Universität Ulm
Fakultät für Informatik

Verteilte Unix-Betriebssysteme

Robert Regn
Universität Ulm

Nr. 94-16
Ulmer Informatik-Berichte
Dezember 1994

Verteilte UNIX*-Betriebssysteme

Robert Regn
Fakultät für Informatik
Universität Ulm
D-89069 Ulm, Germany
email: regn@informatik.uni-ulm.de

Zusammenfassung

Während es bereits kommerzielle Betriebssysteme für speichergekoppelte Rechner gibt, sind Betriebssysteme für verteilte Rechnerkonfigurationen weiterhin im Forschungsstadium. Dieser Artikel erläutert zunächst die Ziele eines verteilten Betriebssystems und führt dann zu den Entwurfskonzepten, unterteilt nach Kommunikation, Dateiservice und Prozeßmanagement. Dabei werden auch Ansätze zur Verteiltheit, wie das NFS von Sun, diskutiert. In der zweiten Hälfte des Artikels werden die beiden UNIX-kompatiblen Betriebssysteme Locus und Chorus eingehend beschrieben.

*UNIX ist ein eingetragenes Warenzeichen der AT&T

1 Einführung

1.1 Unix für speichergekoppelte Rechner

Das einfachste parallele Modell, für das man ein Unix Betriebssystem modifizieren kann, ist ein Rechner mit mehreren Prozessoren, die sich einen gemeinsamen Speicher und alle peripheren Geräte teilen (sogenannte enge Kopplung). Durch die hinzugefügten Rechenwerke soll der Durchsatz des Systems gesteigert werden. Alle Prozessoren arbeiten unabhängig voneinander und führen die gleichen Maschinenbefehle des Betriebssystemkerns aus. Diese Konfiguration ist zwar im strengen Sinn nicht verteilt, da alle Prozessoren auf engem Raum zusammenarbeiten. Die Nutzung des gemeinsamen Arbeitsspeichers vermeidet aber viele Probleme. So kann der Austausch von Informationen zwischen den Prozessoren fast ohne Zeitverlust über den Arbeitsspeicher erfolgen und die Prozesse sind leichter zu synchronisieren. Das Verlagern von Prozessen von Knoten zu Knoten ist sehr einfach, da das Prozeßimage im Arbeitsspeicher für alle Knoten in gleicher Weise und ohne Verzögerung verfügbar ist – nur die Übergabe des Prozesses muß abgestimmt werden. Der Benutzermodus bereitet keine Schwierigkeiten. Da im Benutzermodus alle Prozesse autonom arbeiten, bereitet es keine Probleme, mehrere Prozesse in diesem Modus parallel auf mehreren Prozessoren laufen zu lassen. Durch den Scheduler, der immer einen freien Prozessor aussucht, erreicht man ganz ohne weitere Maßnahmen einen automatischen Lastausgleich im System.

Eine Spezialität von Unix macht dieses System für eine Parallelisierung besonders geeignet: Die **Pipe**. Die Pipe wurde ursprünglich entworfen, weil der kleine Arbeitsspeicher der PDP 11 (8kB für Benutzerprogramme), auf der UNIX anfang der 70er Jahre lief, es nicht erlaubte, sehr große Programme auszuführen. Mit der Pipe konnte man ein großes Programm sehr leicht als Kombination von einzelnen Teilprogrammen ausführen, die immer wieder ein- und ausgelagert wurden.

Die Größe des Arbeitsspeichers ist zwar längst kein Engpaß mehr, doch die Leistungsfähigkeit der Prozessoren kommt bald an physikalische Grenzen. Die Pipe kann nun dazu dienen, mehrere Teilprogramme auf verschiedenen Prozessoren parallel auszuführen. Parallelisiert man UNIX, so bekommt man als Folge die Unterstützung des **Makropipelinings** quasi geschenkt. Da viele Tools zur Verwendung in Pipes bestimmt sind (sog. Filter), kann jeder Benutzer vom Makropipelining profitieren, ohne dazu Programme umschreiben oder eigene Programme entwerfen zu müssen.

Problematisch ist allerdings die Synchronisation des Betriebssystems, wenn mehrere Knoten im kernel-mode arbeiten, denn nach dem Betreten des Kerns arbeiten die Prozesse auf den globalen Datenstrukturen des Kerns. Zur näheren Erläuterung sei zunächst die Situation auf dem Monoprozessor beschrieben. Auf einem Monoprozessor stellt der Unix Kern einen Monitor dar.

Wenn ein Prozeß erkennt, daß seine Priorität nicht mehr die höchste ist, wird ein Zustandsübergang von *Systemmodus laufend* nach *bereit* durchgeführt. Dieser Übergang ist jedoch nur an einer ganz bestimmten Stelle, nämlich am Ende von Systemaufrufen möglich, also kurz bevor der Prozeß in den Benutzermodus wechseln wird.

Prozesse, die im Systemmodus laufen, können nicht von anderen Prozessen verdrängt werden. Der Kern wird daher **verdrängungsfrei** (non-preemptive) genannt, obwohl Prozesse im Benutzermodus verdrängt werden können. Durch die Verdrängungsfreiheit

sichert der Kern die Konsistenz der globalen Daten und löst das Problem des gegenseitigen Ausschlusses. Auch eine Unterbrechung kann nicht dazu führen, daß ein Prozeß im Systemmodus den Prozessor verliert. Erst wenn ein Prozeß sich blockiert, erlaubt er einen Prozeßwechsel. Die globalen Daten befinden sich dann in einem konsistenten Zustand.

Einem Prozeß im Systemmodus kann die CPU nur entzogen werden, wenn er selbst dies veranlaßt; im allgemeinen deswegen, weil er auf Beendigung eines Ein- oder Ausgabevorgangs warten muß, also sich über den Aufruf von *sleep* blockiert. An solchen natürlichen Wartestellen sind die Datenstrukturen des Kerns konsistent. Kritische Abschnitte sind also vor (quasi-) parallelem Zugriff geschützt. Die Monitoreigenschaft macht eine Synchronisation von mehreren laufenden Prozessen im Systemmodus unnötig.

Dieser Monitor zerbricht beim Einsatz von parallelen Prozessoren, wenn mehrere Prozessoren kernel-Code ausführen können.

Als Abhilfe sind nun zwei Möglichkeiten denkbar:

- Die Monitoreigenschaft des Unix-Kerns wird beibehalten.
- Es werden alle kritischen Abschnitte im Kern lokalisiert und Synchronisationsmaßnahmen eingebaut.

Die erste Alternative führt zu einem sogenannten Master-Slave System. Auf diesen Systemen darf nur eine CPU, genannt der Master, Prozesse im Systemmodus ausführen. Die anderen CPUs, genannt Slaves, arbeiten nur im Benutzermodus. Der Master übernimmt alle Systemaufrufe und Interrupts.

Wenn ein Prozeß auf einem Slave einen Systemaufruf durchführt, informiert der Slave den Master darüber. Auf den Slaves laufen die Prozesse nur im Benutzermodus, wo die Konsistenz der Systemdaten garantiert ist, da jeder Prozeß nur seine eigenen Daten bearbeitet. Da aber in einer Unix Softwareentwicklungsumgebung im Mittel 40 Prozent der Rechenzeit auf den Systemmodus entfallen, wird der Kern schon bei einer geringen Anzahl von Prozessoren zum Engpaß. Eine Leistungssteigerung ist nur bei rechenintensiven Applikationen zu erwarten. Die zweite Alternative führt zu sogenannten reintranten Systemen, die maximale Parallelität innerhalb des Kerns erlauben. Das Auffinden aller kritischen Abschnitte verursacht jedoch einen erheblichen Aufwand, da der Kern nicht streng modular strukturiert ist.

Besser ist es, den Kern in größere Blöcke aufzuteilen und nur zu erlauben, daß Routinen in unterschiedlichen Blöcken parallel ablaufen können. Eine Einteilung kann folgendermaßen aussehen:

- Das Dateisystem,
- das Prozeßmanagement,
- die Interprozeßkommunikation,
- die Netzwerkanbindung,
- jeder einzelne Treiber.

Diese Blöcke werden als separate Monitore realisiert. Die Monitormechanismen müssen in die unterste Schicht des Betriebssystems aufgenommen werden und interagieren eng mit der Prozeßverwaltung.

Ein spezielles Problem des Monitorkonzepts zeigt sich, wenn ein Prozeß, der in einem Monitor A arbeitet, einen Monitor B betreten will. Wird A beim Betreten von B nicht freigegeben, ist die Gefahr von Verklemmungen sehr hoch. Außerdem behindern sich die Prozesse zu stark. Für den Unix-Kern gilt jedoch, daß die Daten eines Monitors in einem konsistenten Zustand sind, wenn ein anderer gerufen wird. Betrachtet man als Beispiel das Betreten eines Treibers aus A, so ist dies offensichtlich, denn der Kern muß immer damit rechnen, daß der Treiber den aufrufenden Prozeß blockiert. Dabei gibt auch im Monoprozessorsystem der Kern den Monitor auf und setzt somit die Konsistenz der Daten voraus.

Monitore werden durch Semaphore implementiert, für deren effiziente Realisierung die Hardware "Test und Set" Instruktionen bereitstellen sollte. Die meisten Multiprozessoren haben solche Maschinenbefehle, die je nach Hersteller auch "read and clear" oder "compare and swap" heißen können. Diese Anweisung prüft eine angegebene Speicherzelle im Arbeitsspeicher und setzt sie auf Null. Der Rückgabewert zeigt an, ob die Zelle vorher auf Null oder auf Eins stand. Wenn mehrere Prozessoren diesen Befehl gleichzeitig ausführen, so erhält nur ein Prozessor den Wert Eins und die übrigen lesen Null. Die Hardware sichert also die Unteilbarkeit des Lese- und Setzvorgangs.

Mit dieser Anweisung werden Spinlocks aufgebaut, die innerhalb der Prozeßverwaltung angewendet werden. Ein Spinlock führt den Test und Set-Befehl auf eine Sperrvariable solange aus, bis der Rückgabewert Eins beträgt. Es handelt sich hier also um ein aktives Warten, das zur Koordinierung von sehr kurzen kritischen Abschnitten eingesetzt wird. Unter diesen sind besonders die Zustandsänderungen von Prozessen hervorzuheben. Der Zustand wird nicht nur vom Prozeß selbst, sondern auch durch andere Prozesse geändert. Dabei besteht das Problem, daß Zustandsänderungen von Bedingungen abhängig sein können, die sich in der Zwischenzeit geändert haben. So kann es vorkommen, daß ein Prozeß die CPU freigibt und P als Nachfolger bestimmt und der Swapper gleichzeitig P zum Auslagern auswählt. Diese Probleme lassen sich lösen, indem man eine Gruppe von Status-Transfer-Funktionen einführt, die man als neue Schicht der Prozeßumschaltung unter die alten Funktionen *sleep, wakeup* und *setrun* einschiebt.

Die Untersuchung dieser Probleme wurde schon vor einiger Zeit begonnen ([Janssens85] und [Regn88]) und man kann davon sprechen, daß diese Systeme nun dem Forschungsstadium entwachsen sind. Einige Firmen bieten Rechner an, die mit solchen Systemen arbeiten. Genannt sei hier das DYNIX System von Sequent und das DG/UX der Data General AViiON Reihe. Seit kurzem gibt es auch von Sun Microsystems ein solches System, genannt Solaris 2.0.

1.2 Netztopologien

Von einem verteilten System spricht man, wenn mehrere eigenständige Rechner über ein Netzwerk zusammengeschaltet sind. Sie verfügen über keinen gemeinsamen Speicher. Der Vorteil dieser (lose gekoppelten) Systeme liegt darin, daß sie wesentlich weiter ausbaufähig sind als die eng gekoppelten Systeme, bei denen die Bandbreite des Busses

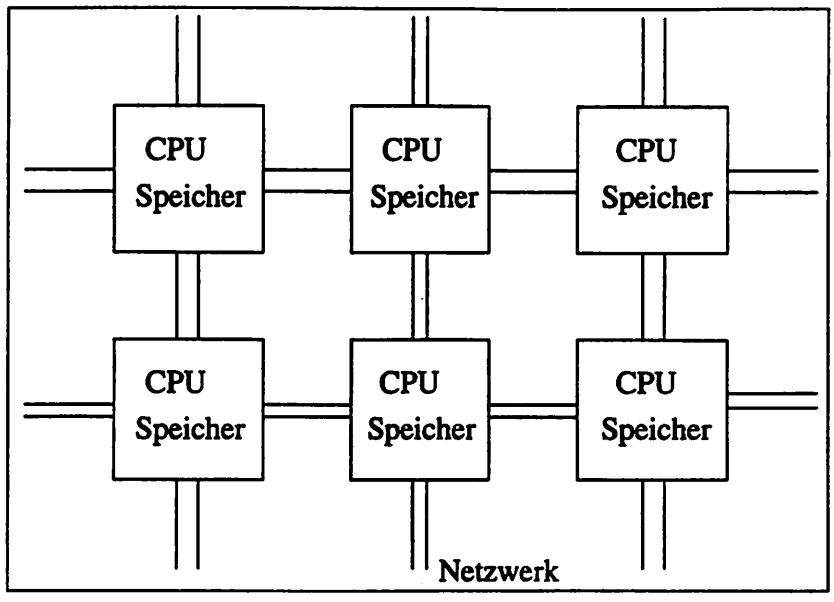

Abbildung 1: Beschränkte Nachbarschaft

bald zum Engpaß wird. Verteilte Multiprozessoren kommunizieren über spezielle Verbindungselemente, die logisch wie E/A-Operationen zu bedienen sind. Heute ist dieses Verbindungselement ausschließlich ein Netzwerk. Die Kommunikation, die vom Kern realisiert wird, ist hier wesentlich aufwendiger als in speichergekoppelten Systemen. Man kann 2 Topologien unterscheiden:

1.2.1 Busstruktur

Für die Verbindung von Rechnern wird ein lokales Netzwerk (*local area Network, LAN*) benutzt. Das z.Zt. gängigste Netzwerk, Ethernet, verbindet Rechner über ein Koaxialkabel in einer Busstruktur. Wesentlich dabei ist, daß der Kommunikationsaufwand zwischen 2 Rechnern immer gleich hoch ist, egal an welcher Stelle im Netz sie sich befinden.

1.2.2 Beschränkte Nachbarschaft

Von beschränkter Nachbarschaft spricht man, wenn die Rechner in solcher Weise angeordnet sind, daß sie unmittelbaren Kontakt nur zu ihren direkten Nachbarn haben (typischerweise 4 Nachbarn).

Die Übertragung von Information zu weit entfernten Knoten erfordert hohen Aufwand, da alle dazwischenliegenden Knoten die Information aktiv übertragen müssen. Die Anordnung kann matrix-, baum-, stern- oder ringförmig sein. Diese Topologie wird meist für Rechner benutzt, die gezielt für die Parallelisierung von Anwenderprogrammen gedacht sind (z.B. Suprenum). Auch der Hypercube gehört in diese Kategorie. Der Vorteil der beschränkten Nachbarschaft ist, daß der Verbindungsaufwand nur logarithmisch mit der Anzahl der Rechner steigt. Für verteilte Betriebssysteme wird eine busförmige Verbindungsstruktur bevorzugt.

2 Ziele eines verteilten Betriebssystems

Welche Vorteile hat nun ein verteiltes Betriebssystem? Vor 15 Jahren war es so, daß ein Rechner mit doppelter Rechenleistung weniger als das doppelte des Rechners mit einfacher Rechenleistung kostete. Die Folge war, daß vorwiegend Großrechner (Mainframes) betrieben wurden. Mit der Verbreitung der Mikroprozessoren ist das Gegenteil richtig: Das beste Preis/Leistungsverhältnis haben kleine Rechner.

Durch die Verbindung vieler CPU's lassen sich Gesamtleistungen erreichen, die auf einem einzigen Prozessor aus physikalischen Gründen (Verbindungswege, Wärmeentwicklung) niemals möglich sein können. Ein anderer Vorteil verteilter Systeme ist die Vergrößerbarkeit. Wenn die Rechenleistung eines Mainframes nicht mehr ausreicht, muß ein größerer unter hohen Kosten angeschafft werden. Einem verteilten System lassen sich leicht entsprechend den Erfordernissen weitere Rechner hinzufügen.

Auf solchen verteilten Systemen werden heute noch keine optimalen Betriebssysteme, sondern nur angepaßte Einzelrechnerbetriebssysteme gefahren.

Es gibt verschiedene Entwurfsaspekte für verteilte Betriebssysteme. Allem voran wird eine größtmögliche Transparenz des Systems angestrebt, d.h. das Betriebssystem soll so aussehen wie ein normales zentralistisches Betriebssystem, aber auf mehreren Rechnern laufen. Die Transparenz bezieht sich auf:

Zugriff: Für lokale und globale Daten wird dieselbe Zugriffsmethode verwendet

Ort: Der Zugreifer weiß nicht, wo die Information sitzt – er greift nur durch logische Namen zu

Replikation: Falls Daten repliziert sind, um Zugriffe zu beschleunigen, ist das für den Benutzer unsichtbar.

Migration: Objekte können von einem Knoten zu einem anderen gebracht werden, ohne daß der Benutzer dies bemerkt oder sich Zugriffsmethoden ändern.

Ausfallsicherheit: Der Ausfall einzelner Komponenten legt nicht das ganze System lahm.

Die Rechner sollen also einen virtuellen Monoprozessor bilden. Es gibt eine Menge von Systemen, die Teile dieser Anforderungen erfüllen, aber wenige erfüllen alle Kriterien. Als einfache Regel kann man aufstellen: Wenn jemand sagen kann, welchen Rechner er benutzt, dann hat er kein verteiltes Betriebssystem.

Der Begriff verteiltes Betriebssystem muß streng vom Begriff Netzwerkbetriebssystem getrennt werden. Ein Netzwerkbetriebssystem läuft auf einer Anzahl von Rechnern, die durch ein Netzwerk verbunden sind. Es unterstützt einfache Kommunikation zwischen Rechnern und Benutzern des Gesamtsystems. Peripheriegeräte wie Drucker können gemeinsam benutzt werden und es gibt einen Dateiservice. Solche Systeme haben aber einige der nachstehenden Eigenschaften, die verteilte Betriebssysteme nicht haben:

- jeder Knoten fährt sein privates Betriebssystem anstelle eines Teils eines globalen Systems

- Jeder Benutzer arbeitet an seinem eigenen Rechner, es gibt "remote login"
- Die Benutzer wissen, wo ihre Daten liegen und müssen sie mit speziellen Filetransfer-Kommandos zwischen den Knoten übertragen.
- Das System hat keine Fehlertoleranz

Von frühen Versuchen der Rechnerverbindung abgesehen kam der Begriff Netzwerk in den 70er Jahren mit dem ARPANET ins Bewußtsein. Ursprünglich wurde aber keineswegs erwogen, ein Netzwerkbetriebssystem zu schaffen. Das Netzwerk bestand nur aus Telefonleitungen für remote-login und Filetransfer. Mitte der 80er Jahre wurde die Vernetzung mehr in das Betriebssystem und schließlich in den Kern hineingebracht.

Der entscheidende Aspekt zur Unterscheidung von Netzwerkbetriebssystem und verteiltem Betriebssystem ist, ob der Benutzer mit der Tatsache belastet wird, daß mehrere Maschinen benutzt werden. Dieser Aspekt erscheint in 3 Bereichen: Filesystem, Zugriffsschutz und Programmausführung.

Verbindet man 2 Maschinen, so ergibt sich das Problem, wie deren Filesysteme zu verbinden sind. Abgesehen von der trivialen Möglichkeit, sie nicht zu verbinden, gibt es verschiedene Ansätze, die ein unterschiedliches Maß an Transparenz gestatten. Für den Zugriffsschutz ist unter Unix jedem Benutzer eine Kennummer (*uid*) zugeordnet. Von einem verteilten Betriebssystem muß erwartet werden, daß ein Benutzer in allen Prozessoren dieselbe *uid* hat; damit werden die Zugriffsschutzprobleme elegant gelöst.

Wenn ein Benutzer einen neuen Prozeß erzeugen will, wo wird dieser erzeugt? Netzwerkbetriebssysteme haben die Möglichkeit, ein Programm auf einem entfernten Knoten zu starten. In vielen Fällen arbeiten Signale und andere Mittel der Interprozeßkommunikation dann nicht mehr so, wie wenn der Prozeß lokal laufen würde. Man wird jedoch erwarten, daß ein verteiltes Betriebssystem die Verteilung der Prozesse – für den Benutzer unsichtbar – selbst vornimmt.

2.1 Entwurfskriterien

2.1.1 Kommunikation

Da die Prozessoren eines verteilten Betriebssystems über keinen gemeinsamen Speicher verfügen, sind Synchronisationstechniken wie Monitore und Semaphore generell nicht anwendbar. Ein weitverbreitetes Modell für Kommunikation ist das ISO-OSI Schichtenmodell. Allerdings ist der Overhead in diesem Modell sehr groß. Verbindet man Mainframes über schwache Leitungen, so mag das tolerabel sein, da genug Rechenleistung zur Verfügung steht. Der heutige Standard in Netzwerken ist aber 10 MB/sek., so daß viel einfachere Kommunikationsmechanismen nötig sind.

Message-passing Wenn zwei Rechner Informationen austauschen, so kann man diesen Dialog rein symmetrisch betrachten. Man kann aber auch die Asymmetrie hervorheben, die dadurch entsteht, daß der Dialog immer von einem Knoten initiiert wird, der irgendeine Anfrage oder einen Auftrag hat. Dieser Knoten – genauer der Prozeß – wird

Klient genannt. Der andere Knoten bzw. Prozeß bietet eine Dienstleistung an und wartet auf Aufträge. Er ist der Server[1] in dieser Konfiguration, die Client/Server-Modell genannt wird. Auf einem Rechner können dabei mehrere Server und viele Klienten laufen.

Ein Beispiel für einen Fileserver und einen Klienten, der eine Datei kopiert, ist in Bild 2 in ein Programm gefaßt, wobei nur die oberste Ebene des Servers wiedergegeben ist. Auch sind Details sowie die Fehlerbehandlung weggelassen, um das Konzept besser zu sehen. Ein nicht gezeigtes Headerfile definiert die Elemente der Struktur "message" und die Konstanten READ, WRITE, etc.

Für die Kommunikation stellt man 2 Primitiva zur Verfügung: *Send* und *receive*. *Send* spezifiziert das Ziel und enthält eine Nachricht, *receive* sagt, woher eine Nachricht erwartet wird und stellt einen Puffer zur Verfügung. Es ist kein Verbindungsaufbau nötig. So verblüffend einfach dies klingt, so problematisch wird es, wenn man versucht, die Semantik dieser Operationen exakter zu spezifizieren.

Zum einen können diese Operationen **zuverlässig** oder unzuverlässig sein. *Send* kann eine Nachricht in das Netz geben und keine Garantie bieten, daß sie wirklich angekommen ist. Wird sie verloren, wird keine neue Übertragung versucht. Es obliegt allein dem Benutzer, die Zuverlässigkeit der Übertragung zu implementieren. Ein zuverlässiges *send* erwartet eine Bestätigung und kümmert sich um nötige Übertragungswiederholungen. Wenn es endet, ist der Programmierer sicher, daß die Nachricht zugestellt worden ist.

Die Bestätigung (acknowledge) läuft ausschließlich von Kern zu Kern und wird vom Benutzer nicht gesehen. Da auf einen Auftrag des Klienten meist bald eine Antwort des Servers folgt, kann die Bestätigung für das *send* des Klienten auch eingespart werden. Der Kern des Servers kann hier die Kontrolle übernehmen und, falls der Server länger für den Auftrag benötigt, eine Bestätigung für den Auftrag senden.

Auch die Antwort des Servers muß bestätigt werden. Es gibt jedoch Aufträge, die einfach wiederholt werden können, falls die Antwort ausbleibt. Dazu gehören Anfragen nach einem bestimmten Block in einer Datei. Solche Nachrichten werden **idempotent** genannt, d.h. wiederholtes Senden ändert die Semantik nicht. Ein Gegenbeispiel wäre der Überweisungsauftrag an eine Bank.

Zum Zweiten können die Daten **gepuffert** oder ungepuffert übertragen werden. Bei der ungepufferten Übertragung wird das Senden blockiert, wenn der Empfänger gerade kein *receive* aufgerufen hat. Erst beim *receive* des Empfängers wird die Nachricht übertragen. Diese Technik wird Rendezvous-Prinzip genannt. Möchte man das Blockieren vermeiden, kann der Kern eine oder mehrere Nachrichten für den Empfänger zwischenspeichern, er verwaltet dann eine Mailbox, die bei einem Server praktisch eine Auftragswarteschlange darstellt. Pufferung ist komplexer, da sie die Handhabung der Puffer erfordert, und führt zudem zu einer Reihe von Problemen, z.B.: Wenn der Pufferbereich des Empfängers voll ist, sieht man sich mit derselben Situation konfrontiert wie im ungepufferten Fall. Das Problem ist also nicht grundsätzlich gelöst. Außerdem müssen Nachrichten behandelt werden, die zu inzwischen beendeten Prozessen gehören.

