikationen von verschiedenen Herstellern diese IPC-Mechanismen verwenden und unterschiedliche Verteilschemen oder -Algorithmen benutzen, dann kann es zu Überschneidungen von Keys kommen. Als Lösung wird empfohlen den Systemaufruf "ftok (path, id)" zu verwenden, welcher aufgrund eines bestehenden Filesystemeintrags (path) und einer von der Applikation vergebenen Identifikation (id) einen einmaligen Key retourniert, der wiederum zur Identifikation der verwendenden IPC-Ressource benutzt werden kann. Sofern sich möglichst alle an diesen System-Call halten, sollte es auch nicht zu Überschneidungen kommen.

Weiter empfiehlt es sich bei der Verwendung von Message-Queues, pro Prozess genau eine Message-Queue zu kreieren (KEY könnte z.B. der Prozess-Identifier sein), über die der Prozess die Meldungen empfängt. Das erlaubt dem Prozess nur von einem Kanal Meldungen zu erwarten, ohne komplexe Abfrage-Algorithmen bewerkstelligen zu müssen. Dies ist natürlich keine Forderung, denn es kann absolut angebracht sein, dass ein Prozess mehrere Message-Queues hat, um zu mehreren externen Prozessen eine gerichteten Kommunikations-Kanal zu haben. Dies muss je nach Anwendung und Durchsatzanfordernissen entschieden werden. Zu beachten ist aber, dass die maximale Anzahl von Message-Queues pro Benutzer und pro System beschränkt sind (kann im Kernel konfiguriert werden).

Sollten einmal, beim Test oder beim unkontrollierten Abbrechen eines Programmes IPC-Ressourcen (Message-Queues, Shared-Memory-Segmente, Semaphoren) "liegen" bleiben, so helfen die beiden Kommandos ipcs zum Anzeigen der IPC-Statusliste und ipcrm zum forcierten Löschen von unerwünschten Ressourcen.

Baut man in einem Projekt auf einem Message-Passing-System-Service auf und setzt dabei auch eine objektorientierte Programmiersprache wie C++ ein, lohnt es sich die IPC-Mechanismen in Klassen zu verpacken, um die Anwendung derer zu vereinfachen und sicherer zu machen ([1]). Der Autor hat solche Klassen, welche UNIX-Prozesse (fork) und Message-Passing mit Message-Queues unterstützen in mehreren Versionen entwickelt und damit sehr gute Erfahrungen gesammelt. Vorallem der Aspekt, ganze Objekte, unabhängig von ihrer Komplexität via Message-Queues an einen andern Prozess zu senden, eröffnet neue, sehr interessante und mit Hilfe von OO sehr mächtige Perspektiven.

5 Zusammenfassung

In diesem Artikel wurde versucht einen groben Überblick über die Interprozess-Kommunikation zu geben. Ausgehend vom gewohnten Kommunikations-Schichtenmodell (OSI), wurde aufgezeigt welche Aufgaben, Topologien und Protokolle sich in der "Data Exchange Facility" verbergen.

Dabei wurde auch versucht, einen Vergleich zwischen RPC, wie in OSF/DCE verwendet und der Message-Passing-Methode zu geben. Die Arbeiten von OSF, vorallem im Teilgebiet DCE wird zukunftsbestimmend sein.

Ebenfalls im Überblick wurden die IPC-Mechanismen von UNIX System V gestreift und einige Tips für die Implementation gegeben.

Zu erwähnen wären noch die Sockets, die im BSD Unix zur Interprozess-Kommunikation auch über Rechnergrenzen hinweg verwendet werden. Die Verwendung von Sockets ist einfach aber von der Datensicherheit eher schwach. Da es nicht Bestandteil von UNIX System V ist, wird deren Verwendung in Neuentwicklungen von SUN nicht mehr empfohlen. Als Alternative dazu sollen TLI oder andere Mechanismen verwendet werden.

Die Verwendung der behandelten IPC in verteilten Systemen, betrifft zukünftig auch bald Objekte, welche verteilt werden müssen. Die objektorientierte Technologie, kann sowohl beim Einsatz von IPC-Klassen-Bibliotheken wie aber überhaupt beim Entwerfen von verteilten Applikationen von grosser Bedeutung sein. Dabei darf man den Object Request Broker von der Object Management Group nicht aus den Augen lassen, denn daraus werden zusammen mit DCE von OSF die Standards der Zukunft wachsen ([9],[2]).

6 Literatur

J.Th.Berry, "The Wait Group's C++ Programming", Howard W. Sams&Company, Indianapolis, USA.

T. Beyer, "Objektbörse, CORBA-OMG Standard für verteilte Objekte", iX 2/93, S. 24-33.

G.Booch, "Object Oriented Design with Applications", The Benjamin/Cummings Publishing Company, Inc. New York, ISBN 0-8053-0092-0, 1991.

M.Busch, "Erste Schritte, Hilfestellung bei der RPC-Programmierung durch System-Tools", iX 6/1993, S.184-186.

M.Busch, "Einführung in Suns ToolTalk-Service", iX 5/1993, S.122-127.

R.J.Cypser, "Communications for Cooperating Systes, OSI, SNA and TCP/IP", Addison-Wesley Publishing Company Inc., ISBN 0-201-50775-7, 1991.

Danneger/Geugelin, "Parallele Prozesse unter UNIX", Carl Hanser Verlag München/Wien, ISBN 3-44-15911-8, 1991.

A. Koch/TSE. Maibaum, "Message oriented Language for System Applications", Proceeding The 3rd International Conference on Distributed Computing Systems, Miami/Florida, October 1982.

OSF,"DCE Application Development Guide", Open Software Foundation, Cambridge, USA, 1992.

ParaSoft, "A Tutorial Introduction to Express", Version 3.2, ParaSoft Corp., 2500, E.Foothill Blvd., Pasasdena, CA 91107.

M.J. Rochkind, "Advanced UNIX Programming", Prentice-Hall Inc., Englewood Cliffs, New Jersey, ISBN 0-73-011818-4, 1985.

A.Schill, M.Person, "Verteilte objektorientierte Erweiterung des OSF Distributed Environments", Offene Systeme Nr.2 /1993, S.94-100, Springer International, 1993.