Zum Dritten kann die Übertragung **blockierend** oder nichtblockierend sein. Beim blockierenden (synchronen) Senden wird der Sender solange blockiert, bis die Nachricht

[1] Dieses Wort wird hier nicht übersetzt, da es in die Fachsprache Eingang gefunden hat.

```
#include <header.h>
void main()
{
  struct message m1, m2;
  int r;

  while (1) {                          /* Arbeitsschleife des Servers */
    receive(ANY, &m1);                   /* warten auf Auftrag */
    switch(m1.opcode) {
       case READ:       r = do_read(&m1, &m2); break;
       case WRITE:      r = do_write(&m1, &m2); break;
       default:         r = E_BAD_OPCODE;
    }                       /* analog: create, delete, link .. */
    m2.result = r;
    send(m1.source, &m2);                /* sendet Antwort */
  }
}

#include <header.h>
int copy(char *src, char *dst)           /* kopiert eine Datei */
{
  struct message m1;
  long position;                         /* Dateiposition */
  long result;

  initialize();                          /* Oeffnen der Datei */
  position = 0;
  do {
    m1.opcode = READ;                    /* einen Block lesen */
    m1.offset = position;
    m1.count  = BUF_SIZE;
    strcpy(&m1.name, src);
    send(FILE_SERVER, &m1);              /* Auftrag senden */
    receive(FILE_SERVER, &m1);           /* Antwort erwarten */

    if ((result = m1.result) >0)     /* result ist die tat- */
       {                          /*  saechlich gelesene Bytezahl */
       m1.opcode = WRITE;
       m1.offset = position;
       m1.count  = m1.result;
       strcpy(&m1.name, dst);
       send(FILE_SERVER, &m1);           /* Auftrag senden */
       receive(FILE_SERVER, &m1);        /* Antwort erwarten */
       position += result;
       }
  } while (result > 0);                  /* solange, bis Dateiende */
  return(result >= 0 ? OK : result);/* OK oder Fehler zurueckgeben */
}
```


Abbildung 2: Client- und Serverprogramm

abgesandt ist. Analog blockiert das synchrone *receive*, bis eine Nachricht empfangen und in den angegebenen Puffer gebracht worden ist. Ein nichtblockierendes *send* kehrt sofort zurück, nachdem die Nachricht für die Übertragung vorgemerkt ist. Nun kann der Prozeß parallel zur Nachrichtenübertragung laufen. Dabei besteht aber die Gefahr, daß die Nachricht vorher oder sogar während der Übertragung verändert wird. Nun kann der sendende Prozeß aber nicht wissen, wann die Übertragung beendet ist; er kann aber auch nicht auf die Wiederverwertung des Puffers verzichten. Ein Ausweg besteht darin, daß der Kern den Puffer kopiert und dann den Prozeß weiterlaufen läßt. Für den Sender ist es dann so, als ob die Nachricht gesandt worden wäre, obwohl sie sich noch im Kern befindet. Es ist aber eine zusätzliche Kopie notwendig (später wird die Nachricht noch in den Puffer des Netzwerkcontrollers kopiert), die eigentlich unnötig ist und den Durchsatz der Netzwerkschnittstelle reduziert.

Die Alternative besteht darin, das Programm zu unterbrechen, um anzuzeigen, daß der Puffer wiederverwendet werden kann. Hier wird der Kopiervorgang unnötig, so daß die Nachricht schneller gesandt werden kann. Man kann sich ebenfalls ein asynchrones *receive* vorstellen: Das nichtblockierende *receive* stellt einen Puffer zur Verfügung und kehrt zurück. Wenn die Nachricht eingetroffen ist, muß der Prozeß irgendwie informiert werden. Dazu kann ein eigener *wait*-Aufruf vorhanden sein, der dann blockierend ist. Falls das nicht gewünscht ist, kann auch eine Variable abgefragt werden, die der Empfänger beim *receive* spezifiziert hat, und die der Kern setzt, wenn die Nachricht eingetroffen ist. Zuletzt gibt es wie beim *send* die Möglichkeit, eine Unterbrechung vorzusehen: Ist die Nachricht eingetroffen, wird das Programm unterbrochen.

Der Vorteil der nichtblockierenden Primitiva ist sicher die hohe Flexibilität. Ihre Programmierung ist aber trickreich und schwierig. Man erhält zeitkritische Programme, die schwer zu erstellen und wegen ihrer nicht reproduzierbaren Fehler kaum noch zu debuggen sind. Die nichtblockierenden Primitiva benutzen den Nachrichtenaustausch, um Parallelität zu erhalten. Konsequenterweise sind die blockierenden Primitiva zu bevorzugen: *Send* kehrt erst zurück, wenn die Nachricht gesandt worden ist (und falls ein Acknowledge gekommen ist – im zuverlässigen Fall) und *receive* blockiert, bis die Nachricht verfügbar ist.

Unter dem Client/Server-Modell betrachtet, lassen sich die Kommunikationsprimitiva auch so spezifizieren, daß die Asymmetrie zwischen Server und Klienten zum Ausdruck kommt: *Sendrec* benutzt der Klient, um eine Anforderung zu senden und auf die Antwort zu warten, der Server benutzt *receive*, um Aufträge entgegenzunehmen und *reply*, um zu antworten. Diese Aufrufe sind alle blockierend. Durch die Kombination des *sendrec* wird die Interaktion des Klienten mit dem Systemkern effizienter, da nur ein Systemaufruf ausgeführt werden muß.

Remote Procedure Call Aus der Sicht des Klienten ähnelt dieses *sendrec* stark einem Prozeduraufruf. Diese Modellierung ist in der Literatur als Remote Procedure Call bekannt. Da der lokale Prozeduraufruf als Mittel zur Abstraktion vertraut und von allen Programmierern verstanden ist, liegt es nahe, den entfernten Prozeduraufruf in seiner Semantik dem lokalen so ähnlich wie möglich zu gestalten.

Insbesondere verdeckt der RPC, daß eigentlich eine Kommunikation zwischen verschiedenen Knoten aufgebaut werden muß. *Send* und *Receive* sind ja E/A-Operationen. Das Verdecken dieser E/A erhöht die Transparenz.

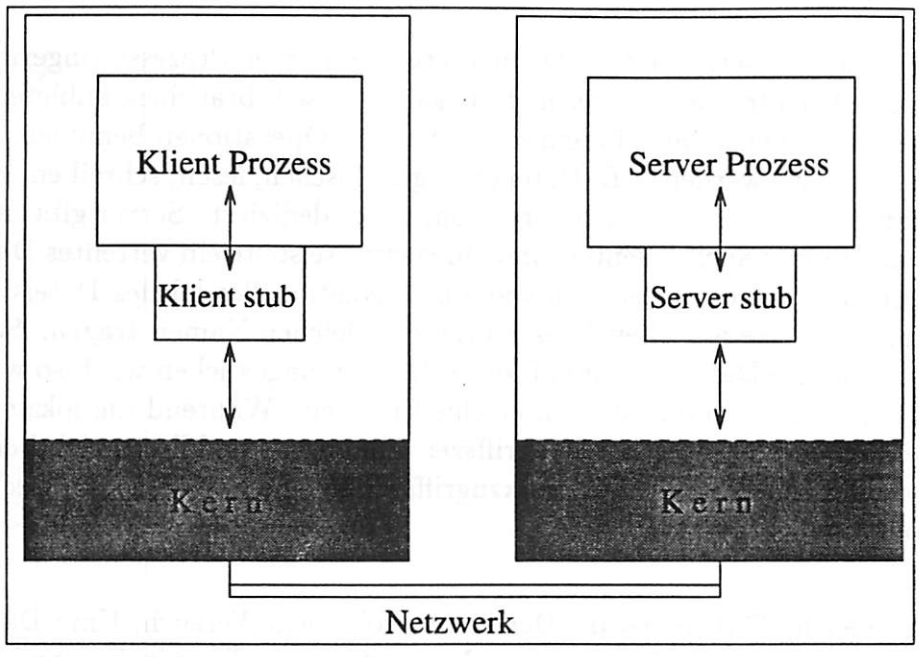

Abbildung 3: Stubprozesse

Im Prinzip wird beim RPC eine Prozedur auf einem entfernten Knoten aufgerufen. Wie gewohnt, wird der Aufrufer solange blockiert, bis die Prozedur abgearbeitet ist. Informationen können als Parameter übergeben und als Resultat zurückgegeben werden.

Der Remote Procedure Call kann als Kombination von zuverlässigen blockierenden *sendrec*, *receive* und *reply* realisiert werden. Im Klienten-Programm wird nur eine Prozedur P gerufen. Der Nachrichtenaustausch wird von einer sogenannten Stub-Prozedur P durchgeführt, die ihre Parameter in eine Nachricht verpackt und diese an den Server schickt.

Auf dem entfernten Prozessor wartet eine andere Stub-Routine durch ein *receive* auf eingehende Anforderungen. Sie packt die Parameter aus und führt dann den lokalen Aufruf von P aus. Außer den beiden Stubs weiß niemand, daß ein entfernter Aufruf durchgeführt wurde (Bild 3). Wenn P beendet ist, läuft das Ergebnis den gleichen Weg zum Klienten zurück.

So sauber und einfach das Konzept des RPC aussieht, hat es dennoch einige Probleme. Sie resultieren daher, daß rufende und gerufene Prozedur auf verschiedenen Rechnern in verschiedenen Adreßräumen ablaufen. Das größte Problem ist die Parameterübergabe. Wurden die Parameter *by value* übergeben, so gibt es keine Probleme: Sie werden einfach vom Stub in die Message kopiert. Werden Parameter *by reference* – also Zeiger – übergeben, tritt das Problem auf, daß diese Zeiger auf dem entfernten System nicht mehr gültig sind. Ein weiteres Problem ist die Darstellung von Parametern und Ergebnissen in der Nachricht. Falls die beteiligten Maschinen Datentypen unterschiedlich im Arbeitsspeicher ablegen, kommt man um eine Konvertierung nicht umhin.

2.1.2 Dateiservice

Für Dateizugriffe werden eine oder mehrere Dateiserver-Prozesse eingerichtet, die ihre Dienste den Klienten, die vorher nicht bekannt zu sein brauchen, anbieten. Die Client-Prozesse rufen Dienste auf, indem sie bestimmte Operationen benutzen, die durch ein Interface definiert werden; z.B. Datei erzeugen, löschen, lesen, schreiben, Link erstellen. Man kann sich Konfigurationen vorstellen, wo es dedizierte Server gibt, aber auch solche, wo alle Server auch Klienten sind. Idealerweise sollte ein verteiltes Dateisystem für die Benutzerprozesse so aussehen wie ein konventionelles lokales Dateisystem. Daher sollten alle Objekte auf allen Prozessoren die gleichen Namen tragen. Sosehr der Zugriff auf entfernte Dateien dem auf lokale Dateien angeglichen wird, so wird aber doch in der Performance immer ein Unterschied bleiben. Während die lokale Performance im wesentlichen von der Plattenzugriffszeit abhängt, so addieren sich bei den entfernten Operationen Platten-, Netzwerkzugriffszeiten und verschiedene Rechenzeiten der Protokolle.

Die Newcastle Connection Der erste erfolgreiche Versuch, Unix Dateisysteme zu verbinden, wurde an der University of Newcastle upon Tyne in England durchgeführt. Die Dateisysteme mehrerer Rechner werden dabei verbunden und an eine sogenannte Superroot gehängt. Diese Superroot wird mit "/.." bezeichnet. Beginnt man einen Suchpfad mit "/../Maschine_x", so kann man alle Dateien auf der Maschine_x ansprechen. Der Vorteil dieses Namensschemas ist, daß Dateinamen ganz konventionell gebildet werden können, ohne daß ein neues, spezielles Zeichen für entfernten Zugriff eingeführt werden müßte.

Die Besonderheit an der Newcastle Connection ist, daß sie nicht im Kern, sondern in den C-Bibliotheksfunktionen realisiert ist. Alle Funktionen, die mit Dateizugriffen zu tun haben, untersuchen den Pfad auf den String "/.." und kommunizieren dann mit einem Stub-Prozeß auf dem Server, der beim ersten entfernten Zugriff eines Prozesses dort erzeugt wird. Dieser sollte die gleichen Zugriffsrechte zu Dateien haben, die der Benutzer auf der entfernten Maschine hat. Daher sollte der Stub unter der Benutzernummer des lokalen Benutzers laufen und die Benutzernummern auf den Systemen sollten abgeglichen sein. Diese Vorgehensweise führt zu Problemen beim Benutzer *root*, dessen Rechte sich nicht über mehrere Maschinen erstrecken sollten. Allerdings gibt es einige Programme, die nicht korrekt arbeiten, wenn sie auf dem entfernten Rechner keine root-Rechte besitzen; zum Beispiel das Programm *mkdir*, das ein neues Directory erzeugt. Dieses Problem läßt sich durch die Einführung eines *mkdir* Systemaufrufes lösen, wie es im BSD-System der Fall ist.

Dem Vorteil, daß im Kern keine Änderungen durchzuführen sind, steht der Nachteil gegenüber, daß die Anforderungen auf dem Server durch Stub-Prozesse erfolgen, die im Usermode laufen, was den Dateiservice verlangsamt, da die Daten zuerst vom kernel- in den user-Bereich und anschließend wieder zurückkopiert werden. Wenn ein entfernter *open*-Aufruf erfolgreich ist, vergibt die Bibliotheksfunktion einen eigenen Filedeskriptor, den sie selbst verwaltet. Die Bibliotheksfunktionen für *read* und *write* prüfen, ob der übergebene Filedeskriptor zu einem entfernten File gehört und senden in diesem Fall eine Nachricht an den Stub, anstelle den normalen Systemaufruf durchzuführen. Ein Stub bedient alle Anforderungen eines entfernten Prozesses. Wenn mehrere entfernte Prozesse Dateizugriffe durchführen, so wird für jeden ein Stub erzeugt. Ebenso kann

es sein, daß für einen Prozeß mehrere Stubs erzeugt werden, wenn dieser Prozeß auf verschiedenen entfernten Rechnern Dateizugriffe durchführt.

Wenn ein Prozeß *fork* aufruft, so sendet die Bibliotheksfunktion jedem Stub eine *fork*-Nachricht. Der Stub führt dann ein *fork* aus und sein Sohnprozess bedient fortan den lokal erzeugten Sohnprozess. Die *exit* Bibliotheksfunktion andererseits enthält zunächst eine *exit* Nachricht an alle zugehörigen Stubs, bevor lokal das *exit* ausgeführt wird. Ein *chdir*-Aufruf, der einen entfernten Pfad enthält, führt zu einer Nachricht an den Stub, der sein current directory entsprechend ändert, und die Bibliothek notiert, daß das momentane Directory entfernt ist. Ein lokaler *chdir*-Aufruf muß hier überhaupt nicht durchgeführt werden. Endet der Prozeß durch ein Signal und bekommt einen Speicherabzug (core), so gibt es aber keine Möglichkeit, dieses core-File dann auf dem entfernten Dateisystem zu haben. Noch komplizierter ist der *exec*-Aufruf mit einem entfernen Pfad. Da der Kern nicht geändert ist, ist es nicht möglich, ein Programm, das auf einem fremden Rechner liegt, aufzurufen; also muß das Programmfile zuerst von der Bibliothek auf den lokalen Rechner kopiert werden. Da der Status der entfernten offenen Dateien nur in den Tabellen der Bibliothek enthalten ist, müssen die Tabellen erst gesichert werden, da sie bei einem *exec* überschrieben werden.

Die Netzwerkfähigkeit wird also mit einer Reihe von Nachteilen erkauft. Zum einen wird die C-Bibliothek wesentlich größer, da sie die gesamte Implementation der Newcastle Connection enthält. Auch Programme, die lediglich lokale Dateianforderungen haben, werden verlangsamt, da ständig Filedeskriptoren und Pfade geprüft werden müssen. Programme, die das Netzwerk benutzen wollen, müssen neu gebunden werden. Daher ist auch gekaufte Software, die nicht in der Quelle vorliegt, nicht netzwerkfähig. Dieser Nachteil wird nur vermieden, wenn das System dynamisch gebundene Bibliotheken vorsieht.

Wie ist nun die Namenstransparenz der Newcastle Connection zu beurteilen? Man unterscheidet zwischen Ortstransparenz und Ortsunabhängigkeit. Ortstransparenz bedeutet, daß der Name einer Datei keine Hinweise auf seine physikalische Lage gibt, Ortsunabhängigkeit bedeutet, daß der Name nicht geändert werden muß, wenn die Datei ihren Ort wechselt. Da die Newcastle Connection den Ort von Dateien nicht selbständig wechselt, entfällt die Bewertung des zweiten Punktes. Aber auch eine Ortstransparenz ist nicht gegeben, da der Dateiname ja Hinweise auf den Rechner enthält, auf dem die Datei liegt. An ein Betreiben von plattenlosen Workstations ist nicht zu denken. Da es im Klienten keinen Cache gibt, ist die Performance bei kleinen Schreib- und Leseanforderungen schlecht. Bei einem Ausfall des Servers müssen Programme, die auf diesen Server zugegriffen haben, abgebrochen werden.

Transparent verteilte Dateisysteme Bei einem transparent verteilten Dateisystem muß zumindest die Ortstransparenz gegeben sein. Um ein entferntes Dateisystem flexibel an ein lokales zu binden, bietet sich die Erweiterung des bekannten *mount*-Systemaufrufes an. Man kann dabei beliebige Unterbäume eines entfernten Systems an beliebige Stellen des lokalen Dateisystems binden. Ähnlich wie beim herkömmlichen *mount* wird auch hier die inode, auf die montiert wird, markiert, so daß Zugriffe, die diese inode kreuzen, zu Netzwerkoperationen führen (Bild 4). Ein besonderes Problem tritt auf, wenn der Pfadname ".." enthält. Hat der vordere Teil des Namens schon zum entfernten Dateisystem geführt, sollte eine wiederholte Benutzung von ".." ins lokale

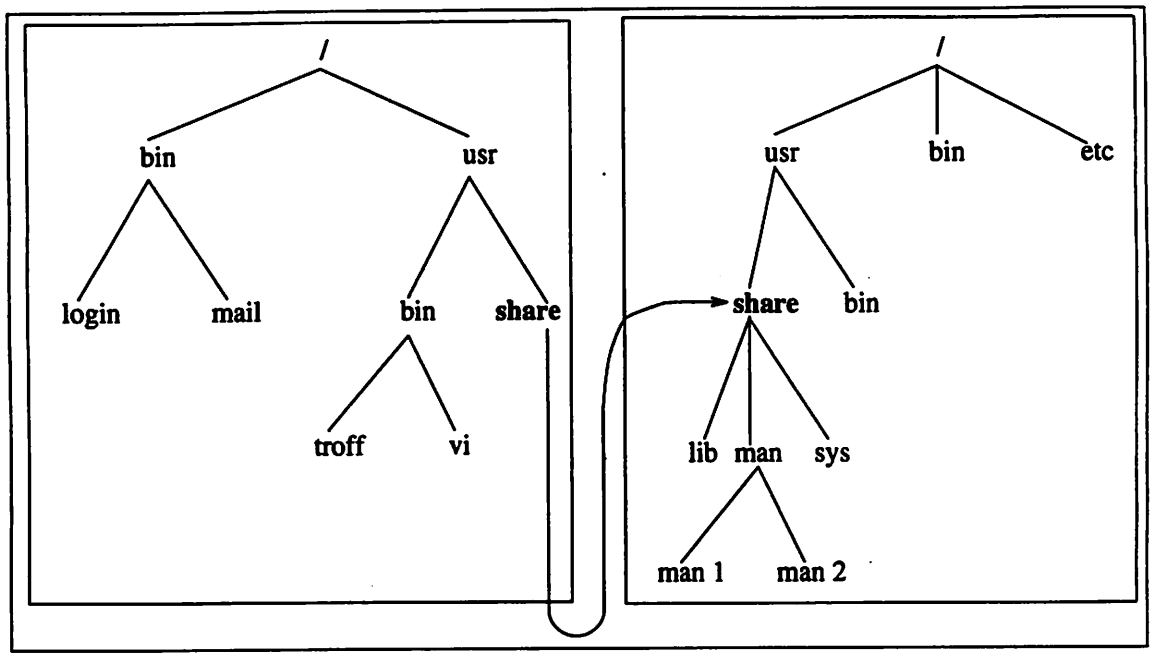

Abbildung 4: Montieren von Dateisystemen

Dateisystem zurückführen, anstelle Zugriffe über dem entfernt montierten Directory zu erlauben.

Zur Kommunikation kann zwischen zwei Modellen gewählt werden, remote procedure call und remote system call. Beim remote procedure call prüft jede kernel-Prozedur, die mit inodes umgeht, ob eine spezielle inode zu einem lokalen oder einem entfernten File gehört und sendet im letzteren Fall eine Nachricht zur entfernten Maschine, um eine spezielle inode-Operation durchzuführen. So kann ein Systemaufruf, der auf ein entferntes File zugreift, zu mehreren Nachrichten über das Netzwerk führen, abhängig davon, wieviele inode-Operationen durchgeführt werden müssen. Dadurch ergibt sich eine größere Antwortzeit und eine stärkere Netzauslastung. Verschiedene Optimierungen dieses puren Modells wurden gefunden, um verschiedene inode-Operationen in eine einzelne Nachricht zu kombinieren und um wichtige Daten in einen Cache zu legen.

Im Falle des remote-system-calls erkennt der lokale Kern, daß ein Systemaufruf ein entferntes File betrifft und sendet den Aufruf mitsamt den Parametern zum Server, der diesen Systemaufruf durchführt und die Ergebnisse zum Klienten zurückbringt. Der Klient kann dann gleich mit einem *longjmp* aus dem Systemaufruf in den Benutzermodus zurückspringen. Die meisten Systemaufrufe können mit einer einzigen Netzwerknachricht durchgeführt werden, was zu einer guten Antwortzeit führt. Aber einige Kernoperationen passen nicht in dieses Modell. Zum Beispiel erzeugt der Kern ein Core-File für einen Prozeß, wenn dieser bestimmte Signale erhält. Dieses Erzeugen des Core-Files korrespondiert nicht mit einem Systemaufruf, aber führt zu verschiedenen inode Operationen. Im Beispiel eines *open*-Systemaufrufes geht der lokale Kern den Pfad durch, bis er erkennt, daß dieser sich auf eine entfernte Datei bezieht und schickt dann den Rest des Pfadnamens zusammen mit verschiedenen Informationen wie Benutzer- und Gruppen-ID des Prozesses an den zugehörigen Server. Der neugebildete Eintrag in der kernel-file-table kann dann zu einer dummy inode zeigen, die anzeigt, auf welcher Maschine die tatsächliche inode liegt. So kann bei *read*- und *write*-Aufrufen, die sich auf diesen Filedeskriptor beziehen, eine Nachricht an den Server geschickt werden.

Transparent verteilte Dateisysteme ohne Stubprozesse Das Verwenden von Stubprozessen auf dem Server führt zu einer einfachen Implementierung der Serverseite des verteilten Dateisystems. Jedoch wird die Prozeßtabelle des Servers mit Stubprozessen, die zumeist untätig sind, überfüllt. Betreibt man eine Workstation plattenlos, so können dafür auf den Servern leicht Dutzende von Stubprozessen nötig sein. Besser ist es, einen beschränkten Pool von Serverprozessen zur Verfügung zu haben, die man temporär den eingehenden Anforderungen zuweist. Nach Ausführung der Anforderung kommt der Serverprozeß in den Pool zurück und ist für weitere Anforderungen verfügbar. Er merkt sich deswegen keinen Benutzerkontext und keine Benutzer-ID zwischen Anforderungen, da er Systemaufrufe für verschiedene Prozesse bedienen kann. Die Verbindung zwischen Klient und Server existiert nur für die Dauer einer Anforderung.

Im Vergleich zu Stubs ist das Modell des Serverpools aufwendiger. Wenn zum Beispiel ein Prozessor mit Anforderungen von vielen Klienten überflutet wird, muß er diese Anforderungen in eine Warteschlange einreihen, wenn er nicht genug Serverprozesse hat. Ein Stub dagegen kann nicht mit Anforderungen überflutet werden, da alle Transaktionen mit dem Klienten synchron sind. Ein Klient kann höchstens einen ausstehenden Auftrag haben. Ein Serverprozeß kann nicht ohne weiteres für einen Systemaufruf benutzt werden, der zu einem Blockieren von unvorhersehbarer Dauer führen könnte, z. B. zum Lesen von einem entfernten Terminal. Dadurch geht der Serverprozeß dem Pool effektiv verloren, so daß ernsthafte Engpässe auftreten können.

Trotz der Vorteile des Stubs ist ihr Verbrauch an Prozeßtabelleneinträgen in der Praxis so kritisch, daß die meisten Konzepte einen Pool benutzen, darunter auch das bekannteste, das Sun Networkfilesystem.

Das Sun Networkfilesystem Der Vorgänger des Networkfilesystems (NFS) war die Network-Disk (ND), ein Netzwerk-Plattenserver, der eingeführt wurde, um plattenlose Workstations zu unterstützen. Die Platte auf dem Server war unterteilt in verschiedene Partitionen, und jede Partition war eine virtuelle Platte für eine Workstation.

Wollte diese auf eine Datei zugreifen, wurde die Anforderung verarbeitet, bis sie auf der Ebene des Gerätetreibers war. Dann wurde der nötige Block durch das Senden einer Nachricht an den Server geholt. Effektiv war das Netzwerk hauptsächlich benutzt, um einen Platten-Controller zu simulieren. Das gemeinsame Nutzen von Plattenpartitionen war nicht möglich. Für den Dateiaustausch zwischen Workstations gab es das *remote copy* Programm (rcp).

Die Network-Disk wird heute von Sun nicht mehr unterstützt, da sie durch das NFS abgelöst worden ist. NFS basiert auf dem UDP-Protokoll, das dem Internet-Protokoll übergeordnet ist. Die Schnittstelle – also das Protokoll von NFS – ist von Sun offengelegt, um anderen Herstellern zu ermöglichen, ebenfalls über NFS zu kommunizieren. Um Ortstransparenz zu erreichen, ermöglicht NFS dem Klient, jeden beliebigen Unterbaum eines Dateiservers an jede beliebige Stelle im Klienten zu montieren. Angenommen, auf der Workstation S befinden sich die Home-Directories der Benutzer unter `/home`. Alle anderen Workstations können nun ein leeres Directory `/home` einrichten und mit dem Befehl

```
mount S:/home /home
```

den `/home` Unterbaum des Servers so montieren, daß die Benutzerdaten auf allen Klienten in der gleichen Weise zur Verfügung stehen. Dadurch kann der Benutzer an jeder beliebigen Workstation arbeiten. Die Namen der Dateien enthalten dabei keinen Hinweis, auf welcher Maschine die Daten wirklich sind. Wie dieses Beispiel zeigt, gilt die Transparenz nicht für das *mount*-Kommando selbst. Ein Rechner, der seine Dateien anderen Rechnern zur Verfügung stellen will, spezifiziert in einer Datei `/etc/exports`, welche Dateisysteme welchen Rechnern mit welchen Zugriffsrechten zur Verfügung gestellt werden.

Es ist sogar möglich, das *root*-Filesystem eines Rechners über NFS zu montieren, so daß Rechner ganz ohne eigene Platten betrieben werden können (diskless clients). Damit kann die Systemverwaltung eines größeren Rechnernetzes erheblich vereinfacht werden.

Wenn ein Klient `S:/home` montiert hat, hat er dadurch noch keinen Zugriff auf Dateisysteme, die der Server S unterhalb des Baumes `/home` von anderen Servern montiert hat. Der *mount*-Mechanismus enthält also keine Transitivitätseigenschaft. Vielmehr muß sich der Klient den anderen Teilbaum vom zweiten Server durch einen eigenen *mount*-Aufruf holen. Der Klient kann auf ein Directory X, das sich auf einer anderen Maschine befindet, eine weiteres Dateisystem montieren, so daß ab X der Klient einen ganz anderen Unterbaum hat als der Server.

Eines der Entwurfsziele von NFS war, in einem heterogenen Enviroment von verschiedenen Maschinen und Betriebssystemen zu arbeiten. Daher wurde in NFS, das auf dem RPC-Mechanismus basiert, eine External Data Representation Schicht (XDR) eingezogen, die Konvertierungen zwischen verschiedenen Maschinen durchführen kann. Die NFS Spezifikation unterscheidet zwischen dem eigentlichen Dateizugriff und dem Montiermechanismus, so daß es noch ein separates Mount-Protokoll gibt.

Über das **Mount-Protokoll** wird die erstmalige logische Verbindung zwischen Server und Klient aufgenommen. Der Klient sendet dem Server den Pfadnamen des Directories, das er montieren möchte. Dabei kann jeder Server auch Klient sein, er kann also Dateibäume montieren und selbst welche exportieren; welche spezifiziert er in der Datei `/etc/exports`.

Die Klienten montieren die nötigen entfernten (*remote*) Dateisysteme ebenso automatisch, wie sie lokale Dateisysteme montieren bzw. montieren würden. Entfernte Dateisysteme werden dann in die Konfigurationsdatei des Klienten (z.B. `/etc/fstab` in BSD-UNIX, `/etc/vfstab` bei System V) eingetragen.

Außerdem sind verschiedene NFS-Implementierungen mit dem **automount-daemon** ausgestattet. Er meldet sich beim Kern als NFS-daemon an und kann so entfernte Dateibäume erst bei Zugriff darauf montieren. Er liest eine Konfigurationsdatei, in der eine Gruppe von entfernten Directories mit je einer lokalen Directory assoziiert ist.

Beim ersten Zugriff innerhalb eines solchen Directories sendet er eine Nachricht an alle Server aus der zugehörigen Gruppe von entfernten Directories. Montiert wird dann das Dateisystem des ersten antwortenden Servers. Nach einer gewissen Zeit ohne Zugriff (ca. 5 Minuten) versucht der automount-daemon, das Dateisystem wieder zu entfernen.

Das dynamische Montieren hat Vorteile, wenn Server ausgefallen sind, die zur Zeit nicht gebraucht werden und ein Klient hochgefahren wird. Beim herkömmlichen Montieren führt das zumindest zu Verzögerungen und Fehlermeldungen. Server, die nur vom automount-daemon gebraucht werden, werden beim Hochfahren dagegen nicht angesprochen.

Da ein Klient für jeden Montierpunkt in der automount-Konfiguration mehrere Server eingetragen haben kann, ergibt sich die Möglichkeit, Ersatzserver einzuplanen und so die Ausfallsicherheit zu erhöhen. Dabei müssen jedoch alle alternativen Dateisysteme identisch sein. Da NFS die Replikation in keiner Weise unterstützt und so dem Benutzer das Konsistenzproblem überlassen wird, ist es nahezu unmöglich, diese Fähigkeit des automount-daemon für beschreibbare Dateisysteme zu verwenden. Folglich werden mehrere alternative entfernte Directories nur für read-only Dateisysteme, wie die Systemprogramme und fremdentwickelte Anwenderprogramme, benutzt.

Jedes Directory innerhalb eines exportierten Dateisystems kann von einem Klienten remote montiert werden. Bei einer erfolgreichen Mount-Anforderung gibt der Server dem Klienten ein Filehandle zurück, das der Klient für weitere Zugriffe innerhalb dieses montierten Dateisystems benutzt. Das Filehandle besteht aus einem Filesystem-Identifikator (major- und minor-number) und einer inode-Nummer, die angibt, welches Directory exakt montiert worden ist. Dazu wird noch eine Generationsnummer gefügt. Der Klient soll dieses Filehandle aber nicht manipulieren. Neben der *mount*-Prozedur enthält das Mount-Protokoll noch andere Prozeduren, wie z.B. *unmount* und die Anzeige der Exportliste.

Das NFS-Protokoll Das NFS-Protokoll besteht aus einer Menge von RPC-Aufrufen für den entfernten Dateizugriff. Diese Prozeduren unterstützen die folgenden Operationen:

- einen Block von Directoryeinträgen lesen
- nach einer Datei in einem Directory suchen
- links und Directories manipulieren
- das Lesen von Dateiattributen
- Lesen und Schreiben von Dateien

Für den Aufruf dieser Funktionen wird das Filehandle benötigt, das für das entfernt montierte Directory übergeben worden ist. Auffällig ist das Fehlen von *open* und *close* Operationen. Eine bekannte Eigenschaft der NFS-Server ist ihre Statuslosigkeit. Sie halten zwischen zwei Zugriffen keine Informationen über ihre Klienten. Daher muß jeder Zugriff eine vollständige Liste von Argumenten enthalten, eingeschlossen einen File-identifikator und einen absoluten Offset innerhalb der Datei für die zugehörige Operation. Daher geht auch keine Information verloren, wenn ein Server abstürzt und neu hochgefahren wird. Für den Klienten sieht es genauso aus, als ob der Server nur sehr langsam geantwortet hätte.

Der Statuslosigkeit des Servers entsprechend wird auch kein verbindungsorientiertes Protokoll (virtual circuit) benutzt, sondern das *User Datagram Protokoll* (UDP), das den Overhead von Verbindungsauf- und Abbau vermeidet. Da UDP aber nicht zuverlässig ist, muß das Wiederholen verlorengegangener Pakete von NFS selbst abgewickelt werden.

NFS-Architektur Die NFS-Architektur besteht aus drei Schichten. Die oberste ist das UNIX-Filesystem-Interface. Es behandelt Systemaufrufe und bildet jeden Pfadnamen in eine generalisierte inode, genannt vnode (virtual inode), ab. Auch Filedeskriptoren werden in vnodes abgebildet. Die zweite und wichtigste Schicht ist das *virtual file system* (VFS), in das die vnode den Eintrittspunkt darstellt. Sie hat zwei Aufgaben: Sie trennt generelle Dateisystemoperationen von ihrer Implementation. Daher können verschiedene Implementationen für das VFS Interface auf der gleichen Maschine gemeinsam existieren und so Zugriff zu unterschiedlichen Dateisystemtypen erlauben, die lokal montiert sind. Durch die Informationen in den vnodes trennt das VFS außerdem lokale von entfernten Zugriffen. Die vnode enthält einen netzwerkweit einheitlichen Filedeskriptor.

Der Kern verwaltet eine vnode für jedes aktivierte Dateiobjekt. Ähnlich wie im Standard-Unix benutzt der Kern eine Mount-Tabelle und markiert die vnode für jedes Directory, das ein Montierpunkt ist, so daß Pfade, die über ein solches Directory laufen, mit Hilfe der Mount-Tabelle zu den richtigen montierten Dateisystemen umgelenkt werden. Die vnode zeigt also entweder zu einer internen inode oder sie enthält ein Filehandle für ein montiertes Dateisystem. Wird eine entfernte Datei angesprochen, so ruft VFS die untere Schicht auf, die das NFS-Client-Protokoll implementiert. Ein RPC-Aufruf wird an die Server-Schicht auf dem entfernten System durchgeführt. Dort wird er an die VFS-Schicht hochgegeben, die feststellt, daß es sich um einen lokalen Zugriff handelt und so die entsprechende Dateisystemoperation ausführt. Das Ergebnis läuft auf dem umgekehrten Weg zurück (Bild 5). Ein *open*-Aufruf des Klienten führt dazu, daß der Server ein Filehandle für diese Datei zurückgibt und die VFS-Schicht des Klienten eine neue vnode erzeugt, die auf dieses Filehandle verweist.

Client- und Server-NFS-Code sind also im Kern enthalten. Der nfs-Daemon (nfsd) dient lediglich dazu, einen UDP-Port zu öffnen und dann den Systemaufruf *nfs_svc* durchzuführen; er läuft nicht weiter im Benutzermodus.

Pfadnamen-Bearbeitung Wie in Standard-Unix wird ein Pfadname Komponente für Komponente abgearbeitet. Sobald ein Montierpunkt zu einem entfernten Dateisystem überschritten worden ist, führt jede Komponentenbearbeitung zu einem separaten Remote-Procedure-Call zum Server. Hierbei wird die *lookup*-Routine gerufen, die Einträge in einem Directory sucht. Dieses aufwendige Pfadnamenübersetzungsschema ist notwendig, da der Server nicht weiß, welche weiteren Montierpunkte es im Klienten gibt. Es wäre effizienter, wenn man dem Server den gesamten restlichen Pfad übergeben könnte, sobald ein entfernter Montierpunkt überschritten ist, und dann nur eine vnode für den gesamten Pfad zurückbekommen würde. Aber jede Komponente in diesem restlichen Pfad kann auf dem Klienten ein neuer Montierpunkt für ein anderes Dateisystem sein. Um dieses *lookup* zu beschleunigen, hält der Klient einen Cache, der die vnodes für entfernte Directorynamen enthält.

Wenn vom Klienten eine Pfadkomponente auf dem Server geprüft wird, wird dessen Mount-Tabelle nicht berücksichtigt. Mehrere Dateisysteme des Servers müssen also mit genausovielen Mount-Anweisungen an den Klienten gebunden werden. Ebenso hat der Klient nicht teil an Datei-Systemen, die der Server seinerseits von anderen Servern montiert hat. Dies wäre von der Effizienz auch höchst unbefriedigend. Der Klient muß diese anderen Server selbst ansprechen. Diese Nichttransitivität kann dazu führen,

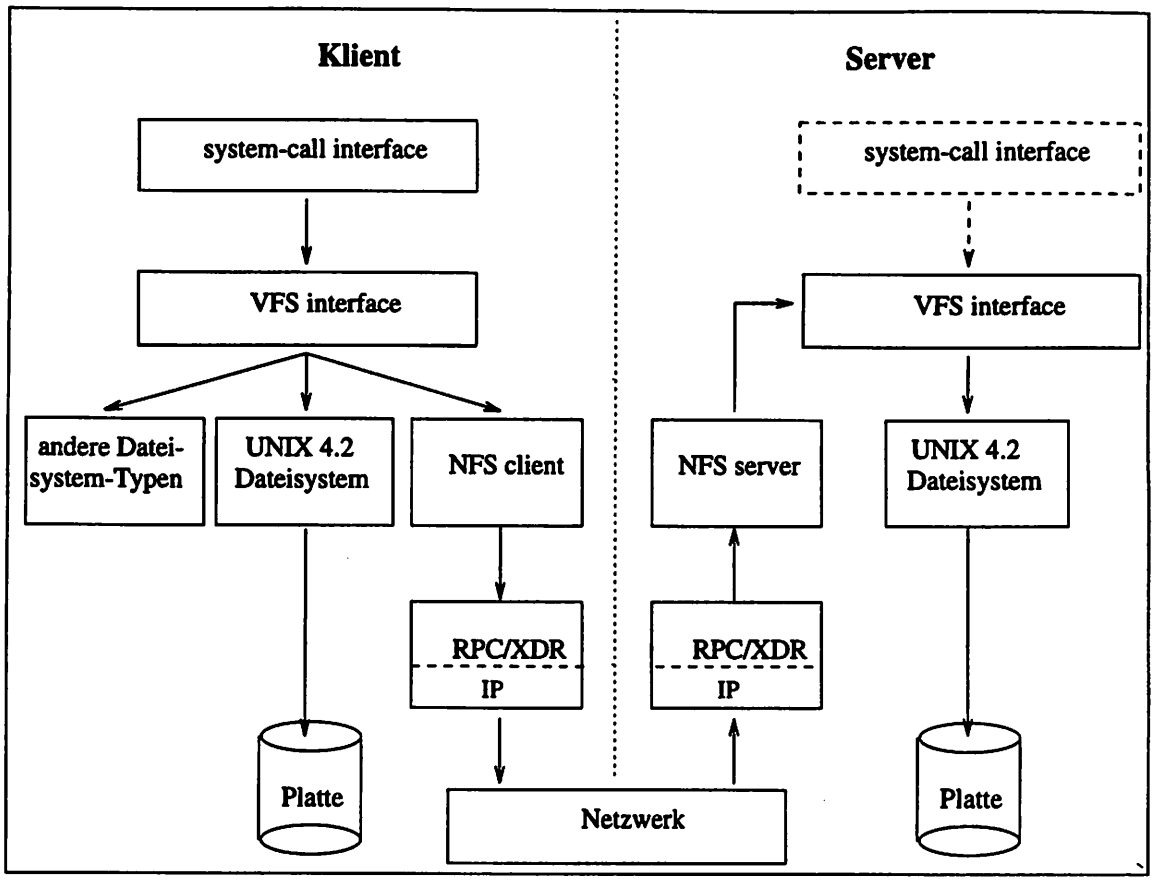

Abbildung 5: NFS-Architektur

daß ein Klient Dateien zur Verfügung hat, die auf einem Server stehen und dort nicht sichtbar sind. Der Klient sieht also den darunterliegenden Directory-Baum.

Caching Da der Server statuslos ist, die Semantik des RPC aber synchron, verlangt NFS, daß modifizierte Daten zur Platte des Servers geschrieben worden sind, bevor die RPC-Ergebnisse dem Klienten zurückgegeben werden. Um dadurch keine großen Performance-Nachteile zu erleiden, wird normalerweise auf dem Klienten ein Block-Caching aktiviert (durch das Programm biod, das den Aufruf *async_daemon* benutzt). Außerdem werden Dateiattribute (vnode-Informationen) im Cache gehalten. Wird eine Datei geöffnet, holt der Klient-Kern diese Attribute von neuem und vergleicht sie mit den Attributen im Cache. Die Datei-Blöcke im Cache werden nur benutzt, wenn die Attribute übereinstimmen. Solche Attribute werden bei Dateien nach drei Sekunden und bei Directories nach 30 Sekunden verworfen.

Zwischen Server und Klient werden sowohl read-ahead- als auch delayed-write-Techniken realisiert. Delayed-write-Blöcke werden vom Klienten erst als sauber markiert, wenn der Server bestätigt, daß die Daten zur Platte geschrieben worden sind. Durch die Pufferung kann es passieren, daß verschiedene Klienten verschiedene Inhalte der gleichen Datei sehen. Es ist nicht vorherzusagen, ob Schreibzugriffe auf eine Datei für andere Klienten sichtbar sind, in denen diese Datei bereits geöffnet worden ist. Die Daten im Cache des Klienten werden alle 30 Sekunden zum Server zurückgeschrieben. Beim Schließen einer Datei werden deren geänderte Blöcke sofort zurückgegeben. Neue *open*-Aufrufe bringen nur die Änderungen zum Vorschein, die bereits zum Server übertragen

worden sind. Die Caches sind also nicht kohärent. Außerdem sind unter NFS Schreibzugriffe nur atomar, wenn sie nicht die bei NFS eingestellte Block-Größe (zumeist 8 KByte) überschreiten. Bei größeren Schreibzugriffen von verschiedenen Klienten zum gleichen entfernten File kann es zu einer Vermischung der Daten kommen. Ein weiteres Problem ist, daß Dateilocking inhärent statusbehaftet ist und so ein Lock-Service nur außerhalb des NFS-Protokolls vorgesehen werden kann.

Diese Aufzählung zeigt, daß die Unix-Semantik von NFS nicht exakt eingehalten wird. Ein weiteres Problem tritt auf, wenn auf eine geöffnete Datei ein *unlink*-Aufruf ausgeführt wird. Unter Unix bleibt diese Datei für alle Prozesse, die sie geöffnet haben, bis zum Schließen les- und schreibbar. Der Server hat jedoch keine Information, aus der er entnehmen könnte, ob eine Datei noch von Prozessen offengehalten wird. Gelöst wird dieser Konflikt dadurch, daß der Klient einen *unlink*-Call auf eine geöffnete Datei in einen *rename*-Aufruf umsetzt und das *unlink* erst durchführt, wenn die Datei von allen Prozessen geschlossen worden ist (der temporäre Name für die Datei ist dann .nfsxxxxx). Dies klappt aber nur, wenn das *unlink* auf demselben Klienten ausgeführt wird, wo Prozesse diese Datei offenhalten. Beim Absturz des Klienten bleiben dann solche .nfsxxxxx Dateien auf dem Server übrig, der sie regelmäßig entfernt (crontab-Dienst).

Die Semantik der Unix-Dateioperationen ist im Prinzip statusbehaftet. Es muß ein *open*-Aufruf durchgeführt werden und unterhalb des Benutzerprogramms ist eine Instanz, die beim Lesen einen Zeiger mitführt. Es fragt sich daher, ob es überhaupt vernünftig ist, ein statusloses Protokoll für den Dateizugriff zu entwerfen. Die genannten Spezialprobleme zeigen die Schwierigkeiten auf, die bei der Implementation von NFS entstehen. Andererseits gibt es viele Anwendungsgebiete, in denen die Unempfindlichkeit gegenüber Serverzusammenbrüchen von großem Vorteil ist. Das Konzept von NFS ist also durchaus zufriedenstellend, läßt aber noch genügend Raum für Verbesserungen.

2.1.3 Prozeßmanagement

Auf einem verteilten Betriebssystem sollte der Benutzer nicht mit dem Problem konfrontiert werden, auf welchem Prozessor sein Prozeß startet. Vielmehr sollte das ganze System wie ein (sehr leistungsfähiger) Monoprozessor aussehen. Die Verteilung der Prozesse wird also vom Betriebssystem übernommen. Dabei gibt es mehrere Ziele:

Lastausgleich: Die Prozessoren sollen in etwa gleich ausgelastet sein, um alle Prozesse möglichst schnell zu bearbeiten.

Effizienter Dateizugriff: Verarbeitet ein Prozeß große Datenmengen, so sollte er bevorzugt auf den Rechner gebracht werden, auf dem die Daten liegen, damit sie nicht über das Netzwerk übertragen werden müssen.

Kommunikation: Prozesse, die intensiv miteinander kommunizieren, sollten auf dem gleichen Prozessor laufen, damit die Kommunikation effizient wird.

Es ist klar, daß einige dieser Ziele unvereinbar sind. Nehmen wir an, auf einem völlig untätigen System wird eine Pipe mit n Prozessen erzeugt. Die Zahl der Prozessoren sei größer n. Für den Lastausgleich ist es günstiger, diese n Prozesse zu verteilen; für die

Kommunikation ist es besser, die Prozesse zu konzentrieren. Da aber durch die Konzentration der Prozesse die durch die Pipe ermöglichte Parallelverarbeitung verhindert wird, muß das Ziel der Konzentration als nachrangig angesehen und erst berücksichtigt werden, wenn jeder Prozessor mit mehreren lauffähigen Prozessen belastet ist.

Bei Prozeßverteilungsalgorithmen muß berücksichtigt werden, ob das verteilte System aus Prozessoren gleicher oder verschiedener Architektur besteht, ob es also homogen oder heterogen ist. In einem heterogenen System wird die Transparenz natürlich eingeschränkt, da schon bei jedem Compileraufruf entschieden werden muß, für welche Architektur das Objektfile sein soll. Eine logische Partitionierung in Cluster gleicher Architektur ist dann sinnvoll. Leistet man sich den Luxus, bei jedem Compileraufruf mehrere Programme für die verschiedenen Prozessoren zu erzeugen, so kann man zumindest beim Programmstart den Prozessor wechseln. Der Mehrverbrauch an Plattenplatz ist aber erheblich. Die weiteren Betrachtungen sollen daher nur für homogene Systeme gelten.

Lastverteilung Lastausgleich kann dynamisch oder statisch erfolgen. Statischer Lastausgleich bedeutet, daß jeder Prozeß beim Starten auf einem beliebigen Prozessor gestartet werden kann, jedoch dann an diesen Prozessor gebunden ist. Diese Vorgehensweise ist vor allem dann erfolgreich, wenn das Verhalten von Prozessen von vorneherein bekannt ist. Diese Bedingung ist jedoch nur selten erfüllt; z.B. bei Produktionsautomatisierungen. Im Normalfall, speziell im Softwareengineeringbereich kann man keine Vorhersagen über das zukünftige Verhalten von Prozessen treffen. Für den statischen Lastausgleich gibt es verschiedene Algorithmen. Möchte man z.B. die Kommunikation, die zwischen verschiedenen Rechnern notwendig wird, minimieren und hat man weniger Prozessoren als Prozesse, so kann man die Prozesse mitsamt ihren Kommunikationslinien als Graph aufzeichnen, wobei die Knoten die Prozesse und die Kanten die Kommunikation, beschriftet mit der Intensität der Kommunikation, darstellen. Nun partitioniert man diesen Graph in Teile, wobei man an jedem Schnitt die Summe der Beschriftungen der Kanten bildet. Diese Summen sollen dann minimal ausfallen.

Interessanter ist der dynamische Lastausgleich, auch Prozeßmigration genannt. Dabei kann ein Prozeß zu jedem beliebigen Zeitpunkt von einem Rechner auf den anderen gebracht werden. Dabei treten erheblich mehr Probleme auf. Wird ein Prozeß von A nach B verlagert, so wird dazu sowohl von A als auch von B Rechenzeit gebraucht. Niemand kann aber garantieren, daß der Prozeß vielleicht bald zu Ende gegangen wäre, und zwar in einer Zeit, die geringer ist als allein die von A zur Verlagerung benötigte Rechenzeit. Zum zweiten benötigt man systemweit einheitliche Prozeßidentifikatoren. Um eine Verlagerung zu ermöglichen, müssen diese Identifikatoren transparent sein, das heißt, ihnen soll der Prozessor nicht angesehen werden können. Für diese transparente Benennung braucht man entweder einen zentralen Namensgeber oder eine verteilte Intelligenz, die solche einheitliche Namen vergeben kann. Dennoch bleibt bei einer Migration das Problem, daß Nachrichten zum neuen Prozessor umgeleitet werden müssen. Eine andere Möglichkeit, Prozesse anzusprechen, besteht darin, Nachrichten an eine feste Kommunikationsinstanz zu richten, deren Verbindung mit dem Prozeß dann dynamisch korrigiert werden kann (siehe Chorus). Als grober Schätzwert für die vom Prozeß noch benötigte Rechenzeit kann einfach die bisher verbrauchte Rechenzeit betrachtet werden.

Hat nun jeder Prozessor seine Belastung ermittelt, stellt sich die Frage, was zu tun ist. Eine Möglichkeit besteht darin, daß jeder Prozessor periodisch seine Belastung mit

einem Broadcast auf das Netz gibt. Ein stark ausgelasteter Prozessor kann dann einen Teil seiner Last an schwach ausgelastete Prozessoren abgeben. Abgesehen von dem Nachteil, daß nicht alle Netzwerke diese Broadcastmöglichkeit haben, braucht dieses Verfahren eine beträchtliche Netzbandbreite. Das größte Problem ist aber, daß ein momentan schwach ausgelasteter Prozessor schlagartig sehr viele Aufgaben übertragen bekommt und total überlastet wird. Im nächsten Entscheidungsschritt wird die Last dann wieder von ihm abgezogen, was zu einem Schwingen des gesamten Systems führt. Eine Verbesserung der Strategie besteht darin, Lastausgleich immer nur zwischen wenigen benachbarten Prozessoren durchzuführen. In einem Netz mit Bustopologie – wie z.B. Ethernet – kann zufällig bestimmt werden, welche Prozessoren als benachbart anzusehen sind. In diesem Fall diffundieren die Prozesse und damit die Belastung durch das Netzwerk wie eine Gaswolke. So einfach die Prozeßmigration in der Theorie aussieht, so schwierig ist sie doch in der Praxis. Das Problem liegt nicht darin, den Code, die Daten, den Stack und die Register zu transferieren, sondern darin, die Umgebung zu übertragen, wie z.B. die momentanen Positionen aller offenen Dateien, den Wert von laufenden Zeitgebern, Behandlung von Signalen oder Verbindungen zu E/A-Geräten. Diese Informationen sind alle im Kern verborgen. Unter Unix befindet sich ein großer Anteil dieser Daten zwar in der user-Struktur, andere sind aber über den ganzen Kern verteilt.

Unter Unix ist es auch nicht einfach, einen neu erzeugten Prozeß zu migrieren. Denn nach dem **fork**-Aufruf ist ja der neue Prozeß ein getreues Abbild des Vaterprozesses und damit genauso kompliziert wie bereits laufende Prozesse. Da *fork* häufig von einem *exec* gefolgt wird, ist es sinnvoll, nach einem *fork* den Sohnprozeß zunächst lokal weiterlaufen zu lassen und ihn erst nach einer kurzen Zeit als Kandidat für eine Migration zu betrachten.

Dies gilt umso mehr, wenn sogenannte copy-on-write Techniken angewandt werden. Dabei werden beim *fork*-Aufruf Text- und Datensegment nicht kopiert, sondern der Sohn arbeitet auf den Segmenten des Vaters. Diese werden für den Sohn in der Memory-Management-Unit allerdings als nur lesbar eingestellt. Zunächst läuft der Sohn weiter. Erst wenn er Schreibzugriffe auf diese copy-on-write Segmente durchführt, werden die entsprechenden Seiten kopiert. Durch diese Technik ist ein *fork*-Aufruf, der bald von einem *exec* gefolgt wird, so effizient geworden, daß es nicht sinnvoll erscheint, den Prozeß in diesem Moment zu migrieren.

Bei einem **exec**-Aufruf ist die Migration mit geringerem Aufwand zu erreichen, da die usermode-Bereiche nicht übertragen werden müssen. Wenn ein sehr großes Programm aufgerufen wird, ist es von Vorteil, den Prozeß dorthin zu übertragen, wo die Programmdatei liegt, vorausgesetzt dieser Prozessor ist nicht überdurchschnittlich ausgelastet. Verwaltet das Dateisystem Replikate von Dateien, so kommen auch alle Prozessoren in Frage, die ein Replikat lokal haben. Dadurch wird der Programmstart wesentlich beschleunigt. Wenn das Betriebssystem den Code nicht komplett in den Swap-Bereich lädt, sondern demand paging aus der Datei betreibt (z.B. SunOS 4.1), so werden die Verzögerungen bei späteren Seitenfehlern vermindert. Für stark interaktive Prozesse ist es nützlich, sie dorthin zu verlegen, wo es eine direkte Verbindung zum Terminal des Benutzers gibt, vorausgesetzt dieser Prozessor ist nicht überlastet. Da man beim Start einem Programm nicht ansieht, ob es sehr interaktiv ist, ist es nützlich, Informationen über die Interaktivität zu sammeln, um so später über eine Migration entscheiden zu können. Auch die Größe des Programms kann Entscheidungskriterien geben. Durch das

Programm ist zwar die Größe des Prozesses noch nicht bestimmt, dennoch ist bereits die Größe des Textsegmentes und die anfängliche Größe des Datensegmentes gegeben. Deuten diese Werte auf einen sehr großen Prozeß hin, so kann eine Migration zu einem Prozessor mit viel Arbeitsspeicher in Betracht gezogen werden.

Effizienter Dateizugriff Da Dateizugriffe nicht vorhersehbar sind, kann der Kern erst bei *open*-Aufrufen auf größere entfernte Dateien entscheiden, ob eine Migration auf den Knoten in Frage kommt, auf dem sich die Datei befindet. Der zugehörige Algorithmus sollte nicht nur sinnvolle Migrationen unterstützen, sondern auch verhindern, daß ein Prozeß, der nacheinander Dateien auf verschiedenen Knoten öffnet, ständig von Knoten zu Knoten weitergereicht wird. Eine Migration könnte z.B. erst dann in Betracht gezogen werden, wenn der Prozeß beginnt, die entfernte Datei sequentiell zu lesen. Dabei ist auch zu berücksichtigen, ob der Prozeß noch andere Dateien geöffnet hält, auf denen er vielleicht später noch Lesezugriffe durchführen wird. Eine interessante Möglichkeit bietet sich durch die Natur des kommandozeilenorientierten Dialogs. Bei den Argumenten der Kommandozeile handelt es sich zumeist um Namen von Dateien, die während des Programmlaufes geöffnet und gelesen werden. Die Namen sind entweder absolut oder relativ zum aktuellen Directory. Nur in seltenen Fällen beziehen sie sich auf irgendein konfiguriertes Directory, von deren Existenz nur das aufgerufene Programm weiß. Daher kann der *exec*-Service des Kerns so modifiziert werden, daß der Kern versucht, die in der Argumentliste von *exec* spezifizierten Dateien zu finden und daraus einen Prozessor bestimmt, der für das Lesen dieser Dateien am besten geeignet ist. Dieser Algorithmus ist nur sinvoll, wenn es sich bei den angegebenen Dateien um sehr große Dateien handelt. Ein Compiler z.B. sollte nicht dort gestartet werden, wo sich das möglicherweise kleine Quellprogramm befindet, sondern eher dort, wo die viel umfangreicheren Bibliotheken stehen. Ist der Compiler durch mehrere Pässe realisiert, die über Pipes kommunizieren, so können diese selbstverständlich auf verschiedenen Prozessoren laufen und zwar so, daß jeder Pass den optimalen Zugriff zu den für ihn nötigen Dateien hat. Die Syntaxanalyse kann also dort laufen, wo die Quellen sind und der Linker dort, wo die Bibliotheken sind. Durch die mögliche Parallelverarbeitung wird die Rechenzeit dann erheblich vermindert. Programme, die ohne Dateinamen als Argumente gestartet werden, sind entweder überhaupt nicht oder sehr stark interaktiv, wie z.B. Programme mit graphischen Benutzeroberflächen, die erst nach dem Start die Dateinamen vom Benutzer abfragen, auf denen sie arbeiten sollen.

Ein generelles Problem bei der Migration von Prozessen unter Unix ist die schon erwähnte Übertragung des Environments. Betrachten wir zwei Prozesse, die von einem gemeinsamen Ahnen eine offene Dateiverbindung geerbt haben und somit auf diese Datei einen gemeinsamen Schreib-Lesezeiger besitzen. Wird einer davon aus Lastausgleichsgründen migriert und liest in der Datei weiter, so müßte die dann erfolgte Änderung des Offsets in der kernel-open-file-table sofort auf den anderen Prozessor übertragen werden, damit der dort noch befindliche Prozeß richtig weiterlesen kann. Dies führt zwangsläufig zu Wettkampfbedingungen: Falls die beiden Prozesse zum gleichen Zeitpunkt weiterlesen, kann nicht verhindert werden, daß sie dieselben Informationen bekommen. Die Möglichkeit gemeinsamer Schreib-Lesezeiger für mehrere Prozesse wird von vielen Programmen benutzt und muß also erhalten werden. Dies ist zum Beispiel dadurch möglich, daß die Prozesse sich nicht an ihren lokalen Kern wenden, sondern an einen Dateiserver, der ihre Anforderungen serialisiert, und dadurch die alte Semantik

erhalten kann.

Ein weiteres Problem ist der für Migrationen wenig geeignete *fork*-Systemaufruf. Es sind natürlich auch andere Vorgehensweisen zur Erzeugung von neuen Prozessen denkbar. So wird zum Beispiel im V-Kernel und in Amoeba ein Prozeß erzeugt, indem der Kern zunächst beauftragt wird, Speicherbereiche für diesen Prozeß zu allokieren. Danach kann der Aufrufer diese Bereiche lesen und schreiben und er kann sie mit Code-, Daten- und Stacksegmenten für einen neuen Prozeß laden. Schließlich kann er die gefüllten Segmente zum Kern zurückgeben und ihn beauftragen, einen neuen Prozeß aus diesen Teilen zu erzeugen. Dieses Schema erlaubt das entfernte oder lokale Erzeugen von Prozessen in gleicher Vorgehensweise.

Ein anderes, für verteilte Betriebssysteme ebenfalls sinnvolles Konzept, ist das der leichtgewichtigen Prozesse (light weighted processes). Darauf wird im Kapitel über Mach näher eingegangen.

Prozeßmigration unter Condor Viele Universitäten ersetzen ihren zentralen Großrechner durch einen Pool von Workstations. Auf alten Zentralrechnern konnten Programme, die sehr hohe Rechenleistungen benötigten, im Hintergrund gestartet werden und die Ergebnisse standen nach vielen Stunden zur Verfügung. Mit einer Workstationstruktur haben alle Mitarbeiter einen persönlichen Rechner. Dieser wird aber auch durch die tägliche Arbeit belastet, so daß lang laufende Prozesse nicht genügend Rechenkapazität bekommen, während andere Workstations unbenutzt herumstehen. Die Mitarbeiter mit großen Rechenzeitanforderungen suchen daher nach solchen ungenutzten Workstations. Um diesen Vorgang zu automatisieren, wurde Condor an der Universität von Wisconsin entwickelt.

Auf jeder Workstation läuft ein spezieller user-mode-Scheduler, der sich nur um die von Condor verwalteten Hintergrundprozesse kümmert. Er stellt auch fest, ob die Workstation ungenutzt ist. Auf einer ausgewählten Maschine läuft ein zentraler Koordinator. Er sammelt die Lastinformationen der Workstations und verteilt die Aufgaben. Sobald ein Condor-Scheduler feststellt, daß die Workstation wieder vom eigenen Benutzer gebraucht wird, werden alle fremden Aktivitäten gestoppt. Die Besitzer von unausgelasteten Workstations haben also auf ihren eigenen Maschinen Priorität. Zum Wiederaufsetzen im Fehlerfall und für die Migration wird das Checkpointing benutzt. Darunter versteht man das Sichern eines kompletten Prozesses in regelmäßigen Zeitabständen.

Die Benutzer mit hohen Rechenanforderungen müssen ihre Programme nicht modifizieren. Sie werden lediglich mit einer speziellen Bibliothek gebunden. Shared libraries können deswegen nicht benutzt werden. Benutzer müssen sich keine Gedanken um Dateizugriffe machen. Die Daten werden mit Stub-Prozessen geholt. Condor übernimmt auch das Lokalisieren freier Workstations.

Wie auch die Newcastle-Connection ist Condor komplett außerhalb des Kerns in Bibliotheken realisiert. Daher sind auch einige Einschränkungen zu erklären: Die Programme dürfen keine neuen Prozesse erzeugen, d.h. *fork* und *exec* sind nicht erlaubt. Es gibt keine Signalübertragung, daher ist auch das Abfangen von Signalen sinnlos. Interprozeßkommunikation, z.B. über sockets, wird nicht unterstützt. Auf der auftraggebenden und der entfernten Maschine muß genügend Plattenplatz für das Checkpointfile sein. Dateien können gelesen oder geschrieben werden, aber Lese- und Schreibzugriffe auf dieselbe Datei werden nicht unterstützt.

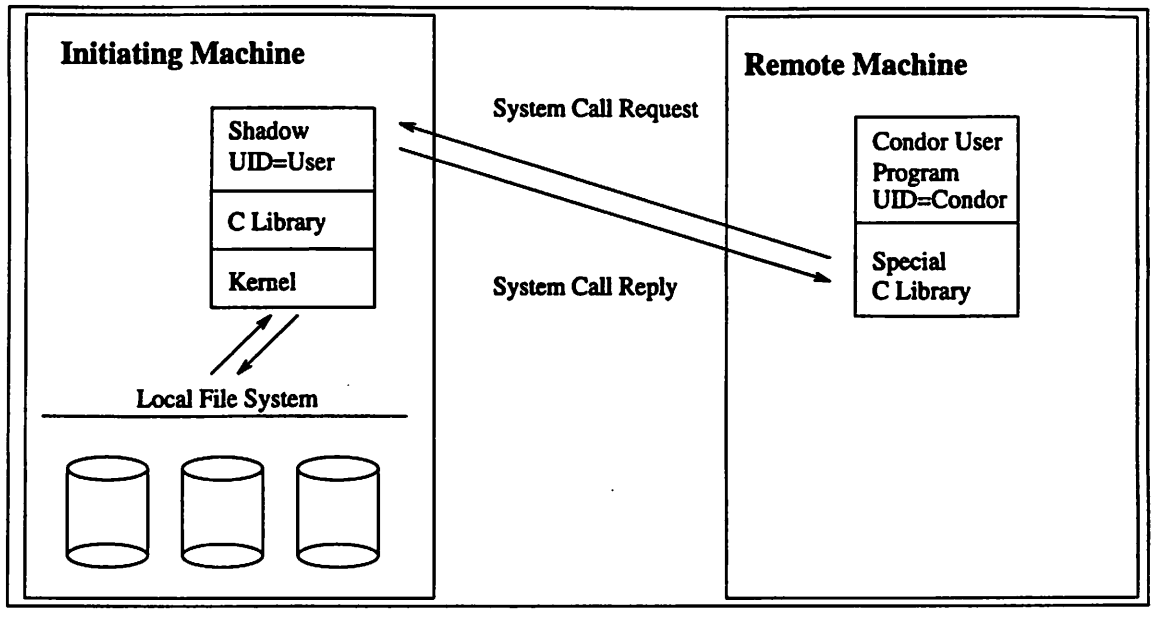

Abbildung 6: remote system call

Das Condor System benutzt das Konzept des Remote System Calls. Dadurch erhalten Prozesse die Illusion, auf der lokalen Maschine zu laufen. Immer wenn ein Condor Programm entfernt ausgeführt wird, läuft ein Stub, genannt Shadow auf der lokalen Maschine. Er führt die Systemaufrufe lokal durch und sendet die Resultate zum entfernten Stub zurück. Der Shadow läuft unter der *uid* des Benutzers, während der entfernte Stub unter einer speziellen *uid* von Condor läuft, die keine besonderen Rechte hat (Bild 6).

Checkpointing Während es vom Kern aus leicht ist, Text-, Daten- und Stacksegmente zu migrieren, gelingt dies ohne Kernmanipulationen nur mit besonderen Tricks. Offene Dateiverbindungen werden über die spezielle C-Bibliothek aufgezeichnet.

Zunächst ist jedes Condor Programm mit einer besonderen `crt0`-Routine gebunden. Dies ist der Code, mit dem jedes C-Programm startet und der *main* aufruft. Diese `crt0` setzt *ckpt* als Signal-handler. In *ckpt* wird nun die Position jeder offenen Datei abgefragt und in eine von *open* erzeugte Tabelle eingetragen, in der schon zu jedem Filedeskriptor der Dateiname steht. Dann wird ein *setjmp* aufgerufen, um die Register zu sichern. Schließlich schickt *ckpt* dem eigenen Prozeß ein Signal, um einen Speicherabzug (core dump) zu erzeugen. Dieses File wird mit dem ausführbaren Programmfile von Condor zu einem Checkpointfile kombiniert, das selbst wieder ausführbar ist.

Wenn es ausgeführt wird, startet es wieder mit der besonderen *crt0*-Routine, die nun *restart* als Signalhandler vermerkt und sich gleich ein Signal schickt. *Restart* kann nun, da es in einem eigenen Signal-Stack arbeitet, aus dem Checkpointfile den gesicherten Stackbereich selbst wieder restaurieren. Dann öffnet es alle vorher geöffneten Dateien und repositioniert sie entsprechend den Aufzeichnungen. Nun wird ein *longjmp* ausgeführt. Dadurch kommt das Programm aus dem *setjmp* in *ckpt* zurück und arbeitet auf dem alten Stack. *Ckpt* kehrt dann zu der Stelle zurück, an der das Programm zum Zeitpunkt des Checkpoints war. Aus der Benutzersicht sieht die ganze Aktion aus, wie wenn ein Signal abgefangen wurde und danach das Programm ganz normal vom Signalhandler zurückkehrt und weiterläuft.

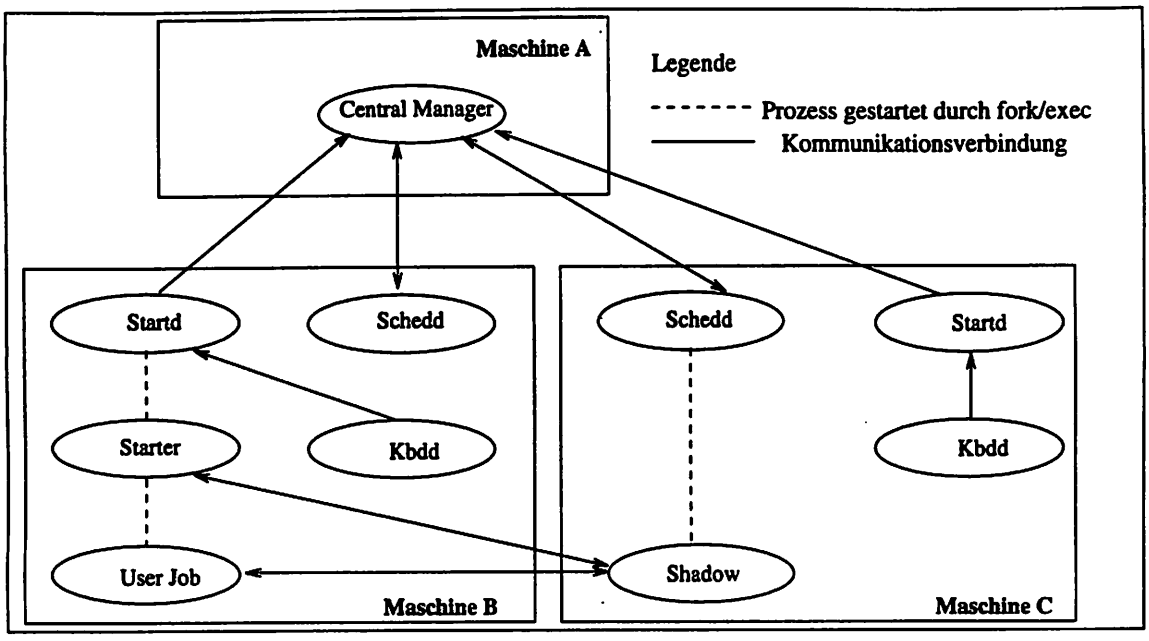

Abbildung 7: Condor Prozesse mit einem laufenden Job

Migrationsverwaltung Zur Kontrolle des Gesamtsystems laufen auf jeder Workstation, die Rechenleistung zur Verfügung stellen will, zwei Daemons, schedd und startd. Auf einer Workstation die alles kontrolliert, laufen zwei Prozesse, die zusammen den Central-Manager bilden. Sie werden hier nur gemeinsam als Central-Manager behandelt. Schedd verwaltet alle lokal erzeugten Condor-Jobs und meldet diese dem Central-Manager. Jeder startd stellt die Auslastung der Workstation fest und meldet diese ebenso. Wenn der Central-Manager entscheidet, einen Job der Workstation A auf B laufen zu lassen, so gibt er an den schedd von A die Erlaubnis. Dieser wählt einen Job aus seiner Warteschlange aus, erzeugt einen Shadow-Prozeß für ihn und nimmt Kontakt mit dem startd auf B auf. Falls sich die Auslastung von B nicht wieder erhöht hat, antwortet ihr startd mit OK und erzeugt einen Starter-Prozeß. Dieser bekommt das Checkpointfile vom Shadow auf A übermittelt. Der Starter erzeugt einen neuen Prozeß, der wie beschrieben das Checkpointfile ausführt (Bild 7). In festen Zeitabständen wird nun ein neues Checkpointfile vom Condor-Job erzeugt.

Wenn der Benutzer von Workstation B zurückkehrt, erkennt das der startd entweder an einem Ansteigen der Prozessorlast oder auch an Zugriffen auf (virtuelle) Terminals. Er stoppt den Job zunächst mit einem Suspend-Signal, denn oft sind Benutzer nur für wenige Sekunden aktiv (z.B. um zu sehen, ob neue Mail gekommen ist). Wenn die Workstation länger als 5 Minuten aktiv bleibt, veranlaßt startd den Starter, den Job abzubrechen. Beim nächsten Start wird also auf das letzte Checkpoint-File zugegriffen. Checkpointfiles werden in regelmäßigen Zeitintervallen erzeugt, damit diese E/A-aufwendige Operation nicht erst dann durchgeführt werden muß, wenn der Benutzer zurückgekehrt ist.

Startd beachtet eine Konfigurationsdatei, in der der Workstationbenutzer definieren kann, wann seine Maschine als unbenutzt angesehen wird und Condor-Prozesse zugelassen werden. Er kann z.B. verlangen, daß der *load average* unter 0,3 und der letzte Terminalzugriff 15 Minuten zurückliegt. Schedd kann Prioritäten berücksichtigen, die der Benutzer bei der Übergabe eines Jobs an Condor angibt. Bei gleicher Priorität

werden die Prozesse bevorzugt, die schon ein Checkpointfile haben, da sie mehr Plattenplatz beanspruchen. Unter diesen werden ältere Jobs bevorzugt abgearbeitet. Der Central-manager berücksichtigt keine Prioritäten einzelner Jobs, sondern ordnet den einzelnen Workstations verschiedene Prioritäten zu. Diese erhöhen sich, wenn Jobs mehrerer Benutzer warten und vermindern sich desto mehr, je mehr Jobs schon laufen.

Das Condor-System ist ein erster Ansatz zur Prozeßmigration. Es wurde festgestellt, daß Prozeßmigration nicht notwendigerweise eine schnellere Prozeßbeendigung ermöglicht. Aufgaben, die länger als eine Nacht benötigen, werden am Tage wieder suspendiert, wenn keine Workstation frei ist. Verglichen mit einer Ausführung im Hintergrund mit geringer Priorität, benötigen solche Aufgaben länger. Abhilfe kann hier nur durch Hinzunahme von Workstations erfolgen, die nicht für persönliche Benutzung gedacht sind.

3 Beispiele verteilter Unix Betriebssysteme

3.1 Locus

Locus wurde Anfang der 80er Jahre an der University of California at Los Angeles (UCLA) als volles verteiltes Betriebssystem entwickelt. Das System ist kompatibel mit UNIX, aber die Änderungen sind groß und erforderten einen vollkommen neuen Kern. Locus wurde auf VAX/750, die über ein Ethernet verbunden waren, betrieben.

3.1.1 Überblick

Locus erreicht einen hohen Grad von Netzwerktransparenz. Während des normalen Betriebes sind dem Benutzer die Maschinengrenzen vollständig verborgen. Das Dateisystem präsentiert sich als ein einziger Baum. Dieser enthält alle Objekte (Dateien, Directories, ausführbare Dateien und Geräte) von allen Maschinen im System. Die Namen sind ortstransparent; es ist nicht möglich, vom Namen auf den Ort eines Objektes im Netzwerk zu schließen. Zu einer Locus Datei kann es verschiedene Kopien auf verschiedenen Rechnern geben. Auch diese Replikation ist transparent, da das System sich darum kümmert, alle Kopien auf dem neuesten Stand zu halten und Zugriffe von der neuesten Version zu bedienen. Für die Programme ist die Replikation vollkommen unsichtbar. Sie wird hauptsächlich benutzt, um Dateien zum Lesen zu Verfügung zu haben, auch wenn Rechnerausfälle und Netzwerkunterbrechungen auftreten. Prozesse können lokal und entfernt in der gleichen Weise erzeugt werden und die Prozeßinteraktion ist unabhängig vom Ort.

Das Dateisystem spielt eine zentrale Rolle in der Systemstruktur. Locus bietet die gleiche Dateizugriffssemantik, die Standard-Unix hat. Zusätzlich gibt es durch commit- und abort-Systemaufrufe die Möglichkeit, Transaktionen durchzuführen. Ein wesentlicher Forschungsteil wurde auch darin gesehen, das System am Laufen zu halten, wenn ein Teil der Rechner ausfällt oder aber das Netzwerk in separate aber funktionierende Unternetzwerke aufgeteilt wird. Solange eine Kopie einer Datei verfügbar ist, können Lesezugriffe erfüllt werden und es ist immer garantiert, daß die gelesene Version die neueste verfügbare ist. Veraltete Versionen von Dateien werden baldmöglichst und

automatisch entfernt. Für die Bedienung entfernter Anforderungen wurden spezielle leichtgewichtige Prozesse, genannt Serverprozesse, geschaffen. Diese Prozesse bekommen keinen geschützten Adreßraum, sondern laufen mit privilegiertem Speicherzugriff. Ihr gesamter Code und ihre Stacks sind im Betriebssystemkern. Sie können alle internen Systemroutinen direkt aufrufen und Datenstrukturen des Kerns lesen. Diese Prozesse bedienen Netzwerkanforderungen, die sich in einer system-queue befinden.

3.1.2 Dateisystem

Wie im Standard-UNIX wird der Dateibaum aus einer Sammlung von Filegroups aufgebaut. Eine Filegroup ist ein Filesystem, also ein sich vollständig selbst enthaltender Unterbaum, einschließlich Speicherplatz für alle Dateien und Directories in diesem Unterbaum. Diese Filegroups werden mit dem Mount-Mechanismus zusammengebaut. Der Kern enthält eine Mount-Tabelle, die auf allen Prozessoren repliziert ist. In Locus wurde besonders darauf geachtet, daß der Zugriff auf Dateiobjekte schnell ist. Sind diese lokal, ist der Zugriff nicht langsamer als auf einem konventionellen UNIX-System.

Die Replikation von Dateien im verteilten Dateisystem von Locus dient mehreren Zwecken. Mehrere Kopien erhöhen zum einen die Verfügbarkeit, zum anderen die Leistungsfähigkeit. Wenn Benutzer auf mehreren Maschinen eine Datei lesen, können diese Zugriffe deutlich schneller ausgeführt werden, wenn Replikate auf diesen Maschinen existieren. Einige Dateien sind so wichtig, daß ihre Replikation zwingend erforderlich ist, denn sie müssen auch verfügbar sein, wenn mehrere Maschinen ausgefallen sind. Die Start-up-Files oder die verschiedenen Shells, ebenso mail aliases und routing Information sind Beispiele hierfür.

Replikation ist auch wichtig für Directories. Ein globaler Namensraum für Dateien hat zur Folge, daß viele Directories Einträge haben, die zu Dateien auf anderen Maschinen zeigen. Es ist sehr nützlich, alle Directories im Pfad einer Datei dort (repliziert) zu haben, wo auch die Datei gespeichert ist, denn wenn ein Eintrag im Pfad dieser Datei nicht zugreifbar ist, weil ein Rechner ausgefallen ist, so ist das ganze File nicht verfügbar, obwohl es lokal gespeichert sein kann. Die Zugriffsweise auf Directories hängt davon ab, an welcher Stelle sich das Directory befindet. Ist es im Baum weit oben, wird es sehr häufig gelesen und wenig geändert. Eine starke Replikation ist daher nötig und problemlos. Im Gegensatz dazu werden Directories, die sich weiter unten im Baum befinden, weniger oft gelesen und häufiger geändert, da sie nur für wenige Benutzer relevant sind. Es ist also sinnvoll, ihre Replikation geringer zu halten, um den Aufwand der Änderungen zu beschränken.

Die Replikation von Dateien wird erreicht, indem man einer Filegroup mehrere physikalische Container zuweist. Diese Container befinden sich auf verschiedenen Knoten und können Kopien der Dateien dieser Filegroup enthalten. Filesystemintern wird eine Datei durch das Paar (Filegroupnummer, inodenummer) bezeichnet. Diese Bezeichnung verbirgt sowohl den Ort als auch die Replikation. Jeder Knoten, der die Kopie eines Directories hat, über das eine Unterbaum montiert ist, muß die inode dieses Directories im Speicher haben und sie als übermontiert markiert haben. So können Zugriffe auf jeden Knoten zu diesem Directory die Mount-Tabelle berücksichtigen.

Auf der physikalischen Ebene entsprechen Container den Plattenpartitionen. Ein Container wird bezeichnet durch eine Filegroupnummer und eine Containernummer. Einer

dieser Container pro Filegroup wird als das Original (primary copy) betrachtet. Eine Datei muß auf dem Rechner mit der primary copy gespeichert sein und kann zusätzlich auf jedem Rechner liegen, wo es einen Container für ihre Filegroup gibt. Die primary copy speichert die Filegroup also vollständig, während der Rest der Container sie teilweise haben kann. Die verschiedenen Kopien einer Datei haben auf allen Containern die gleiche inode-Nummer. Ein Container hat also einen leeren inode-Eintrag für alle Dateien, die er nicht speichert. Die Datenblocknummern können auf verschiedenen Containern verschieden sein; daher benutzen Netzwerkzugriffe zu Datenblöcken nur logische Blocknummern. Wenn Dateien geändert werden, können nicht alle Kopien gleichzeitig auf den gleichen Stand gebracht werden. Um die Replikationsverwaltung zu vereinfachen, enthält jede inode noch eine Versionsnummer. Diese entscheidet bei allen Zugriffen, welche Version die aktuellste ist.

Jeder Rechner hat eine Container-Table, die für alle Filegroups, die lokale Container auf dem Rechner haben, die Partition dieser Container angibt. So kann festgestellt werden, ob es möglich ist, Dateizugriffe lokal zu erfüllen. So wichtig es ist, einen global einheitlichen Namensbaum zu haben, so ist dies mitunter bei den rechnerspezifischen Einträgen unter `/dev` problematisch. In Locus gibt es die Möglichkeit, Zugriffe zu diesen traditionellen Dateinamen zu rechnerspezifischen Files umzulenken. Die Transparenz des Dateisystems manifestiert sich in seinen Systemaufrufen. Zu den bekannten Aufrufen *open, create, read, write, close* und *unlink* wurden noch *commit* und *abort* hinzugefügt, um Transaktionen zu unterstützen. Der nächste Abschnitt behandelt die Vorgänge, die beim Dateizugriff auf verschiedenen Rechnern ablaufen.

Dateizugriffe Die Vorgehensweise bei Dateioperationen unterscheidet sich von dem sonst üblichen Client-Server Modell. Die Replikation erfordert eine zusätzliche Funktion. Locus unterscheidet drei Rollen im Dateizugriff. Jede kann von einem anderen Rechner ausgeführt werden:

Using Site (US): Die US führt den Dateizugriff aus und bekommt den Inhalt geliefert.

Storage Site (SS): Die SS hat eine Kopie der verlangten Datei und wurde ausgewählt, die Anforderung zu bedienen.

Current Synchronisation Site (CSS): Die CSS ist verantwortlich für die Synchronisation in einer Filegroup und wählt eine SS für jeden open-Aufruf auf eine Datei in dieser Filegroup. Es gibt genau eine CSS für jede Filegroup. Sie wartet die Versionsnummer und eine Liste von Containern für jedes File in der Filegroup.

Für einen *read*-Aufruf erhält die US nach Abarbeitung des Pfadnamens ein Zahlenpaar (Filegroup-Nummer, inode-Nummer). Sie stellt eine freie inode-Struktur zur Verfügung, schlägt in ihrer Mount-Tabelle nach, in der für jede Filegroup die CSS verzeichnet ist und sendet Filegroup-Nummer und inode-Nummer an die CSS. Diese weiß einerseits, welche Rechner Container für diese Filegroup besitzen, andererseits hat sie für jedes File eine aktuelle Versionsnummer. Die CSS fragt nun verschiedene SS, ob sie als Server agieren wollen und übergibt dabei ihre Versionsnummer, so daß die angefragten SS entscheiden können, ob ihre Kopie aktuell ist oder nicht. Ist sie nicht

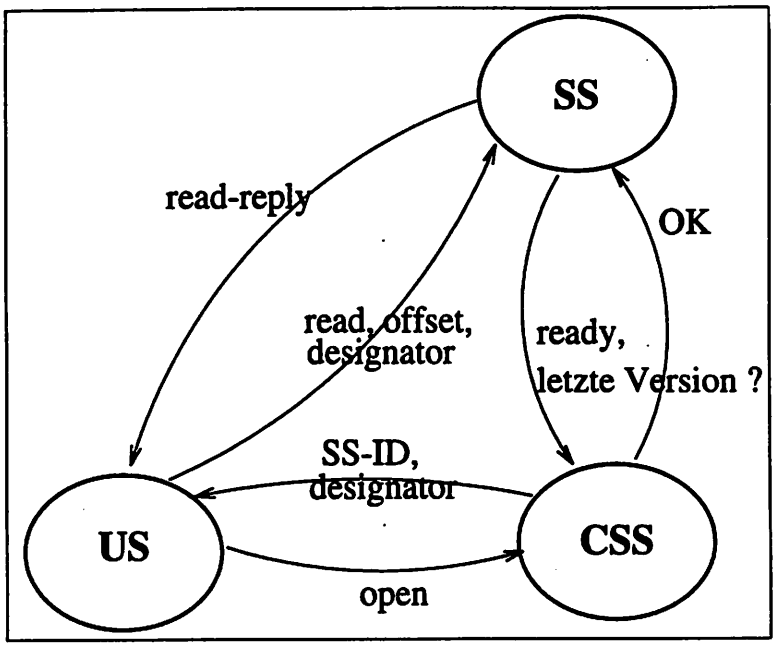

Abbildung 8: Locus open Protokoll

aktuell, verweigert die SS den Dienst. Die CSS wählt nun unter den SS, die geantwortet haben eine geeignete aus und schickt ihre Identität zur US zurück. Sowohl CSS als auch SS allozieren incore-inode-Strukturen für das geöffnete File. Die CSS braucht diese Information für weitere Synchronisationsentscheidungen und die SS benötigt die inode, um Lese- und Schreibzugriffe effizient bedienen zu können. Wenn die US selbst eine aktuelle Kopie des Files zur Verfügung hat, wird sie von der CSS als SS ausgewählt. Die Antwort von der CSS enthält die nötigen Informationen um die inode-Struktur zu vervollständigen, zum Beispiel Dateigröße, Eigentümer, Zugriffsrechte (Bild 8).

Nachdem eine Datei geöffnet worden ist, werden Lesezugriffe direkt an die SS gesandt ohne die CSS zwischenzuschalten. Ein Lesezugriff besteht aus dem Paar (File-Group-Nummer, inode-Nummer) und der logischen Blocknummer innerhalb der Datei. Hat die SS die zugehörige inode gefunden, übersetzt sie die logische Blocknummer in eine physikalische Blocknummer, stellt einen Puffer im Cache zur Verfügung und liest dorthin den Plattenblock ein. Der Block wird zur US übertragen und dort in einem weiteren Kernel-Puffer gespeichert. So können weitere Lesezugriffe möglicherweise von diesem Puffer bedient werden. Wie auch beim lokalen Lesen wird read-ahead benutzt, und zwar sowohl auf der SS als auch von der US aus. Falls ein Rechner die Verbindung mit der SS verliert, während eine Datei gelesen wird, versucht Locus eine andere Kopie der gleichen Version dieser Datei zur Verfügung zu stellen.

Beim letzten close auf der US, sendet diese eine close-Nachricht an sie SS, welche sie an die CSS weitergibt. Die CSS bestätigt diese Meldung der SS und diese gibt die Bestätigung weiter an die US. SS und CSS können ihre incore-inode-Strukturen deallozieren. Das hier beschriebene Protokoll ist die unterste Ebene im System, mit Ausnahme einiger Mechanismen zur Übertragungswiederholung. Da es nicht die vielen Schichten gibt, die vom ISO-Standard vorgeschlagen werden, wird eine wesentlich höhere Leistung erreicht.

Pfadnamenbearbeitung Im vorausgegangenen Abschnitt wurde gezeigt, wie eine Datei geöffnet wird, wenn das Paar (Filegroup-Nummer, inode-Nummer) gegeben ist. Dieses Paar wird aber erst bei der Bearbeitung eines Pfadnamens erhalten. Die Pfadnamen sind reguläre Unix-Pfadnamen ohne besondere Zeichen oder spezielle Muster für entfernten Zugriff wie in der Newcastle Connection. Wie gewohnt starten alle Pfadnamen entweder vom root-Directory oder vom current-working-directory des Prozesses. In beiden Fällen ist eine inode im Arbeitsspeicher für dieses Directory. Daher kann das Directory der ersten Pfadkomponente gelesen werden. Es gibt also keine Parallele zur Lookup-Operation bei NFS; das Durchsuchen des Directory wird vom Klienten und nicht vom Server überwacht. Für das Pfadnamenabarbeiten wird ein Directory nicht normal geöffnet, sondern in einer speziellen, unsynchronisierten internen Weise. Dabei wird kein gegenseitiger Ausschluß (Locking) durchgeführt. Wenn das Directory lokal gespeichert ist, wird die CSS nicht informiert. Änderungen an einem Directory können ausgeführt werden, während diese Suche stattfindet. Da jeder dieser Änderungsvorgänge atomar ist, sieht die Suchoperation niemals einen inkonsistenten Zustand. Ist das Directory geöffnet, werden Seiten daraus geholt und nach der ersten Pfadnamenkomponente gesucht. Ist sie gefunden, wird die zugehörige inode-Nummer gelesen und das Directory geschlossen. Diese Strategie wird weitergeführt bis zur letzten Komponente, genau so wie auf einem herkömmlichen Unix-System.

Dateimodifikation Wird eine Datei zum Schreiben geöffnet, wählt die CSS denjenigen Rechner aus, der den Container mit der primary-copy (dem Original) besitzt. Wenn der Schreibauftrag nicht einen kompletten Block umfaßt, wird der alte Block von der SS gelesen. Wenn ein kompletter Block überschrieben wird, ist das Lesen unnötig und die US setzt einen Puffer auf, ohne zu lesen. Die Blöcke werden über eine delayed-write-Mechanismus zur SS zurückgeschrieben. Spätestens beim Schließen der Datei müssen alle modifizierten Blöcke zur SS gesandt werden, bevor die Close-Nachricht gesandt werden kann.

Aus der Welt der Transaktionssysteme wurde das Konzept der unteilbaren Folgen von Änderungen übernommen. Änderungen an einer Datei werden nicht permanent, bis der Prozeß einen commit-Aufruf absetzt oder die Datei schließt. Die Alternative ist der abort-Aufruf, der alle Änderungen zurück bis zum letzten commit-Aufruf ungültig macht. Um diese Art der Modifikation zu ermöglichen ist es notwendig, sowohl die Original- als auch die geänderten Daten verfügbar zu haben. Dazu gibt es zwei bekannte Mechanismen: Logging und Shadow-Blocks (Schattenblöcke). Locus benutzt Schattenblöcke, hauptsächlich weil Unix-Dateioperationen häufig ganze Dateien überschreiben und weil eine hohe Performance dadurch leichter zu implementieren ist. Da die US nur mit logischen Blocknummern arbeitet, ist der gesamte Mechanismus in der SS implementiert und transparent für die US. Wenn ein Block einer Datei geändert wird, alloziert die SS einen neuen Block. Dies wird ohne zusätzliche E/A-Operationen durchgeführt. Wenn der ganze Block geändert wird, wird der neue Block mit den neuen Daten gefüllt. Wenn nur ein Teil geändert wird, wird der alte Block gelesen, geändert und dann als Schattenblock betrachtet. Beide Änderungen lassen die Blöcke auf der Platte im alten Zustand. Die inode auf der Platte enthält unverändert die alten Blocknummern. Die inode im Arbeitsspeicher zeigt jedoch auf die Schattenblöcke, soweit solche existieren, ansonsten auf die alten Blöcke. Werden in Schattenblöcken Informationen verändert, werden keine neuen Blöcke bereitgestellt, sondern diese Änderungen

in den Schattenblöcken selbst durchgeführt.

Die unteilbare commit-Operation ersetzt nun die inode auf der Platte durch die inode im Arbeitsspeicher. Danach enthält die Datei permanent die neue Information. Um Änderungen zu stornieren (abort), werden die inode im Arbeitsspeicher und die Schattenblöcke freigegeben. Die CSS garantiert, daß nicht zwei verschiedene Kopien der gleichen Datei zur selben Zeit geändert werden können und verhindert, daß eine alte Kopie gelesen wird, während die Datei geändert wird. Als Bestandteil der commit-Operation sendet die SS Nachrichten zu allen anderen SS dieser Datei und zur CSS. Diese Nachrichten enthalten zumindest die Identifikation der Datei und die neue Versionsnummer. Zusätzlich kann die Nachricht einen Hinweis darauf enthalten, ob nur die inode-Information geändert worden ist, oder, falls Daten geändert worden sind, welche logischen Blöcke modifiziert wurden. Ab diesem Zeitpunkt liegt es in der Verantwortung dieser SS, ihre Kopie auf den neuesten Stand zu bringen, indem sie die geänderten Blöcke oder die ganze Datei anfordern. Eine Warteschlange von solchen Anforderungen wird vom Kernel jedes Prozessors verwaltet und ein Kernelprozeß bedient diese Warteschlange. Die Verbreitung der neuen Version wird also durch das aktive Herbeiholen und nicht durch das Versenden ermöglicht. Die SS, die die neue Version holt, benutzt dabei den bekannten Modifikationsmechanismus, so daß, wenn die Verbindung abbricht, die Datei konsistent erhalten bleibt. Jeder Rechner enthält also entweder das veraltete Original oder eine vollständig geänderte Version, aber niemals eine Version, die nur teilweise geändert ist.

Beim Erzeugen von neuen Dateien muß das System wissen, wieviele Kopien anzulegen sind. Durch das Hinzufügen dieser Informationen zum *creat*-Systemaufruf wird dieser Aufruf inkompatibel zu bisherigen Unix-Systemen werden. Daher werden Voreinstellungen verwendet, die zusammen mit dem Prozeßkontext vererbt werden, und neue Systemaufrufe, um sie zu modifizieren. An jeden Prozeß ist eine Variable gebunden, die die Anzahl der Kopien angibt. Wird nun ein neues File erzeugt, so wird das Minimum aus dieser Zahl und dem Multiplikationsfaktor des Parent-Directory genommen. Die Datei wird nur dort repliziert, wo auch die Parent-Directory repliziert ist, und der Rechner, auf dem der Erzeugerprozeß läuft, ist ein bevorzugter Kandidat. Die Information über die Erzeugung einer neuen Datei wird genauso an die anderen Rechner weitergegeben, wie die über die Veränderung einer Datei. Wenn eine Datei gelöscht werden soll, wird deren inode von der US markiert und ein commit-Befehl ausgeführt. Dadurch erhöht sich die Versionsnummer und es wird ausgeschlossen, daß Kopien dieser Datei auf anderen Rechnern noch zum Lesen verwendet werden. Haben alle SS vom Löschen der Datei Kenntnis genommen, kann die inode auf der SS, die den Löschauftrag bekommen hat, freigegeben und wieder verwendet werden.

3.1.3 Prozesse

Entfernte Prozeßerzeugung Locus erlaubt, Programme auf jedem Rechner im Netzwerk auszuführen. Es ist möglich, dynamisch vor dem jeweiligen Prozeßaufruf die Ausführungsstelle zu bestimmen. Der Mechanismus ist vollständig transparent, so daß existierende Software lokal oder entfernt benutzt werden kann, ohne modifiziert zu werden. Die Entscheidung, wo ein neuer Prozeß ausgeführt werden soll, wird ebenfalls mit einer an den Prozeß gebundenen Variable bestimmt. Wie in Unix werden Prozesse mit *fork* erzeugt und mit *exec* wird ein anderes Programm aufgerufen. Beide Aufrufe

werden kontrolliert durch die Ortsinformation in der *per-process*-Variablen. In beiden Fällen werden Prozeßsegmente auf der Zielseite alloziert und das Prozeßenvironment über Nachrichten übermittelt. Die Migration bei einem *fork*-Aufruf wird erleichtert, wenn das Programm auf dem entfernten Rechner bereits läuft und daher der Code nicht mehr erneut geladen werden muß. Zur Optimierung wurde ein run-Aufruf hinzugefügt, der eine Kombination von *fork* und *exec* darstellt. Dabei bleibt das *fork* lokal und erst das *exec* führt zur Migration. Dadurch werden unnötige Kopien vermieden.

Interprozeßkommunikation Eine der Schwierigkeiten eines transparenten Prozeßsystems ist das Erhalten der Semantik der verschiedenen Funktionen, mit denen Prozesse kommunizieren. In Unix gibt es explizite Funktionen, wie Signale und Pipes (benannt oder unbenannt), aber es gibt auch implizite Mechanismen. Gemeinsam geöffnete Dateien sind die wichtigsten. Die größte Schwierigkeit dieser Funktionen ist, daß sie shared memory erwarten. In Unix können Prozesse auf gemeinsam geöffnete Dateien Lese- und Schreibzugriffe ausführen und das System garantiert, daß jede nachfolgende Operation die Effekte aller vorausgegangenen sieht. Dies ist relativ leicht zu implementieren, wenn die Prozesse gemeinsame Betriebssystemdaten und -caches teilen. Wenn die Prozesse auf verschiedenen Maschinen laufen, muß ein wesentlich höherer Aufwand betrieben werden.

Prozesse, die einen gemeinsamen Ahnen haben, können in einer Datei den Schreib-Lesezeiger teilen. In Locus wird dafür ein Token-Mechanismus benutzt, der anzeigt, welche Kopie einer Ressource gültig ist. Der Zugriff zur Ressource, also zum Beispiel das Ändern des Schreib-Lesezeigers ist nur möglich, wenn das Token präsent ist. Unter ungünstigen Umständen kann allerdings die Leistungsfähigkeit durch die Geschwindigkeit, mit der das Token und die zugehörige Ressource weitergegeben werden kann, beschränkt werden. Der Tokenmechanismus wird an verschiedenen Stellen im System verwendet. Inodes und Datenpuffer können auf verschiedenen Prozessoren im Cache enthalten sein. Da es mehrere Leser, aber nur einen Schreiber geben darf, werden Data-Tokens und Write-Tokens benutzt. Nur ein Rechner mit dem Write-Token für eine Datei darf diese modifizieren. Wird das Write-Token abgefordert, so wird die inode und alle modifizierten Seiten zur SS zurückkopiert. Da beliebige Änderungen vorkommen können, während das Token nicht vorhanden ist, werden alle Puffer für ungültig erklärt, wenn das Token abgegeben wird. Nur wenn ein Data-Token anwesend ist, ist garantiert, daß die gepufferten Datenblöcke gültige Daten enthalten.

3.1.4 Betrieb bei Netzwerkfehlern

Netzwerkausfälle unterteilen ein verteiltes System in verschiedene Partitionen. Während normalerweise die Modifikation von Dateien keine Probleme bereitet, da immer mit der Primary copy gearbeitet wird, muß es bei Partitionen auch erlaubt sein, eine beliebige Kopie zu verändern. Wenn das Netzwerk wieder arbeitet, kann Locus für einige Datentypen die es kennt, eine automatische Zusammenführung veranlassen. In anderen Fällen wird das Problem an höhere Schichten gemeldet. Exemplarisch für das Zusammenführen von Dateien werden hier Directories behandelt. Eine Directory ist eine Datenstruktur, in der jeweils einem Pfadnamen, also einem String eine inode Nummer zugeordnet ist.

Auf diese Datenstruktur können zwei Operationen angewandt werden. Einfügen eines Pfadnamens und entfernen einen Pfadnamens. Wenn die Netzwerkaufteilung beendet ist, kann für jede Directory die Versionsnummer und die Liste der letzten Änderungen verglichen werden. Stimmen diese überein, so ist keine Aktion nötig. Ansonsten gibt es folgende Regel: Man geht durch beide Directories und prüft, ob für alle Namen die inode-Nummern gleich sind. Falls dies nicht der Fall ist, werden beide Namen geringfügig geändert und die Eigentümer der beiden Dateien per Mail benachrichtigt. Kommt ein Eintrag in einer Directory vor aber nicht in der anderen, so wird er auch in die andere eingefügt. Falls ein Eintrag gelöscht wurde (die inode dieser Datei wurde speziell markiert, solange bis alle SS vom Löschen Kenntnis genommen haben), so wird der Eintrag auch im anderen Directory gelöscht, wenn dort nicht eine Modifikation der Daten passiert ist.

Bis 1983 wurden ungefähr 50 Mannjahre investiert. Locus zeigt den Wert eines hochtransparenten verteilten Betriebssystems. Heute wird es kommerziell von einer Firma vertrieben.

3.2 Chorus

Chorus war von 1979 bis 1986 ein Projekt am staatlichen französischen Forschungsinstitut INRIA. Es sollte eine neue Generation von offenen, verteilten und erweiterbaren Betriebssystemen konstruiert werden. Chorus ist Message-basiert und hatte von Anfang an das Konzept eines kleinen Kerns, genannt Nukleus. Möglichst viele Betriebssystemfunktionen sollten in Server verlagert werden, die unter Chorus auch Subsysteme genannt werden. Im Nukleus wurden auch Echtzeiteigenschaften realisiert. Außerdem wurde großer Wert auf eine Modularität des gesamten Betriebssystems gelegt. Es ist nicht für eine spezielle Hardware zugeschnitten und kann auch Multiprozessoren unterstützen. Sehr wichtig wurde die Kompatibilität zu Unix genommen. Durch ein Unix-Subsystem kann Chorus binärkompatibel zu Unix sein.

Die erste Version, Chorus-V0, die ab 1980 lief, enthielt bereits das Konzept der Ports für die Nachrichtenübergabe. Sie war in Pascal geschrieben und auf Intel 8086 Maschinen implementiert, die über ein Netzwerk mit 50 Kb/s verbunden waren. Die zweite Version, Chorus-V1, lief ab 1982 auf Microcomputern, die auf Motorola 68000, später 68020 basierten und durch ein Standard Ethernet verbunden waren. Das Hauptziel dieser Version war die Anpassung an Multiprozessoren.

Der Schwerpunkt der dritten Version, Chorus-V2, ab 1984, war das Einbinden eines Unix Subsystems zum Erreichen der Unixkompatibilität. Dabei wurde die Funktionalität von Unix auf mehrere Prozesse verteilt. *Fork* und *exec* konnten auch auf entfernten Prozessoren ausgeführt werden. Weiter wurden Konzepte wie Ports, Messages und Remote Procedure Calls überarbeitet.

Die aktuelle Version, die hier auch beschrieben wird, ist Chorus-V3 (ab 1987). Chorus ist nun in C++ implementiert. Die generelle Struktur des Kerns und des Subsystems mußte aber niemals wirklich geändert werden. Lediglich die Protokolle der Rechnerkommunikation wurden von Version zu Version verfeinert.

Besonders hervorzuheben ist, daß eine transparente Kommunikationsmöglichkeit von Anfang an tief im Nukleus integriert worden ist. Dadurch erreicht man in allen höheren Schichten leicht und ohne weiteren Aufwand die Transparenz des Gesamtsystems. Wie auch bei Mach wurde das Konzept des Prozesses aufgeteilt, und zwar in einen Ausführungskontext, Actor genannt, und der Ausführung selbst, Thread genannt. Ein Thread wird zuweilen auch als leichtgewichtiger Prozeß bezeichnet. Er ist in einen Actor eingebettet, besteht selbst aber nur aus dem Befehlszähler und den Prozessorregistern. Ein Actor kann mehrere Threads enthalten, die parallel im gleichen Adreßraum ablaufen können.

3.2.1 Systemarchitektur

Ein Chorus-System besteht aus einem kleinen Nukleus und einer Anzahl von Systemservern, die als Subsysteme zusammenarbeiten (Bild 9). Diese aus zwei Ebenen bestehende Struktur bildet eine Basis für ein offenes Betriebssystem. Es kann sowohl auf eine zentralisierte als auch auf eine verteilte Rechnerkonfiguration übertragen werden. Insbesondere ist der Nukleus von Chorus nicht der Kern eines speziellen Betriebssystems, sondern er stellt ein allgemeines Werkzeug dar, um verschiedene Betriebssysteme zu realisieren, die unter Umständen sogar auf dem gleichen Rechnersystem angesiedelt werden können. Das Erscheinungsbild eines Betriebssystems wird nun nicht mehr vom

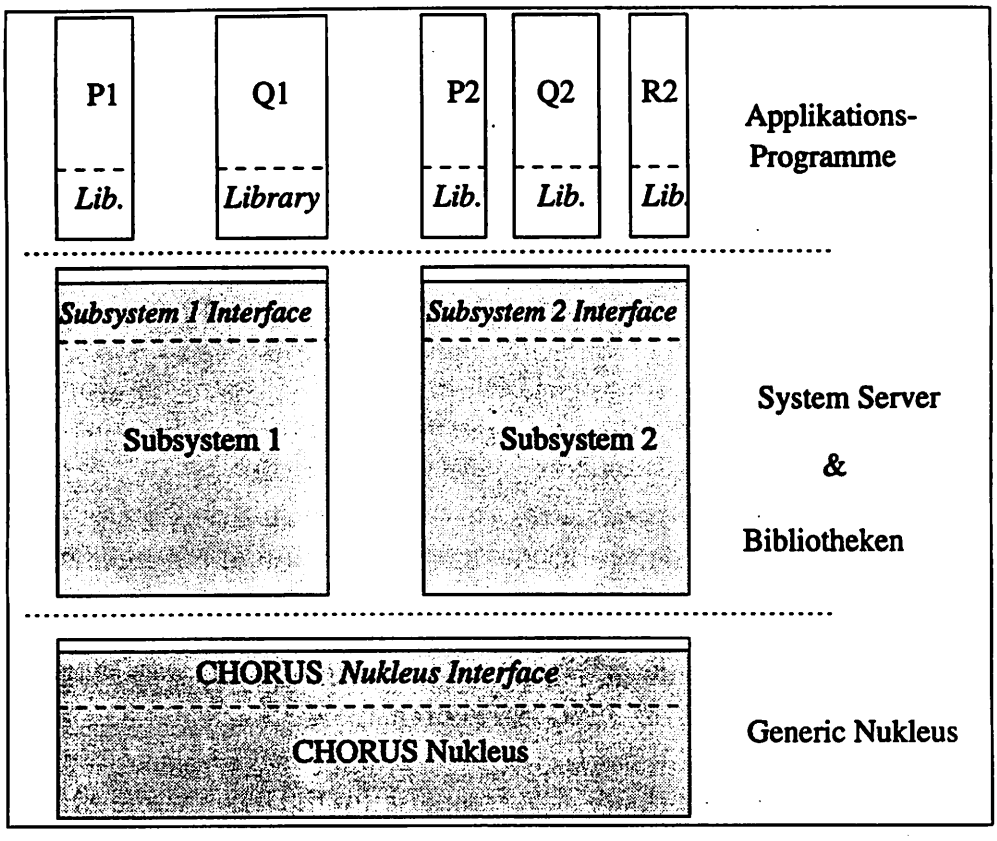

Abbildung 9: Chorus Architektur

Kern bestimmt, sondern von den darüber arbeitenden Servern. Eine Kompatibilität zu existenten Systemen läßt sich erreichen, indem das Subsystem ein entsprechendes Interface zur Verfügung stellt. Ein Beispiel für diese Vorgehensweise ist die Unix-Emulation, genannt Chorus/MiX.

Die Idee, höhere Funktionen eines Betriebssystemes in autonome Server auszugliedern, ist ca. 10 Jahre alt und auch in anderen neuen Betriebssystemen, z.B. Mach realisiert. Dadurch erreicht der Kern wieder eine überschaubare Größe (ca. 100 KB); man spricht von einer Microkernel-Architektur [2] In monolithischen Systemen (UNIX, VMS, OS/360) sind alle diese Funktionen Teil des Kerns. Durch die bessere Modularität der Microkernel-Architektur wird der Entwurf sauberer und die Portabilität des Systems erhöht.

3.2.2 Der Chorus-Nukleus

Der Chorus-Nukleus verwaltet die lokalen Ressourcen eines Rechners. Außerdem stellt er eine ortstransparente Interprozeßkommunikationsmöglichkeit zur Verfügung. Der Nukleus zerfällt in vier Teile (Bild 10).

- Der Supervisor behandelt Interrupts, Traps und Exceptions, die von der Hardware erzeugt werden.

[2] Auf diese Architektur wird im Kapitel über Mach noch näher eingegangen.

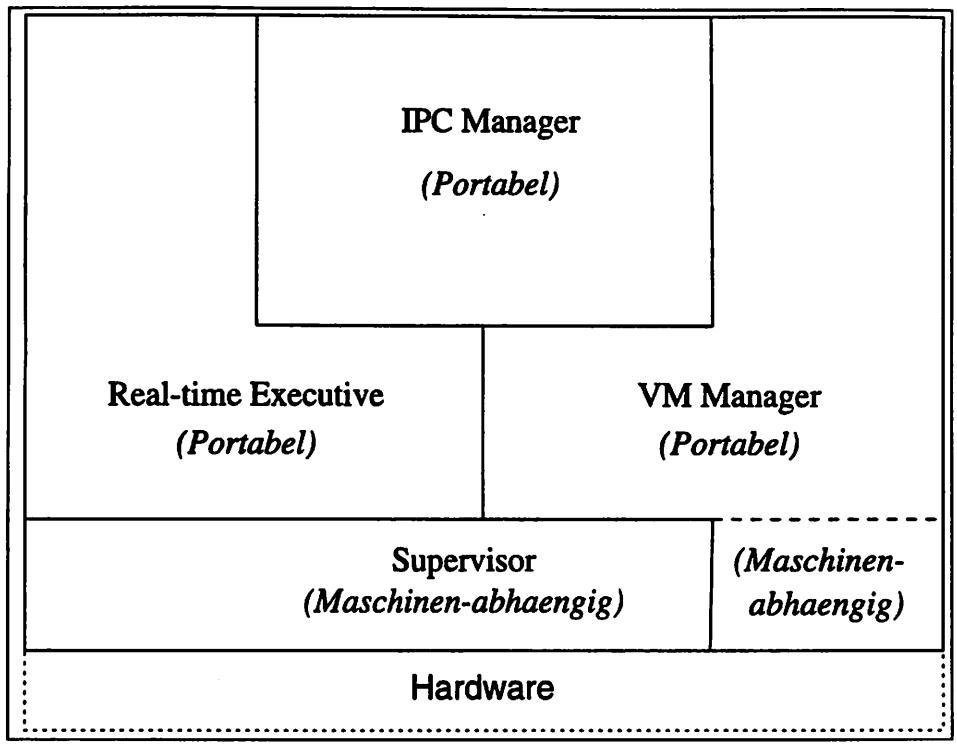

Abbildung 10: Chorus Nukleus

- Der Scheduler (auch Realtime-Executive genannt) kontrolliert die Prozessorvergabe, bietet eine hochauflösende Synchronisation und ein prioritätsbasiertes, preemptives Scheduling.
- Der Virtual Memory Manager verwaltet den lokalen Speicher und manipuliert die Virtual-Memory-Hardware.
- Der Inter-Process-Communication Manager bietet asynchronen Nachrichtenaustausch und den Remote Procedure Call Mechanismus.

Ein Chorus-System bietet drei unterschiedliche Interface-Ebenen. Der Nukleus bietet direkten Zugriff zu einem Interface der unteren Ebene. Ein Subsystem, implementiert durch eine Menge von kooperierenden Servern, repräsentiert typischerweise komplexe Betriebssystemabstraktionen. Benutzerbibliotheken, so wie die C-Bibliothek, bereichern das Interface weiter, indem sie übliche Programmierfunktionen zur Verfügung stellen. Die vier Komponenten des Nukleus sind weitestmöglich getrennt und frei von Abhängigkeiten. Der Chorus IPC-Mechanismus ist das Hauptwerkzeug zur Kommunikation mit Managern. In Chorus-V3 wurden aus Effizienzgründen folgende Funktionen in den Nukleus aufgenommen, die auch von Systemservern bereitgestellt werden könnten: Actor- und Port-Management, Name-Management und RPC-Management.

Folgende grundlegenden Abstraktionen werden vom Chorus-Nukleus zur Verfügung gestellt:

Actor: Einheit der Ressourcenzuweisung

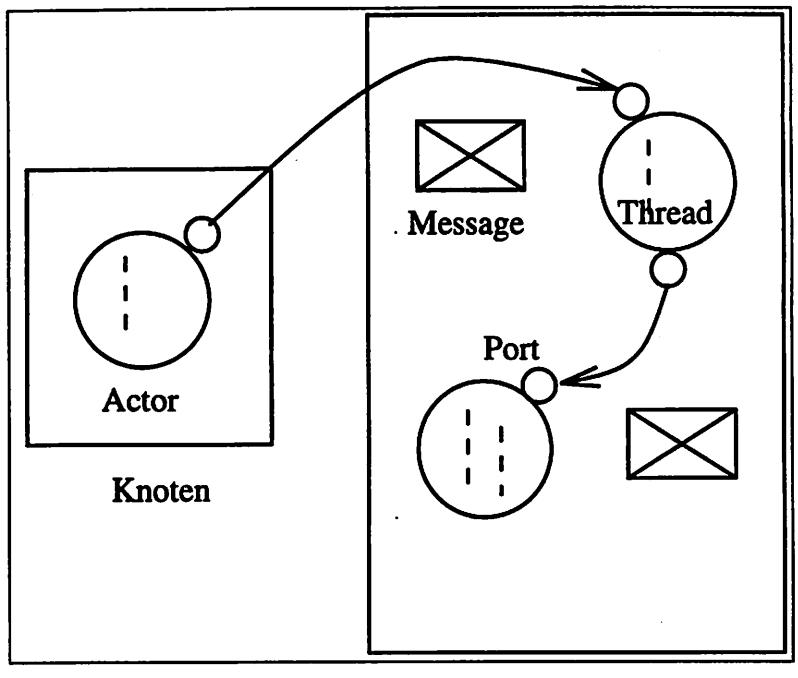

Abbildung 11: Actors, Threads, Ports und Messages

Thread: Einheit der sequentiellen Ausführung

Message: Einheit der Kommunikation

Port, Portgruppe: Einheit der Adressierung und Rekonfigurationsbasis

Region: Einheit der Strukturierung eines Actor-Adreßraumes

Unique Identifier: Globaler Name

Der Nukleus übernimmt sowohl die Repräsentation als auch die Operationen auf diesen Objekten. Folgende weitere Abstraktionen werden vom Nukleus zusammen mit den Subsystemservern verwaltet: Segmente, Capabilities und Protection Identifiers.

3.2.3 Actors und Threads

Ein Actor umschließt eine Menge von Ressourcen, einen virtuellen Speicher, eingeteilt in Regions, einen Kommunikationskontext, der aus einer Menge von Ports besteht, und einen Ausführungskontext, der von den Threads verwirklicht wird. Diese Threads teilen den geschützten Adreßraum und alle anderen Ressourcen des Actors. Der Adreßraum ist aufgeteilt in den Useradreßraum und den Systemadreßraum. Auf einem Rechner sind die Systemadreßräume aller Actors identisch und können nur in einem privilegierten Ausführungslevel benutzt werden.

Actors können verschiedene Privilegien haben. Ein Actor wird **trusted** genannt, wenn er vom Nukleus das Recht bekommt, sensitive Operationen auszuführen. **Privilegiert** wird er genannt, wenn er privilegierte Maschineninstruktionen ausführen kann. Normale Benutzeractors sind weder trusted noch privilegiert. Daneben gibt es noch

Systemactors. Sie sind zwar trusted, aber nicht privilegiert. Supervisoractors dagegen sind sowohl trusted als auch privilegiert. Während privilegierte Actors im geteilten und geschützten Systemadreßraum arbeiten, arbeiten andere Actors nur in ihrem privaten Useradreßraum.

Da jeder Actor seinen eigenen Useradreßraum hat, stellen sich Actors ebenso wie Prozesse als geschützte Einheiten dem Benutzer dar. Daher können auf einem Rechner problemlos mehrere Actors existieren. Jeder Actor ist aber fest mit einem Rechner verbunden. Seine Threads können nur Code ausführen und auf Daten zugreifen, die sich auf diesem Rechner befinden. Weder Actors noch Threads können von einem Knoten zu einem anderen migrieren. Wird ein Actor angesprochen, so kann daher sein Ort immer leicht gefunden werden. Alle nötigen Statusinformationen sind auf einem Knoten konzentriert. Die Bezeichnung von Actors geschieht durch Capabilities, die zu einem Defaultport dieses Actors führen. Durch den Besitz dieser Capability hat man alle Rechte für diesen Actor. Standardmäßig hat nur der Erzeuger eines Actors diese Capability, die er jedoch beliebig an Andere übertragen kann.

Innerhalb eines Actors kann es mehrere unabhängige Ausführungsfäden, genannt Threads, geben. Der Kontext eines Threads besteht nur aus dem Befehlszähler und den Registern des Prozessors. Eine Umschaltung zwischen Threads des gleichen Actors kann deshalb sehr schnell erfolgen. Jeder Thread ist exakt einem Actor zugeordnet, der seinen Ausführungskontext definiert. Innerhalb eines Actors können mehrere Threads erzeugt werden und nebenläufig ablaufen. Auf einem enggekoppelten Multiprozessor können die Threads eines Actors parallel auf verschiedenen CPUs laufen. Der Scheduler arbeitet auf der Ebene der Threads. Er betrachtet sie als unabhängige Einheiten und weiß nichts von der Existenz der Actors. Er arbeitet verdrängend (*preemptive*) auf Basis von Prioritäten. Ein Zeitscheibenverfahren ist ebenso möglich, wie eine Prioritätenherabsetzung einzelner Threads. Diese Kombination von Strategien erlaubt es, die Notwendigkeiten von Echtzeitanwendungen und interaktivem Multiuserzugriff auf dem gleichen Nukleus zu erfüllen.

Die Parallelität von Threads innerhalb eines Actors ist besonders bei Servern interessant. Erhalten Server, die nur aus einem Prozeß bestehen, einen Auftrag, so sind sie während der Abarbeitung dieses Auftrags für andere Klienten blockiert. Hat ein Server aber mehrere Threads, so kann er während der Abarbeitung eines Auftrags weitere Aufträge entgegennehmen und verarbeiten. Es wird daher möglich, ein nebenläufiges oder paralleles Client/Server-Modell zu implementieren. Besonders einfach wird auch die Benutzung mehrerer Recheneinheiten auf einem speichergekoppelten Multiprozessor. Da der Actor-Adreßraum im gemeinsamen Speicher liegt, können die Threads ganz einfach auf den verschiedenen Prozessoren parallel ausgeführt werden. Ein Algorithmus, der Parallelität erfordert, ist durch den Einsatz von Threads wesentlich effizienter zu implementieren als durch unabhängige Prozesse, da die Umschaltung zwischen Threads wesentlich weniger Aufwand verursacht als der Kontextswitch zwischen Prozessen oder zwischen Actors. Da alle Threads eines Actors im gleichen virtuellen Adreßraum laufen, bieten sie eine Möglichkeit für die Implementation paralleler Abläufe auf Prozessoren, die keinen virtuellen Speicher unterstützen, wie z.B. auf Transputern. Außerdem sind Threads nützlich, wenn, wie es bei Chorus möglich ist, Treiber auf einer höheren Ebene im System eingebunden werden. Hier müssen parallele Abläufe, wie z.B. Interrupts, gehandhabt werden, die aber im gleichen Adreßraum stattfinden sollten. Threads ermöglichen hier die Handhabung dieser parallelen Abläufe.

Für die Kommunikation zwischen Threads gibt es generell zwei Möglichkeiten. Allgemein ist Kommunikation und Synchronisation durch den Austausch von Nachrichten für den im Nukleus verankerten Kommunikationsmechanismus möglich (siehe nächstes Kapitel). Threads, die sich im gleichen Actor befinden, können für die Kommunikation aber ebenso den gemeinsamen Speicherbereich benutzen. Wenn der Befehlssatz der CPU Synchronisationsbefehle enthält (wie z.B. test und set), kann die Synchronisation auch damit erreicht werden. Die Kommunikation und Synchronisation über den geteilten Speicher ist effizienter, da es nicht nötig ist, den Nukleus aufzurufen.

3.2.4 Kommunikation

In Chorus erfolgt die Kommunikation durch den Austausch von Nachrichten. Eine Nachricht ist ein zusammenhängender Byte-String, der vom Adreßraum des Senders zum Adreßraum des Empfängers kopiert wird. In lose gekoppelten verteilten Systemen ist dies die einzige Möglichkeit zum Informationsaustausch. Die Verwendung von Messages bietet auch eine Möglichkeit, das generelle Synchronisationsproblem von Betriebssystemen zu lösen. Werden Messages nur lokal ausgetauscht, so kann durch Optimierungen dennoch die Leistung von shared-memory- Systemen erreicht werden.

Wie auch in Mach werden Nachrichten nicht direkt an Actors oder Threads gesandt, sondern an sogenannte Ports. Diese Ports bilden eine Abstraktion, die zwischen der Nachricht und dem Empfänger liegt, und erlauben dadurch eine flexible Verbindung der Serviceschnittstelle und des Servers selbst. Ein Port repräsentiert einerseits ein Ziel, an das Nachrichten gesandt werden können und andererseits eine Warteschlange von noch nicht gelesenen Nachrichten. Jeder Port ist genau einem Actor zugeordnet und ausschließlich die Threads dieses Actors können die einem Port gesandten Nachrichten empfangen. Allerdings kann ein Port von einem Actor zu einem anderen verschoben werden; dabei werden auch die nicht ausgelesenen Nachrichten des Ports verschoben. Welche Vorteile werden nun durch diese zusätzliche Indirektionsstufe erreicht?

- Ein Actor kann mehrere Ports besitzen und damit über mehrere Kommunikationspfade angesprochen werden.
- Da sich die Threads eines Actors einen Port teilen, kann die eingehende Information parallel bearbeitet werden.
- Die Abstraktion des Ports bildet die Basis für eine dynamische Rekonfiguration. Die Serviceleistung eines Actors kann ein anderer übernehmen, ohne daß es für die Klienten sichtbar wird.

Wenn ein Port erzeugt wird, gibt der Nukleus für ihn sowohl einen lokalen Identifikator als auch einen globalen Namen (unique-identifier) zurück. Der lokale Name kann nur in dem Actor benutzt werden, zu dem der Port gerade gehört.

Portgruppen Ports können zu Gruppen zusammengefaßt werden. Dadurch ist es möglich, Nachrichten an eine ganze Gruppe von Threads zu senden. Ein Port kann Mitglied von mehreren Gruppen sein. Ähnlich wie Ports bekommen auch Portgruppen globale Namen. Wenn eine Portgruppe erzeugt wird, ist sie zunächst leer. Ports können

nun hinzugefügt und gelöscht werden. Fest zugewiesene Portgruppen-Identifikatoren können von Subsystemen benutzt werden, um bekannte Dienste anzubieten. Auf diese Art können Systemserver dynamisch an globale Namen gebunden werden. Zur Übersetzungszeit des Betriebssystems definiert das Subsystem statisch die Namen der Portgruppen, durch die die Klienten Dienste anfordern können. Zur Bootzeit erzeugen die Subsysteme Ports, auf denen sie Anforderungen empfangen und beantworten. Die Subsysteme fügen diese Ports in die statisch definierten Portgruppen ein. So kann man beim Schreiben der Klienten diese festen Portgruppen-Namen benutzen.

Die Operationen auf den Portgruppen wie Einfügen und Entfernen von Ports müssen vom Nukleus kontrolliert werden, um die Sicherheit zu garantieren. Dafür stellt der Nukleus dem Erzeuger einer Gruppe den *group manipulation key* zur Verfügung. Mit ihm kann man die Inhalte der Gruppen modifizieren und er kann zwischen Actors beliebig übertragen werden.

Semantik der Kommunikation Der IPC Mechanismus des Nukleus erlaubt Threads, asynchrone Nachrichten zu versenden oder den RPC zu benutzen. Beim Senden einer asynchronen Nachricht ist der Sender nur so lange blockiert, wie die Nachricht lokal bearbeitet wird. Das System garantiert nicht, daß die Nachricht den Zielport erreicht. Ist dieser ungültig, wird die Nachricht einfach verworfen.

Der RPC Mechanismus hingegen erlaubt die Verwendung des "Client-Server" Modells. Er garantiert, daß der Klient eine Antwort vom richtigen Server erhält. Über RPC kann ein Klient auch feststellen, ob seine Anforderung vom Server empfangen worden ist, falls der Server zwischendurch abgestürzt ist oder ob der Kommunikationsweg frei ist.

Diese asynchrone IPC und der synchrone RPC sind die einzigen Kommunikationsdienste, die der Nukleus anbietet. Die asynchrone IPC ist eine tragfähige Basis, um höhere Protokolle in Subsystemen darauf aufzubauen. Im erfolgreichen Fall benötigt sie nur eine geringe Netzwerk-Bandbreite. Daher führt sie zu besserer Leistung und besserer Ausbaufähigkeit auf großen Netzwerken. Der RPC ist ein einfaches Konzept, das leicht zu verstehen und in existenten Sprachen bereits vorhanden ist. Der Nukleus sieht außerdem keine Flußkontrolle vor, da die Anforderungen daran stark variieren.

Beim Versenden von Nachrichten an Portgruppen sind folgende Adressierungsmethoden möglich:

- Broadcast zu allen Ports in der Gruppe (außer bei RPC)
- Senden an einen Port in der Gruppe
- Senden an einen Port in der Gruppe auf einem gegebenen Rechner
- Senden an einen Port in der Gruppe, der sich auf demselben Rechner befindet wie ein gegebener unique-Identifikator.

Rekonfiguration Der Nukleus bietet die Basis, um zuverlässige Dienstleistungen zu implementieren. Er erlaubt, daß diese Dienstleistungen dynamisch rekonfiguriert werden, ohne daß dabei die Klienten gestört werden. Dies ist entweder mit der Migration

eines Ports oder durch die Benutzung von Portgruppen möglich. So kann zum Beispiel ein Server durch eine neue Version ersetzt werden. Bei der Migration eines Ports wird dieser einfach einem anderen Actor zugewiesen. Dieser ist dann in der Lage, die Nachrichten des Ports zu lesen. Auch alle nicht gelesenen Nachrichten werden mitverschoben. So kann der Betrieb der Klienten ohne Unterbrechung weitergehen. Die Portmigration verlangt jedoch, daß die involvierten Server aktiv sind. Wenn ein Server zusammenbricht, kann sein Port nicht mehr migriert werden.

Die Benutzung von Portgruppen stellt eine passive Maßnahme zur Rekonfiguration dar. Nachrichten, die an Portgruppen gesandt werden, werden automatisch an einen Port weitergeleitet, der in der Lage ist, diese Nachrichten zu verarbeiten. Wenn ein Server ausfällt, fügt man einfach den Port eines Ersatzservers in diese Gruppe ein und streicht den ersten Port heraus.

Der Nukleus unterstützt einen *Protection Identifier* für jeden Actor und jeden Port; die Struktur dieses Identifiers ist festgelegt. Jedoch assoziiert der Nukleus keine Bedeutung mit ihm. Diese Identifikatoren können nur von *Trusted Threads* verändert werden. Ein Actor oder Port erbt bei seiner Erzeugung den *Protection Identifier* des Actors, der ihn erzeugt hat. Mit jeder Nachricht überträgt der Nukleus die *Protection Identifier* des sendenden Actors und des Ports. Durch das Auswerten dieser Werte kann der Empfänger einer Nachricht sicher die Identität eines Service-Anforderers feststellen. So lassen sich Objekte schützen, die von Subsystem-Servern verwaltet werden. Da alle Dienste über den Chorus IPC-Mechanismus aufgerufen werden, wird damit das ganze System geschützt.

Network-Manager Der Network-Manager ist zuständig für die Kommunikation zwischen unterschiedlichen Knoten, sowohl für die IPC, als auch für den RPC. Sein Kommunikationskern implementiert das Kommunikationsprotokoll, seine untere Schicht verwaltet die Anfragen an den Network-Device-Manager, der den Gerätetreiber für die Netzwerkkarten implementiert. Der Network-Manager benutzt zwei Protokolle: Das erste enthält Chorus-spezifische Eigenschaften, wie das Lokalisieren von entfernten Ports und die Behandlung von Rechnerausfällen. Das zweite Protokoll ist verantwortlich für die Datenübertragung und ist betriebssystemunabhängig. Da Standards in Chorus, sofern möglich, benutzt werden sollten, entspricht das zweite Protokoll dem OSI Schichtenmodell.

3.2.5 Virtuelle Speicherverwaltung

Der Adreßraum eines Actors ist eingeteilt in *regions*. Daten, auf die der Actor zugreifen will, werden von der virtuellen Speicherverwaltung in eine seiner *regions* eingeblendet (Mapping). Solche Daten heißen bei Chorus Segmente. Sie sind global und durch Capabilities bestimmt. Sie sind generell durch Hintergrundspeicherobjekte implementiert, wie Dateien oder Swap-Bereiche. Segmente werden von System-Actors verwaltet, die unter Chorus *Mappers* genannt werden. Der Zugriffsschutz und die Konsistenz dieser Objekte wird von diesen Servern definiert und realisiert.

Auf ein Segment kann zugegriffen werden durch das Einblenden eines Teils in eine *region* oder explizit durch einen Aufruf der Chorus-Segment-Lese- oder Schreib-Systemaufrufe. Der Mapper selbst hat nur ein Lese/Schreib-Interface, das sowohl als Konsequenz dieser

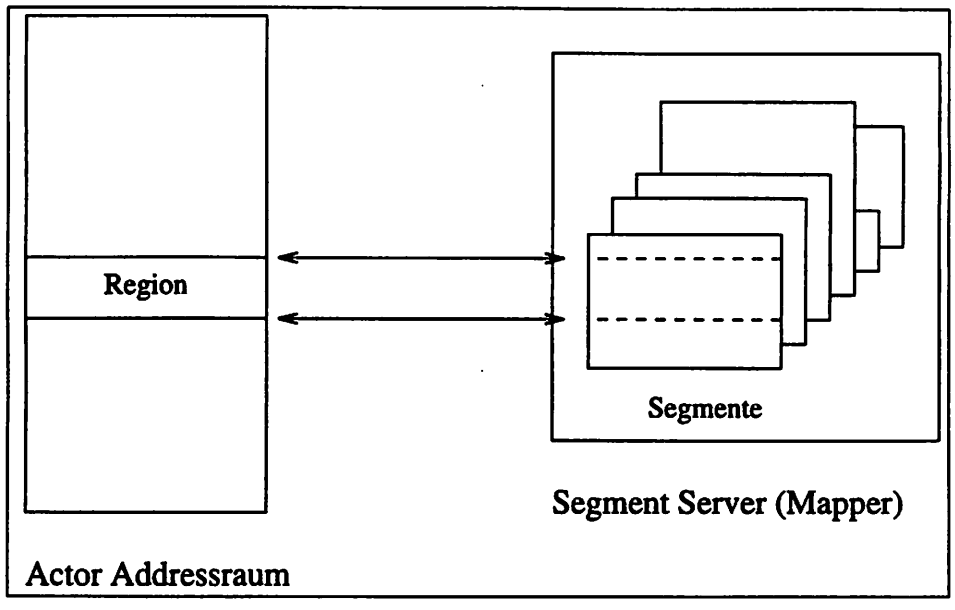

Abbildung 12: Regions und Segmente

Systemaufrufe als auch bei Anforderungen vom Nukleus, z.B. als Ergebnis eines Seitenfehlers, benutzt wird. Damit auch auf sehr große Segmente zugegriffen werden kann, kommt *demand paging* zum Einsatz. Jeder Zugriff zu einer Adresse innerhalb einer *region* bewirkt damit einen Zugriff auf das dort eingeblendete Segment, vorausgesetzt, die Zugriffsrechte werden eingehalten. *Regions* innerhalb des Systemadreßraums eines Knotens können nur von Trusted Threads manipuliert werden. Threads, die *regions* innerhalb des Systemadreßraums benutzen, ersparen den Overhead einer Adreßumschaltung. Chorus nutzt diese Tatsache, indem es IPC-Anforderungen zwischen Subsystemen nicht kopiert, sondern nur einblendet, sofern sie auf demselben Knoten sind. Zur Unterstützung von Echtzeitanwendungen können Seiten von *regions* im Arbeitsspeicher vom Paging ausgenommen werden (locking).

Für jedes Segment verwaltet der Nukleus einen lokalen Cache von physikalischen Seiten. Er enthält Seiten von Segment-Servern, die vielleicht zukünftig wieder gebraucht werden. Schreibzugriffe werden im Cache durchgeführt und gelangen zum Segment-Server zurück, wenn die Seite ausgelagert werden muß oder wenn der Cache für dieses Segment ausdrücklich für ungültig erklärt wird.

Wird nun ein Segment von verschiedenen Actors benutzt, wird seine Konsistenz gewährleistet, da alle auf den gleichen lokalen Cache im Arbeitsspeicher zugreifen (Bild 13). Wenn ein Segment auf unterschiedlichen Knoten benutzt wird, so gibt es pro Knoten einen lokalen Cache und der Segment-Server ist verantwortlich, die Konsistenz dieser Caches zu erhalten (Bild 14). Auch Chorus verwendet die Speicherverwaltung, um unnötige Kopiervorgänge zu vermeiden. Unter dem Namen "deferred copy" wird bei der Duplizierung von Daten ein *copy-on-write*-Mechanismus[3] benutzt. Er erhöht die Leistungsfähigkeit beim Kopieren großer Datenmengen von einem Actor zum andern, wie es z.B. bei der UNIX *fork*-Operation nötig ist, erheblich.

[3]siehe Kapitel über Mach

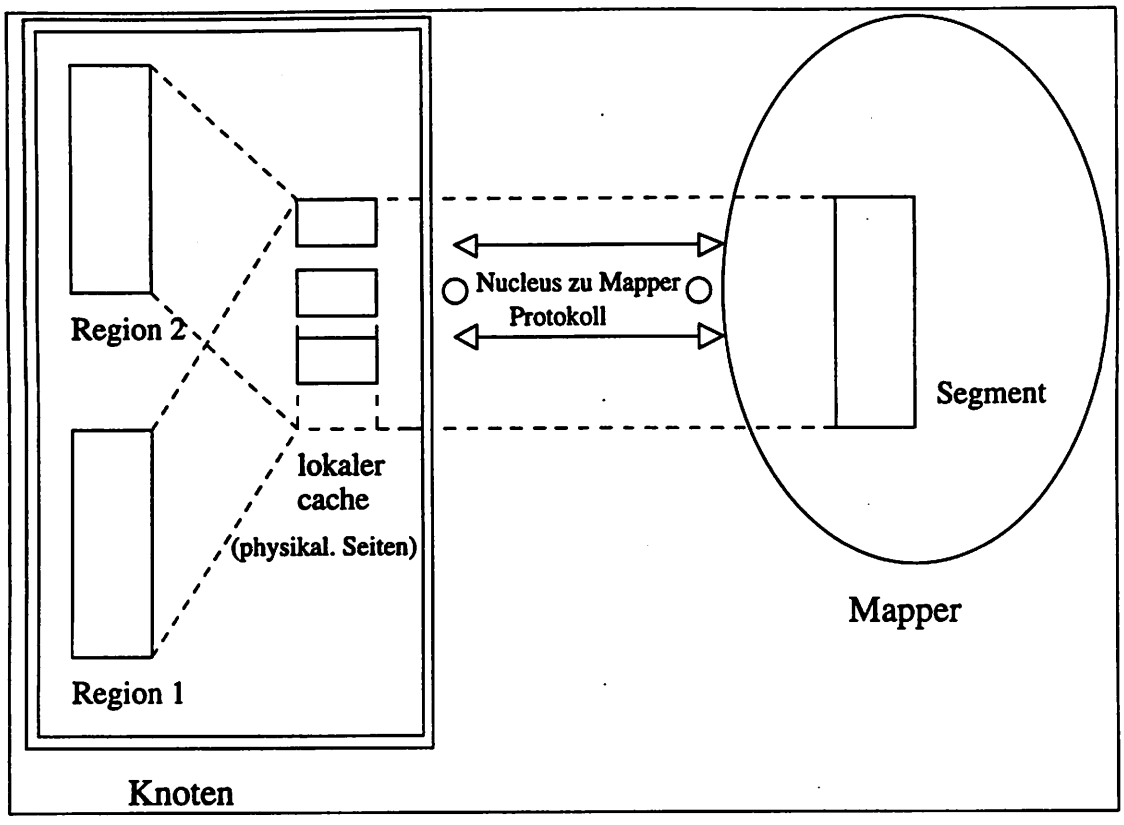

Abbildung 13: Lokaler Puffer

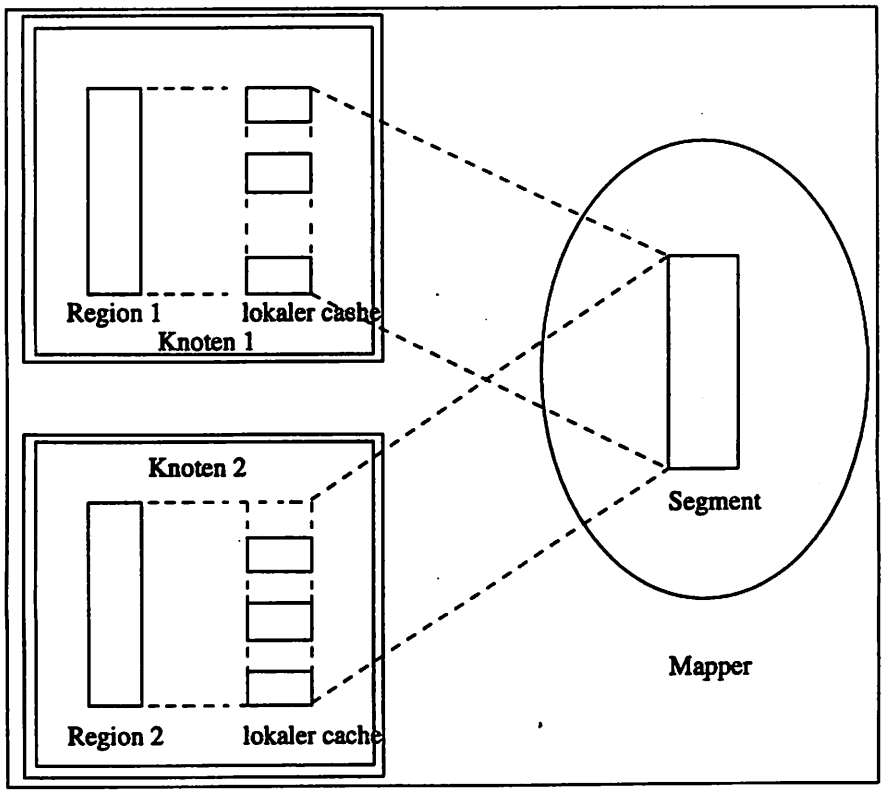

Abbildung 14: Verteilter Puffer

3.2.6 Unterbrechungen

Eine Besonderheit von Chorus besteht darin, daß die Behandlung von Unterbrechungen in vergleichsweise hohe Schichten verlegt werden kann. Der Nukleus ermöglicht es einem User- oder Systemthread, die Kontrolle über Ausnahmebedingungen (z.B. Division durch Null) innerhalb eines Actors zu übernehmen. Wenn der Nukleus eine Ausnahmebedingung erkennt und der Actor ihr einen Port zugewiesen hat, so sendet der Nukleus eine Nachricht zu diesem Port, der die Bedingung beschreibt. Threads, die auf diesem Port Nachrichten empfangen, können dann die für diese Bedingung vorgesehene Aktion durchführen. Wenn kein Port zugewiesen worden ist, dann führen Ausnahmebedingungen zur Beendigung des Threads, der die Bedingung verursacht hat. Diese Möglichkeit entspricht in etwa dem Signalmechanismus unter Unix. In Supervisoractors können Programmierer direkt E/A-Events und Unterbrechungen bearbeiten. Der Supervisor kann einen Geräteinterrupt direkt von der Hardware zu einem Thread weitergeben. Beim Auftreten eines Interrupts sichert er den Kontext des unterbrochenen Threads und ruft eine priorisierte Sequenz von Anwenderroutinen auf, die mit dem Interrupt verbunden wurde. Danach wird der Scheduler aufgerufen.

Eine ganz ähnliche Möglichkeit existiert für synchrone Unterbrechungen (Traps). Der Supervisor ruft in diesem Fall die Routinen eines gegebenen Actors auf. Dadurch können Subsysteme die Schnittstelle eines Betriebssystems simulieren. Durch ein Abfangen des Interrupts 21 (hexadezimal) können zum Beispiel alle Aufrufe eines DOS-Programmes empfangen werden. Diese Möglichkeit wird auch vom Chorus/MiX Prozeßmanager genutzt, um ein Unix Subsystem zu implementieren.

3.2.7 Benennung

In einem verteilten System ist es wichtig, einen konsistenten Weg zu haben, allen Objekten einen global gültigen Namen zur Verfügung zu stellen, der auf allen Knoten des Systems gültig ist. Alle Chorus-Objekte wie Actors, Ports und Segmente werden in einer einheitlichen Weise unter Benutzung von einheitlichen Identifikatoren (Unique Identifier – UI) benannt. Der Nukleus garantiert, daß ein UI auf mindestens einem Knoten existiert und kein zweites Mal verwendet wird. Ein UI ist eine 128 Bit Struktur, die den Erzeugerknoten des Objektes enthält, so daß es leicht gefunden werden kann. Ein UI kann beliebig im Chorus System unter Benutzung beliebiger Mechanismen übertragen werden.

Der Nukleus kontrolliert nicht die Ausbreitung von UIs; dafür ist das zuständige Subsystem verantwortlich.

Lokale Identifikatoren Lokale Identifikatoren (LIs) werden innerhalb des Kontextes eines Servers benutzt, um seine Ressourcen zu bezeichnen. Diese Identifikatoren werden durch ganze Zahlen repräsentiert und vom lokalen Server erzeugt. Sie können zwischen Einheiten innerhalb des Chorus-Systems übertragen werden, aber haben nur eine Bedeutung, wenn sie Ressourcen des Servers ansprechen, der sie erzeugt hat. Im einzelnen können diese Identifikatoren einen Index zu einer Server-Tabelle oder einen Zeiger zu einer Server-Datenstruktur repräsentieren.

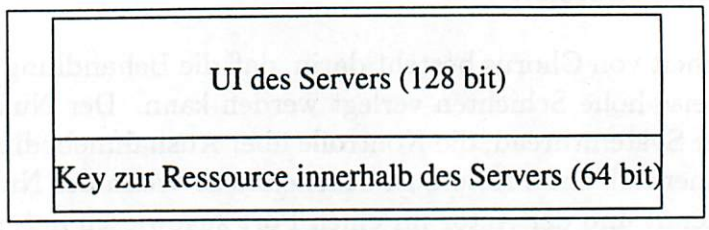

Abbildung 15: Aufbau einer Capability

Capabilities Einige Objekte werden nicht direkt vom Nukleus implementiert, sondern von externen Servern. Diese Objekte werden durch globale Namen, sogenannte **Capabilities**, bezeichnet. Eine Capability ist zusammengesetzt aus einem UI und einem Schlüssel. Der UI bezeichnet den Server, der das Objekt verwaltet. Der Schlüssel ist eine Server-spezifische Zahl, die das Objekt und seine Zugriffsrechte identifiziert. Die Struktur und die Semantik des Schlüssels werden vom Server bestimmt.

Während ihrer Lebenszeit können Ports nacheinander an verschiedene Actors gebunden werden. Der Network-Manager ist verantwortlich dafür, daß der Ort des Zielports einer IPC oder eines RPC verborgen bleibt. Er verwendet den vom Nukleus realisierten *UI-location-service*, um den Knoten zu finden, auf dem sich der Zielport befindet. Um einen entfernten Zielport zu finden, benutzt der Network-Manager einen Cache von bekannten Ports und Portgruppen. Wenn ein entfernter Port aufgrund von Portmigration oder eines Zusammenbruchs ungültig wird, wird sein Eintrag im Cache entfernt. Bis zum Empfang einer negativen Bestätigung beginnt der Network-Manager einen Suchvorgang und verbreitet Anfragen an entfernte Network-Manager. Von Ports, die sich nicht im Cache befinden, wird zunächst angenommen, daß sie sich auf ihrer Erzeugerseite befinden.

3.2.8 Subsysteme

Gruppen von Actors, die zusammenarbeiten, um eine definierte Schnittstelle für Benutzerprogramme zu exportieren, nennt man Subsysteme. Diese Subsysteme reproduzieren eine bekannte Betriebssystemschnittstelle und erlauben damit eine Binärkompatibilität zu solchen Systemen. Sie implementieren typischerweise Betriebssystemabstraktionen einer hohen Ebene wie Prozesse, Sicherungsmodelle und Datenobjekte. Sie konstruieren diese Abstraktionen unter Verwendung der Basiselemente, die der Nukleus bietet. Subsysteme können ihre Dienste sicher exportieren, indem sie einen Zugriff durch synchrone Unterbrechungen (Traps) bieten. Code und Daten können in den Systemadreßraum geladen werden, um die Leistung zu erhöhen.

Im Beispiel (Bild 16) greifen mehrere User-Actors auf die Dienste eines Chorus-Subsystems zu. Das Subsystem ist durch das Trap-Interface geschützt. Ein Teil davon ist als ein System-Actor implementiert, der im Systemadreßraum arbeitet und ein anderer Teil ist als User-Actor implementiert. Bisher wurde nur ein Unix-Subsystem implementiert, das nun besprochen wird.

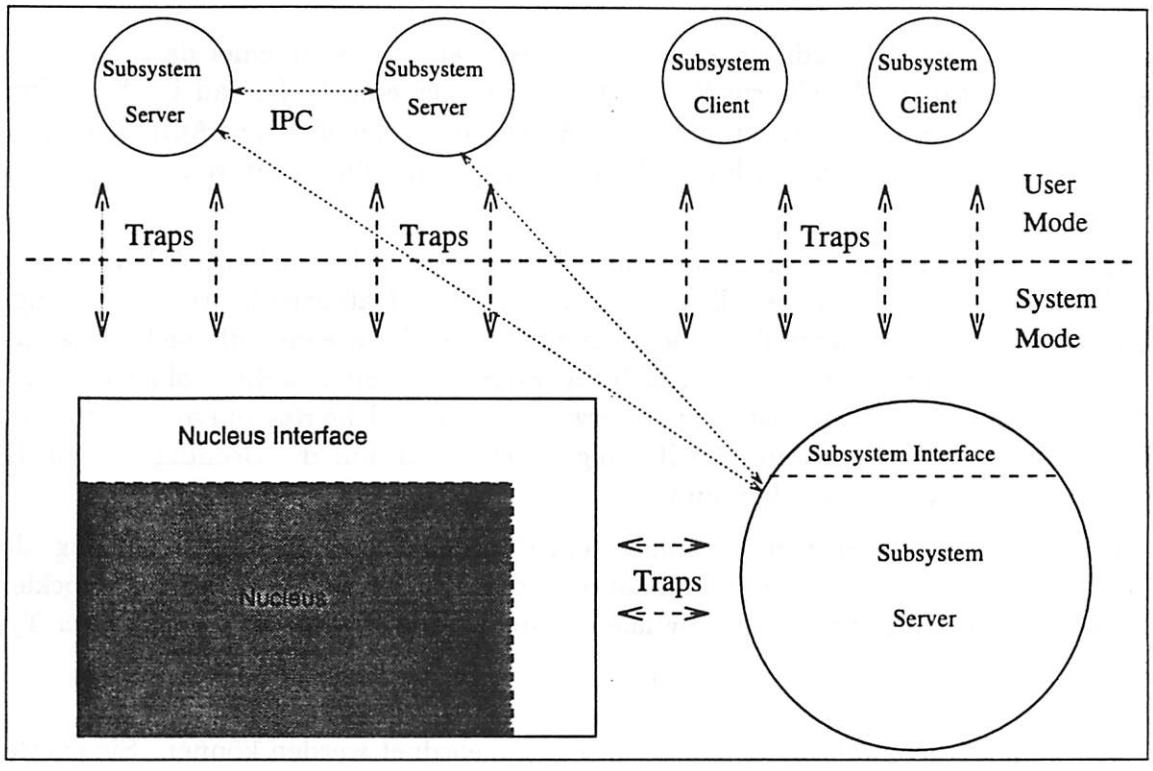

Abbildung 16: Struktur eines Chorus Subsystems

3.2.9 Das UNIX-Subsystem

Das erste Subsystem, das auf der Chorus-Architektur verwirklicht wurde, war ein Unix System V Subsystem. Zusammen mit dem Chorus-Nukleus wird es als Chorus/MiX bezeichnet.

Durch das Aufsetzen von UNIX auf Chorus wurden eine Reihe von bisherigen traditionellen Einschränkungen von UNIX durchbrochen. UNIX-Dienste wurden in einer modularen Weise implementiert, als eine Sammlung von Servern. Diese autonomen Server haben die Eigenschaft, daß sie nur dort vorhanden sein müssen, wo sie benutzt werden. Sie können entsprechend den Bedürfnissen dynamisch geladen oder gestoppt werden. Der Chorus-Nukleus bietet nun die Echtzeiteigenschaften, die vorher in UNIX fehlten. Die UNIX-Dienste arbeiten nun in einer verteilten Umgebung. Das Dateisystem ist voll verteilt und Dateizugriff ist ortstransparent. Dateibäume sind automatisch zu einem Gesamtbaum verbunden. Auf die netzwerktransparente IPC läßt sich von der UNIX-Schnittstelle aus zugreifen, so daß eine einfache Entwicklung verteilter Applikationen möglich wird.

Da jeder UNIX-Prozeß als Actor implementiert ist, ergibt sich automatisch die Möglichkeit, mehrere Threads innerhalb eines Prozesses laufen zu haben. Um das Thread-Interface des Nukleus vom Thread-Interface, das vom Subsystem Chorus/MiX zur Verfügung gestellt wird, zu unterscheiden, werden die Chorus/MiX-Threads im Folgenden als U_Threads bezeichnet.

Das U_Thread-Interface enthält elementare Funktionen für das Erzeugen und das Entfernen von U_Threads, das Suspendieren und Wiederaufnehmen ihrer Ausführung und das Modifizieren ihrer Prioritäten. Threads bereiten Probleme, wenn es um die Semantik der klassischen UNIX-Systemdienste *fork* und *exec* geht. So stellt sich die Frage,

was mit mehreren Threads eines Prozesses passieren soll, wenn einer davon einen *fork*-Aufruf durchführt. In Chorus/MiX wird nur dieser eine U_Thread im Sohn-Prozeß dupliziert, die anderen werden entfernt. Analog dazu hat der *exec*-Aufruf zur Konsequenz, daß im Prozeß nur noch ein Thread arbeitet, unabhängig davon, wieviele vorher existierten.

U_Threads können parallel Systemaufrufe ausführen. Das Chorus/MiX-Subsystem enthält aber Ressourcen wie z.B. gemeinsam benutzte Datenstrukturen, deren Zugriffe kritische Abschnitte darstellen. Beim Zugriff eines U_Threads auf solche Ressourcen werden alle anderen, die auf die gleiche Ressource zugreifen wollen, blockiert, bis der erste Zugriff abgeschlossen ist. Typischerweise werden solche Ressourcen nicht für lange Zeitdauer blockiert, sie werden z.B. freigegeben, wenn auf die Beendigung von E/A-Operationen gewartet werden muß.

Die Einführung der Threads veranlaßte ein Überdenken der Signalbearbeitung. Jeder U_Thread hat seinen eigenen Signalkontext, der aus den Signal-Handlern, blockierten Signalen und einem Stack von Signalen besteht. Die Signale werden in zwei Typen eingeteilt:

- Signale, die eindeutig einem U_Thread zugeordnet werden können. Sie entstehen aus synchronen Unterbrechungen und werden nur diesem U_Thread zugestellt.
- Signale, die logisch an den ganzen Prozeß gerichtet sind. Zu dieser Gruppe gehören Signale, die vom Terminaltreiber bei Betätigung einer Interrupt-Taste erzeugt werden und Signale, die vom *kill*-Systemaufruf generiert worden sind.

Signale des zweiten Typs werden an alle U_Threads im Prozeß gesandt. Jeder U_Thread kann verschiedene Signal-Handler für das Signal einstellen; U_Threads, die keinen Signal-Handler eingestellt haben, werden gemäß der Voreinstellung behandelt, d.h. meistens abgebrochen.

Architektur Die Funktionalität von UNIX kann entsprechend den verschiedenen Objekttypen, die behandelt werden, in verschiedene Klassen eingeteilt werden: Prozesse, Dateien, Sockets oder Geräte. Der Entwurf des UNIX-Subsystems sieht eine saubere Definition der Schnittstellen zwischen diesen Klassen vor und resultiert in einer modularen Struktur. Chorus/MiX wurde als eine Gruppe von System-Servern implementiert, die auf dem Chorus-Nukleus laufen. Jede Ressourcen-Gruppe ist isoliert und wird von einem dedizierten Server verwaltet. Die Kommunikation zwischen diesen Servern geschieht über die Chorus-IPC. Im UNIX-Subsystem gibt es folgende Typen von Servern: Prozeßmanager, Filemanager, Socketmanager, Devicemanager und vom Benutzer eingebrachte Server (Bild 17). Jeder dieser Server ist als Chorus-Actor implementiert. Die Fehlersuche ist wie bei jedem anderen Benutzerprogramm mit Hilfe von Standard-Debuggern möglich. Normalerweise enthalten Chorus/MiX-Server mehrere Threads, wobei jeder eine Anfrage an den Server verarbeitet. Der Kontext des Threads stellt damit den Status der Anfrage dar. Ein Server erhält Anfragen normalerweise über seinen Port, der auch in einer Portgruppe Mitglied sein kann. Es gibt aber auch die Möglichkeit, eine synchrone Interrupt-Schnittstelle zur Verfügung zu stellen. Diese Schnittstelle kann dann kompatibel mit existierenden UNIX-Systemschnittstellen sein.

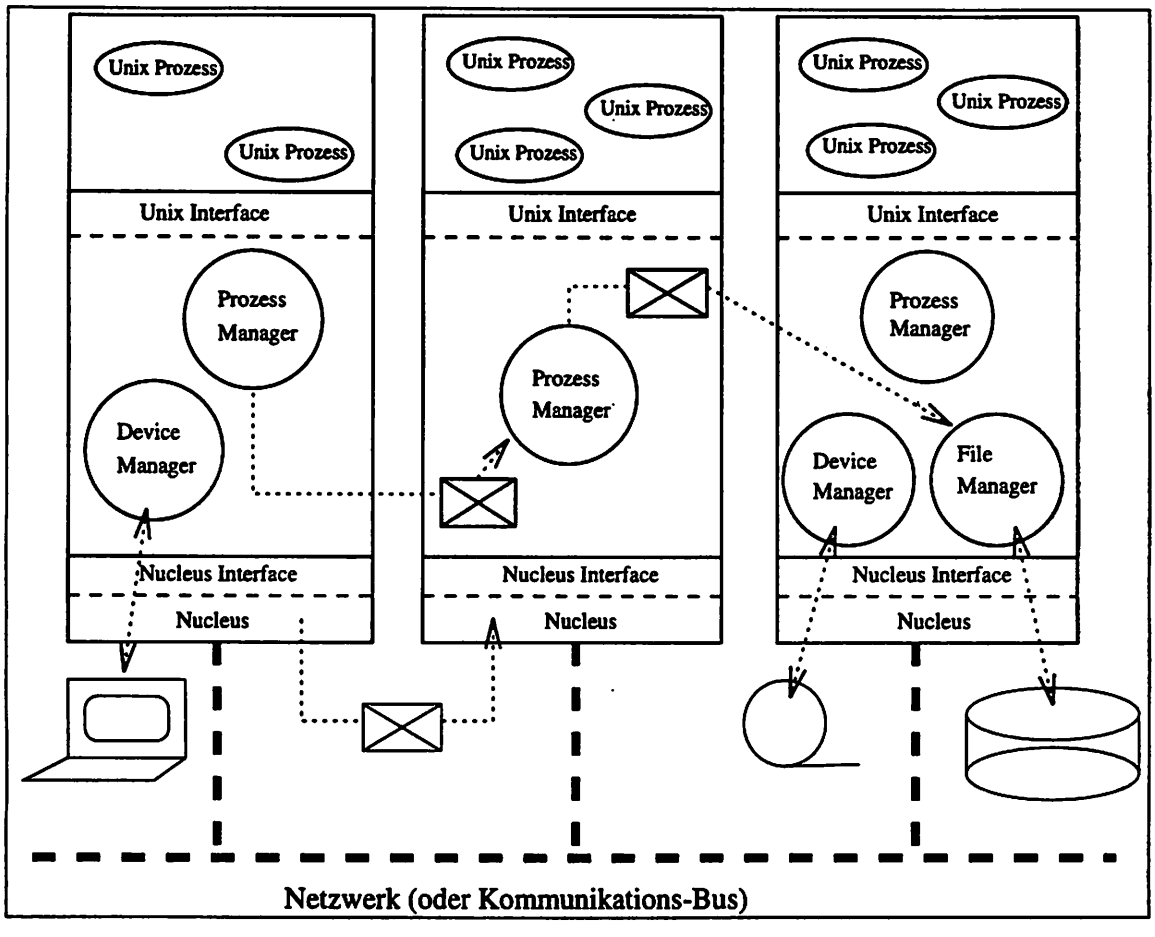

Abbildung 17: Chorus MiX

Der Prozeßmanager bietet diese Schnittstelle an, um Binärkompatibilität mit einem SCO-UNIX-System zu gewährleisten.

Um die Portierung von Gerätetreibern vom UNIX-Kern in einen Chorus-Server zu ermöglichen, wurde eine UNIX-Kernel-Emulationsbibliothek entwickelt, die mit dem Gerätetreiber zusammengebunden wird. Sie enthält Funktionen wie *sleep, wakeup, splx, copyin* und *copyout.* Die Interrupt-Service-Routinen sind einige der wenigen Teile eines traditionellen UNIX-Gerätetreibers, die modifiziert werden müssen, um in die Chorus-Umgebung zu passen.

Prozeßmanager Auf jedem Knoten, der UNIX emuliert, stellt ein Prozeßmanager die mit dem Prozeßbegriff verbundenen Funktionen wie Prozeßerzeugung, Kontextkonsistenz, Vererbung und Signalisierung zur Verfügung. Außerdem implementiert er die Eintrittspunkte, die die Benutzerprozesse erwarten. Da UNIX-Systemaufrufe durch synchrone Unterbrechungen (Traps) erfolgen, läßt der Prozeßmanager den Chorus-Nukleus seine Routinen an solche Traps binden und erreicht damit eine Binärkompatibilität. Der Prozeßmanager selbst erledigt Anforderungen, die mit Prozessen und Signalen zu tun haben wie *fork, exec* und *kill.* Für andere Systemaufrufe wie *open, close, read* und *write* ruft der Prozeßmanager andere Subsystem-Server auf.

Wenn z.B. ein Chorus/MiX-Prozeß den *open*-Systemaufruf durchführt, wird ein Trap generiert und vom lokalen Prozeßmanager behandelt. Dieser benutzt den RPC, um die Anfrage an den Filemanager weiterzugeben. Falls die angegebene Datei nicht vom Filemanager behandelt wird, wie es für Gerätedateien der Fall ist, wird die Anfrage automatisch vom Filemanager an den zugehörigen Devicemanager weitergegeben. Für die Kommunikation mit anderen Managern benutzt der Prozeßmanager den Chorus-RPC.

Für die entfernte Ausführung und die entfernte Signalbehandlung kooperieren die Prozeßmanager folgendermaßen:

- Jeder Prozeßmanager hat einen dedizierten Port, genannt der *request*-Port, um entfernte Anforderungen zu empfangen. Diese werden dann von den Threads des Prozeßmanagers bearbeitet.
- Die *request*-Ports der Prozeßmanager werden in eine Portgruppe eingefügt. Jeder beliebige Prozeßmanager kann so über eine einheitliche Adresse angesprochen werden.
- Operationen, die nicht lokal ausgeführt werden können, werden an den zugehörigen Prozeßmanager weitergeleitet.

Dateimanager Jeder Knoten, der eine Platte besitzt, benötigt einen Dateimanager. Plattenlose Knoten können Anforderungen an entfernte Dateimanager durchführen. Die beiden Hauptfunktionen der Dateimanager sind das UNIX-Dateimanagement und das Arbeiten als Segment-Server (Mapper) für den Chorus-Nukleus.

- Als Dateiserver verarbeiten die Filemanager die herkömmlichen UNIX-Aufrufe, die sie über IPC übermittelt bekommen. Sie realisieren auch einige spezielle

UNIX-Dienste für das Prozeßmanagement wie die Behandlung von gemeinsamen Schreib-Lesezeigern mehrerer Prozesse auf Dateien. Außerdem bilden sie den Pfadnamen einer ausführbaren Datei in die Chorus-Capabilities für Text- und Datensegmente ab.

- Als Segmentserver implementieren Dateimanager Dienste wie sie vom Chorus Virtual Memory Manager verlangt werden, wie z.B. das Verwalten des Hintergrundspeichers. Sie benutzen das erwähnte Mapper-Interface und kontrollieren die Cache-Konsistenz, wenn entfernte Zugriffe auf lokale Daten auftreten.

- Der Chorus/MiX-Dateisystem-Cache (Buffer cache) wird ebenfalls durch Benutzung des Chorus Virtual Memory Managers implementiert. Dies optimiert die physikalische Speicherbelegung und macht die Implementation von Systemaufrufen wie *read* und *write* netzwerktransparent.

Die Menge der Typen des UNIX-Dateisystems wurde erweitert, um die Benennung von Diensten zu erlauben, die über Ports erreicht werden können. Ein neuer Dateityp, genannt *symbolic port,* kann ins Dateisystem aufgenommen werden. Er assoziiert einen Dateinamen mit dem Unique Identifier eines Ports. Wird bei der Pfadnamenabarbeitung eine solche Datei erreicht, so wird die Anfrage an den assoziierten Port weitergeleitet.

Chorus/MiX enthält außerdem die aus dem BSD-System bekannten symbolischen Links. In Verbindung mit symbolischen Ports ermöglichen sie ein sehr mächtiges und flexibles verteiltes Dateisystem.

Devicemanager Geräte wie Terminals, Pseudo-Terminals, grafische Displays, Bandlaufwerke und Netzwerkcontroller werden von Devicemanagern bedient. Ihre Art und Anzahl hängt von den Anforderungen eines Knotens ab. Sie können außerdem dynamisch geladen oder entfernt werden. Die "character device switch table" (*cdevsw*) der traditionellen UNIX-Systeme wird durch folgenden Mechanismus ersetzt: Während der Initialisierung sendet der Devicemanager seinen Port und die Major-Number des Gerätes, das er bedient, zum Dateimanager. Wenn solche Major-Number bei der Pfadnamenbearbeitung erreicht werden, wird die Anforderung zum genannten Port gesandt.

Die Homogenität des Subsystemkonzeptes ermöglicht es dem Systemanwender, neue Server zu entwickeln und sie in das System als Actors zu integrieren. Es ist einfach, mit neuen Systemservern zu experimentieren, da sie auf Anwenderebene entwickelt und getestet werden können, ohne das laufende System zu stören.

Struktur eines UNIX-Prozesses Ein traditioneller UNIX-Prozeß kann als einzelner Thread gesehen werden, der innerhalb eines Adreßraumes läuft. Folglich wird jeder UNIX-Prozeß in einen einzigen Chorus-Actor abgebildet, dessen Systemkontext vom Prozeßmanager verwaltet wird. Sein Adreßraum enthält *regions* für Text, Data und Stack.

Zusätzlich baut der Prozeßmanager einen Kontroll-Port und einen Kontroll-Thread in jeden UNIX-Prozeß-Actor. Sie sind nicht sichtbar für den Benutzer dieses Prozesses. Kontroll-Threads innerhalb des Prozeßkontextes haben zwei Haupteigenschaften:

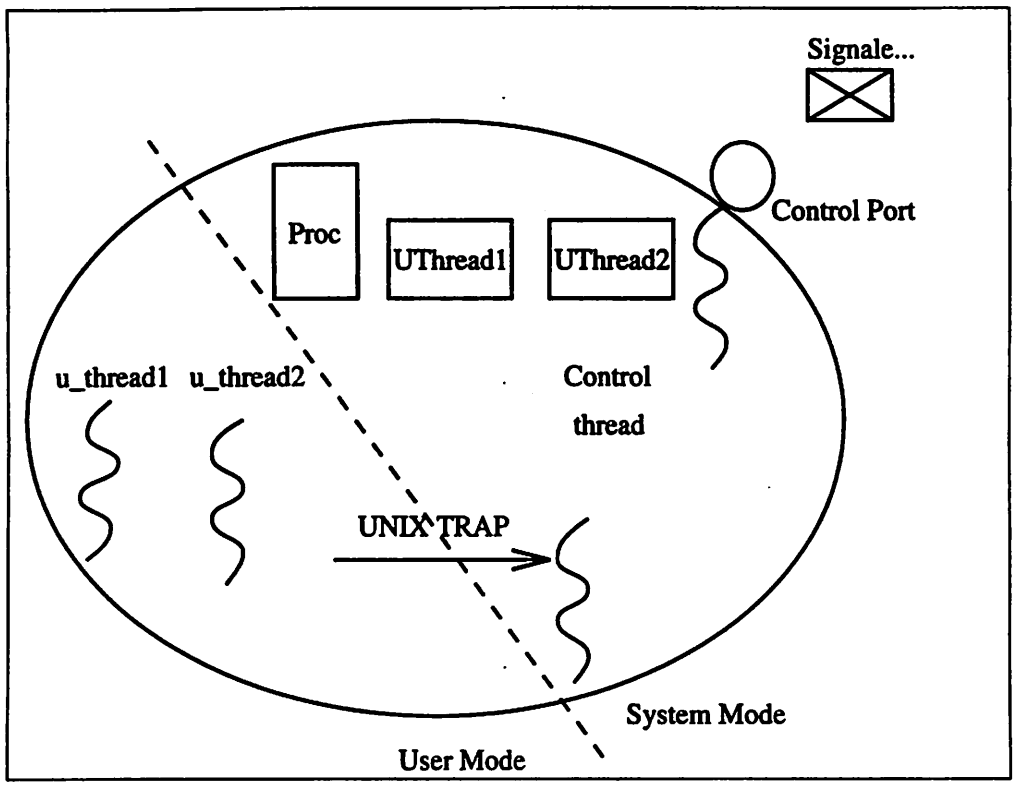

Abbildung 18: Unix Prozeß als Chorus Actor

- Da sie den Prozeßadreßraum teilen, können sie leicht Zugriffe und Modifikationen im Speicherbereich durchführen, wie sie für die Signalverarbeitung oder das Debugging benötigt werden.
- Sie sind jederzeit bereit, um asynchrone Ereignisse auszuführen, wie sie bei Signalen oder der Prozeßbeendigung notwendig sind. Diese Ereignisse werden als Nachrichten an den Kontroll-Port gesandt (Bild 18).

Die Möglichkeit mehrerer Threads in einem Prozeß beeinflußte die Implementation des Prozeßkontrollblocks. Der Prozeßkontrollblock wurde in zwei Teile aufgeteilt: Einen Prozeßkontext und einen U_Thread-Kontext Diese Kontrollstrukturen werden vom Prozeßmanager des Knotens verwaltet, auf dem der Prozeß gerade rechnet. Sie sind nur durch den Prozeßmanager zugreifbar. Zum Prozeßkontext gehört ebenfalls der Kontroll-Port des Vaterprozesses. Wenn ein Prozeß endet, kann so eine Exit-Statusmeldung an den Vaterprozeß gesandt werden.

Jeder Prozeß wird eindeutig bezeichnet durch eine globale 32-Bit-Prozeßidentifikationsnummer, die aus zwei Teilen zu je 16 Bit besteht. Diese Teile enthalten den Erzeugungsknoten und eine traditionelle UNIX-Prozeß-Identifikation.

Beispiele Das aktuelle und das Root-Directory werden im Prozeßkontrollblock durch Capabilities repräsentiert. Wenn ein *open*-Befehl abgesetzt wird, sucht die *open*-Routine des Prozeßmanagers nach einem freien Filedeskriptor, konstruiert eine Nachricht, die den Pfadnamen der Datei enthält und sendet diese an den Port des Servers, der das

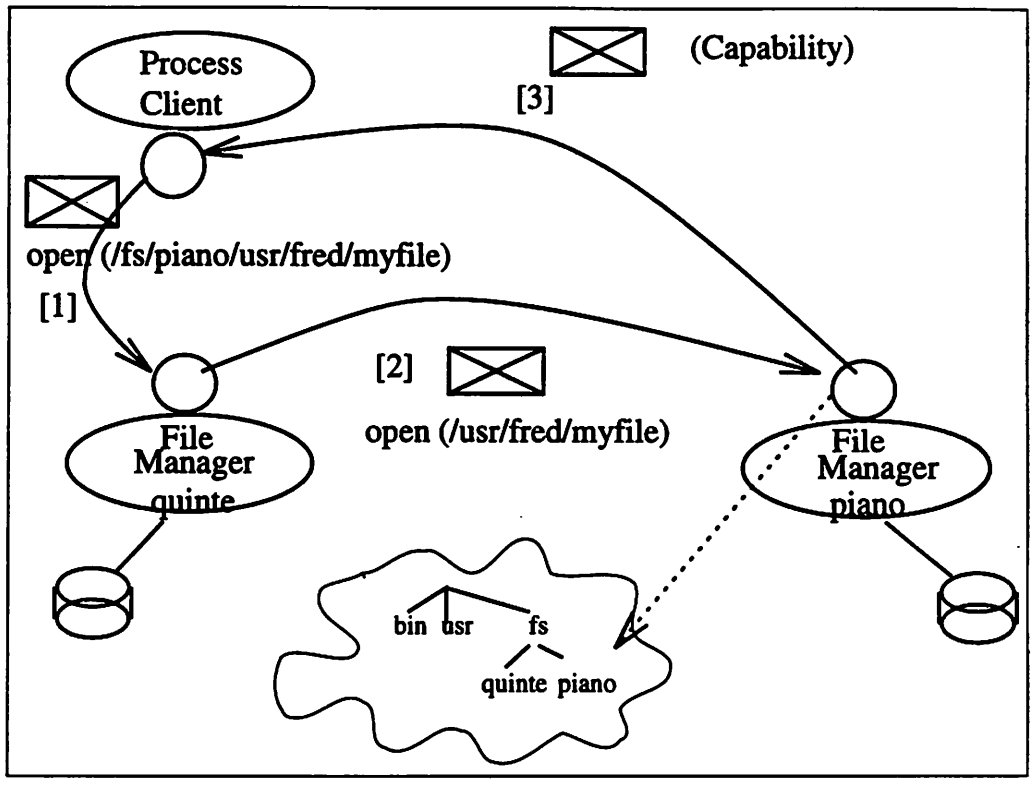

Abbildung 19: Dateizugriff

momentane oder das Root-Verzeichnis verwaltet, abhängig davon, ob der Pfadname absolut oder relativ ist (Bild 19). Sei dieser Pfadname der /fs/piano/usr/fred/myfile und /fs/piano ein symbolischer Port eines Filemanagers, der auf dem Knoten namens Piano läuft. Dieser Pfadname wird an den Filemanager gesandt, der das Root-Directory des Prozesses verwaltet, im Beispiel der Filemanager von Quinte. Er analysiert den Pfadnamen, entdeckt, daß piano ein symbolischer Port ist und gibt eine Nachricht mit dem unverarbeiteten Rest des Pfadnamens (/usr/fred/myfile) an den symbolischen Port. Der Filemanager auf Piano erhält die Nachricht, vervollständigt die Analyse des Pfadnamens, öffnet die Datei, erzeugt eine Capability und sendet sie dem Klienten zurück, der den *open*-Aufruf durchgeführt hat. Alle nachfolgenden Anforderungen, die diese Datei betreffen, werden direkt an den Filemanager von piano gesandt. Der Dateimanager von Quinte wird nicht weiter involviert. Die Beschreibung des *exec*-Algorithmus demonstriert die Interaktionen zwischen den Subsystem-Servern und den Prozeßkontroll-Threads. Der lokale Prozeßmanager führt dabei folgende Aktionen durch:

- Analog zum *open*-Aufruf wird via RPC der Dateiname dem zuständigen Filemanager übergeben. Dieser erzeugt zwei Capabilities, die später für den Text- und den Datenbereich benötigt werden. Wenn der im Prozeßkontrollblock voreingestellte "child-execution-Knoten" ein anderer ist als der momentane, wird dieser Knoten auf Gültigkeit getestet.

- An dessen Prozeßmanager wird eine Anfrage geschickt, die den Prozeß- und Thread-Kontrollblock des aufrufenden U_Threads, die Argumente des *exec*-Aufrufs

und das Environment und die vom Filemanager zurückgegebenen Capabilities enthält.

Diese Anfrage wird via RPC an die Portgruppe der Prozeßmanager geschickt. Der Prozeßmanager der entfernten Seite initialisiert einen Prozeßkontrollblock für den neuen Prozeß aus den mitgelieferten Informationen wie Prozeßidentifikationsnummer, offene Dateien und verbrauchte Rechenzeit. Er erzeugt die Chorus-Einheiten, die den Prozeß implementieren, also einen Actor, einen Kontroll-Thread, einen Kontroll-Port und Regions. Dann initialisiert er den Prozeß:

- Er installiert Argumente und die Environment-Strings im Prozeßadreßraum.
- Er erzeugt einen U_Thread, der die Ausführung des neuen Programmes beginnen wird – nach *exec* enthalten alle Prozesse nur einen einzigen Thread.
- Er sendet eine Antwortnachricht an den U_Thread, der den *exec*-Aufruf durchführte.

Der aufrufende U_Thread erhält die Antwort, gibt die Prozeßkontrollblöcke frei und entfernt den Actor, der den Prozeß implementiert. Der Chorus-Nukleus gibt darauf alle von diesem Actor belegten Ressourcen frei. Neben den Diensten des UNIX-Systems stehen UNIX-Prozessen auch die Dienste des Chorus-Nukleus zur Verfügung. Sie können mit Hilfe der Chorus IPC mit anderen UNIX-Prozessen, mit anderen Chorus-Actors oder Servern anderer Subsysteme kommunizieren, außerdem können sie Ports erzeugen und manipulieren und remote procedure calls ausführen.

Die Echtzeitfähigkeiten des Nukleus sind privilegierten Applikationen auf dem Subsystem-Level zugänglich. Insbesondere ist es möglich, Interrupt-Service-Routinen dynamisch zu Hardware-Interrupts zuzuordnen. In der Interrupt-Service-Routine können UNIX-Server eine Anfrage sofort bearbeiten oder einen Großteil der Verarbeitung an einen dedizierten Thread im Serverkontext delegieren. Diese Funktionalität erlaubt Gerätetreibern, Unterbrechungen für kürzere Zeiträume zu maskieren als es in Standard-UNIX-Implementierungen der Fall ist.

3.2.10 Zusammenfassung

Die Zielsetzung von Chorus war ein Betriebssystem mit hoher Qualität und voller Funktionalität. Folglich wurden Abwägungen zwischen Leistungsfähigkeit und Reichhaltigkeit des Entwurfs oft zugunsten der Performance entschieden. Die allgemeine Natur der Möglichkeiten des Nukleus verhinderte Fähigkeiten mit komplexer Semantik. Eigenschaften wie obligatorische Sicherheitsmechanismen, applikationsorientierte Protokolle und Fehlertoleranz kommen im Chorus-Nukleus nicht vor. Anstelle dessen sieht der Nukleus Primitiva vor, mit Hilfe derer diese Konstrukte in Subsystemen implementiert werden können.

Chorus ist eine kommunikationsbasierte Architektur, die auf einem kleinen Nukleus basiert, der verteilte Verarbeitung und Kommunikation auf einer unteren Ebene integriert und die Dienste durch eine Gruppe vom Subsystemservern implementiert, um Standardbetriebssystemschnittstellen zu erreichen. Die modulare Architektur erleichtert

die Fehlersuche, da Ressourcen innerhalb Actors isoliert sind und klare Messagepassing-Schnittstellen zwischen Actors definiert sind.

Chorus gehört zu einer neuen Generation von offenen, verteilten und erweiterbaren Betriebssystemen. Dies erlaubt dynamische Konfigurationen des Systems und seiner Applikationen über einen weiten Bereich von Hardware- und Netzwerkkonfigurationen einschließlich paralleler und Multiprozessorsysteme.

Inhaltsverzeichnis

Literatur

[1] M. J. BACH: *The Design of the UNIX Operating System*, Englewood Cliffs N. J.: Prentice Hall Inc. 1986

[2] J. K. ANNOT, M. D. JANSSENS: *Multi Processor UNIX*, Delft University of Technology - Department of Electrical Engineering, Juli 1985

[3] R. REGN: *Synchronisation und Prozeßwechselmechanismen für ein symmetrisches Multiprozessor UNIX*, Diplomarbeit, Universität Erlangen-Nürnberg 1988

[4] A. TANENBAUM, R. VAN RENESSE: *Distributed Operating Systems*, Computing surveys Vol 17, No.4, Dez. 1985

[5] R. SANDBERG ET. AL.: *Design and Implementation of the Sun network Filesystem*, Proceedings of the USENIX Conference, Summer 1985

[6] A. SILBERSCHATZ, J. PETERSON, P. GALVIN: *Operating System Concepts 3rd. Ed, Addison Wesley 1991*,

[7] A. BRICKER, M. LITZKOW: *Condor technical summary*, University of Wisconsin

[8] B. WALKER, G. POPEK ET.AL.: *The LOCUS distributed Operating System*, CACM 1983

[9] M. ROZIER, V. ABRASSIMOV ET.AL.: *Overwiev of the CHORUS Distributed Operating System*, Chorus systèmes 1990

[10] R. RASHID: *From RIG to Accent to Mach: The Evolution of a Network Operating System*, Carnegie Mellon University 1987

[11] M. ACCETTA, R. BARON ET.AL.: *Mach: A new kernel Foundation for Unix Development*, Carnegie Mellon University 1986

[12] M. YOUNG, A. TEVANIAN ET.AL.: *The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System*, Carnegie Mellon University 1987

[13] A. FORIN, J. BARRERA ET.AL.: *Design, Implementation and Performance Evaluation of a Distributed Shared Memory Server for Mach*, Carnegie Mellon University 1988

[14] D. GOLUB, R. DEAN ET.AL.: *Unix as an Application Program*, Carnegie Mellon University 1990

Liste der bisher erschienenen Ulmer Informatik-Berichte

Einige davon sind per FTP von `ftp.informatik.uni-ulm.de` erhältlich
Die mit * markierten Berichte sind vergriffen

List of technical reports published by the University of Ulm

Some of them are available by FTP from `ftp.informatik.uni-ulm.de`
Reports marked with * are out of print

91-01 *Ker-I Ko, P. Orponen, U. Schöning, O. Watanabe*
Instance Complexity

91-02* *K. Gladitz, H. Fassbender, H. Vogler*
Compiler-Based Implementation of Syntax-Directed Functional Programming

91-03 *Alfons Geser*
Relative Termination

91-04* *J. Köbler, U. Schöning, J. Toran*
Graph Isomorphism is low for PP

91-05 *Johannes Köbler, Thomas Thierauf*
Complexity Restricted Advice Functions

91-06 *Uwe Schöning*
Recent Highlights in Structural Complexity Theory

91-07 *F. Green, J. Köbler, J. Toran*
The Power of Middle Bit

91-08* *V.Arvind, Y. Han, L. Hamachandra, J. Köbler, A. Lozano, M. Mundhenk. A. Ogiwara, U. Schöning, R. Silvestri, T. Thierauf*
Reductions for Sets of Low Information Content

92-01* *Vikraman Arvind, Johannes Köbler, Martin Mundhenk*
On Bounded Truth-Table and Conjunctive Reductions to Sparse and Tally Sets

92-02* *Thomas Noll, Heiko Vogler*
Top-down Parsing with Simulataneous Evaluationof Noncircular Attribute Grammars

92-03 *Fakultät für Informatik*
17. Workshop über Komplexitätstheorie, effiziente Algorithmen und Datenstrukturen

92-04 *V. Arvind, J. Köbler, M. Mundhenk*
Lowness and the Complexity of Sparse and Tally Descriptions

92-05 *Johannes Köbler*
Locating P/poly Optimally in the Extended Low Hierarchy

92-06 *Armin Kühnemann, Heiko Vogler*
Synthesized and inherited functions -a new computational model for syntax-directed semantics

92-07 *Heinz Fassbender, Heiko Vogler*
A Universal Unification Algorithm Based on Unification-Driven Leftmost Outermost Narrowing

92-08 *Uwe Schöning*
On Random Reductions from Sparse Sets to Tally Sets

92-09 *Hermann von Hasseln, Laura Martignon*
Consistency in Stochastic Network

92-10 *Michael Schmitt*
A Slightly Improved Upper Bound on the Size of Weights Sufficient to Represent Any Linearly Separable Boolean Function

92-11 *Johannes Köbler, Seinosuke Toda*
On the Power of Generalized MOD-Classes

92-12 *V. Arvind, J. Köbler, M. Mundhenk*
Reliable Reductions, High Sets and Low Sets

92-13 *Alfons Geser*
On a monotonic semantic path ordering

92-14* *Joost Engelfriet, Heiko Vogler*
The Translation Power of Top-Down Tree-To-Graph Transducers

93-01 *Alfred Lupper, Konrad Froitzheim*
AppleTalk Link Access Protocol basierend auf dem Abstract Personal Communications Manager

93-02 *M.H. Scholl, C. Laasch, C. Rich, H.-J. Schek, M. Tresch*
The COCOON Object Model

93-03 *Thomas Thierauf, Seinosuke Toda, Osamu Watanabe*
On Sets Bounded Truth-Table Reducible to P-selective Sets

93-04 *Jin-Yi Cai, Frederic Green, Thomas Thierauf*
On the Correlation of Symmetric Functions

93-05 *K.Kuhn, M.Reichert, M. Nathe, T. Beuter, C. Heinlein, P. Dadam*
A Conceptual Approach to an Open Hospital Information System

93-06 *Klaus Gaßner*
Rechnerunterstützung für die konzeptuelle Modellierung

93-07 *Ullrich Keßler, Peter Dadam*
Towards Customizable, Flexible Storage Structures for Complex Objects

94-01 *Michael Schmitt*
On the Complexity of Consistency Problems for Neurons with Binary Weights

94-02 *Armin Kühnemann, Heiko Vogler*
A Pumping Lemma for Output Languages of Attributed Tree Transducers

94-03 *Harry Buhrman, Jim Kadin, Thomas Thierauf*
On Functions Computable with Nonadaptive Queries to NP

94-04 *Heinz Faßbender, Heiko Vogler, Andrea Wedel*
Implementation of a Deterministic Partial E-Unification Algorithm for Macro Tree Transducers

94-05 *V. Arvind, J. Köbler, R. Schuler*
On Helping and Interactive Proof Systems

94-06 *Christian Kalus, Peter Dadam*
Incorporating record subtyping into a relational data model

94-07 *Markus Tresch, Marc H. Scholl*
A Classification of Multi-Database Languages

94-08 *Friedrich von Henke, Harald Rueß*
Arbeitstreffen Typtheorie: Zusammenfassung der Beiträge

94-09 *F.W. von Henke, A. Dold, H. Rueß, D. Schwier, M. Strecker*
Construction and Deduction Methods for the Formal Development of Software

94-10 *Axel Dold*
Formalisierung schematischer Algorithmen

94-11 *Johannes Köbler, Osamu Watanabe*
New Collapse Consequences of NP Having Small Circuits

94-12 *Rainer Schuler*
On Average Polynomial Time

94-13 *Rainer Schuler, Osamu Watanabe*
Towards Average-Case Complexity Analysis of NP Optimization Problems

94-14 *Wolfram Schulte, Ton Vullinghs*
Linking Reactive Software to the X-Window System

94-15 *Alfred Lupper*
Namensverwaltung und Adressierung in Distributed Shared Memory-Systemen

94-16 *Robert Regn*
Verteilte Unix-Betriebssysteme

Ulmer Informatik-Berichte

ISSN 0939-5091

Herausgeber: Fakultät für Informatik

Universität Ulm, Oberer Eselsberg, D-89069 Ulm