Diplomarbeit
zur Erlangung des akademischen Grades
Diplom-Informatiker

Kommunikation in verteilten Systemen mit gemeinsamem Speicher

Thomas Röblitz
Matrikel 111035

Lehrstuhl für Softwaretechnik und Theorie der Programmierung II
Institut für Informatik
Mathematisch-Naturwissenschaftliche Fakultät II
Humboldt-Universität zu Berlin

Berlin, den 6. November 2000

Gutachter:
Dr. Frank Müller
Prof. Dr. Alexander Reinefeld

Thesen zur Diplomarbeit

In den letzten zehn Jahren wurden zahlreiche DSM-Systeme für Cluster von Rechnern entwickelt. Cluster zeichnen sich durch ein gutes Preis-Leistungs-Verhältnis aus, da sie aus Standardkomponenten aufgebaut werden können. Neue Netzwerktechnologien ermöglichen einen effizienteren Datenaustausch zwischen Programmen.

Von dieser Entwicklung können DSM-Systeme, die gemeinsamen Speicher in einem verteilten System simulieren, profitieren. Typischerweise unterstützen DSM-Systeme nicht asynchrone, Multi-Threading-sichere Kommunikation und Multi-Threading. Diese Eigenschaften sind aber hilfreich, um Berechnungen mit Nachrichtenoperationen zu überlagern und die Latenzzeiten der Netzwerke und der Kommunikationssoftware zu tolerieren. Das DSM-System DSM-Threads besitzt diese Eigenschaften.

Auf der anderen Seite ermöglichen Kommunikationsbibliotheken für moderne Netzwerke, Programmen, die nicht Multi-Threading verwenden, einen effizienten Zugriff auf diese Netzwerke. Sie verwenden Polling, um festzustellen, ob eine Nachrichtenoperation abgeschlossen ist.

Die wesentliche Aufgabe bestand darin, anhand einer Fallstudie zu zeigen, wie moderne Kommunikationsarchitekturen, in DSM-Systeme wie z.B. DSM-Threads integriert werden können. Wenn diese modernen Kommunikationsarchitekturen indirekte Kommunikation und Notifikation durch Signale unterstützen, lassen sie sich einfach in DSM-Systeme mit den genannten, besonderen Eigenschaften einbinden.

Ein untergeordneter Aspekt dieser Studie betrifft das Polling, mit dem das Ende einer Nachrichtenoperation festgestellt wird. Es ist inadäquat für DSM-Systeme, die auf asynchroner Kommunikation und Multi-Threading aufbauen. Selbst wenn man die Polling-Frequenz soweit erhöhen könnte, daß asynchron eintreffende Nachrichten unverzüglich bearbeitet werden, wäre dies keine Lösung, da mit jedem Polling ein Mehraufwand entsteht. Je mehr Polling-Tests erfolgen, ohne daß eine Nachricht eingetroffen ist, umso mehr wird die Leistungsfähigkeit des DSM-Systems reduziert.

Ein weiterer untergeordneter Aspekt betrifft die Funktionalität von Kommunikationsbibliotheken. Diese ermöglichen neben der Kommunikation in der Regel auch die statische verteilte Ausführung, wobei die Kommunikation von dem Modell der verteilten Ausführung abhängig ist. Die Integration solcher Bibliotheken in DSM-Systeme, die die dynamische Erzeugung von Parallelität bzw. Verteiltheit unterstützen, wird dadurch erschwert. Das zugrundegelegte DSM-System verliert dabei die Flexibilität bzgl. der Rekonfiguration der eingesetzten Rechner.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Traditionell wurden und werden in Rechenzentren Großrechner oder spezielle Parallelrechner eingesetzt. Heute werden immer häufiger Cluster anstelle von Groß- oder Parallelrechnern verwendet [1]. Unter einem Cluster versteht man eine Menge von Rechnern (PCs oder Workstations), die mit einem Netzwerk verbunden sind. Cluster zeichnen sich durch ein gutes Preis-Leistungs-Verhältnis aus. Dies wird durch den Einsatz von Standard-Komponenten (*COTS - Commodity-Off-The-Shelf*) erreicht. So kann man beispielsweise von der ständig wachsenden Leistungsfähigkeit der CPU's im PC- und Workstation-Bereich profitieren, ohne kosten- und zeitintensive Entwicklungsarbeit leisten zu müssen.

In der Vergangenheit waren Cluster keine wirkliche Alternative zum Einsatz von speziellen Parallelrechnern, wie z.B. IBM SP2 [2] oder CRAY T3E [3], da die Cluster-Netzwerke aus Standardkomponenten bei weitem nicht die Leistungsfähigkeit der Netzwerke von Parallelrechnern erreichten. In den letzten Jahren wurden neue Netzwerke (z.B. Myrinet [4], ServerNet [5], Gigabit Ethernet [6], etc.), die den Unterschied in der Leistungsfähigkeit deutlich verringern, entwickelt. Das macht den Einsatz von Clustern als Alternative zu Großrechnern oder speziellen Parallelrechner attraktiver.

Die alleinige Verbesserung der Kommunikationshardware reicht aber nicht aus. Die Kommunikationssoftware, die einem Programm den Zugriff auf das Netzwerk ermöglicht, muß an die neuen Netzwerktechnologien angepaßt werden. Erste Ansätze auf diesem Gebiet wurden mit ActiveMessages [7] im Rahmen des Projektes *The Berkeley NOW (Network of Workstations)* [8] erforscht. Ziel dieser Kommunikationssoftware und solcher wie z.B. BIP [9] ist, daß Anwendungen die volle Leistungsfähigkeit des Netzwerkes ausschöpfen können. Die zwei grundlegenden Ideen dabei sind, daß Anwendungen direkt, d.h. ohne die Beteiligung des Betriebssystems, auf das Netzwerk zugreifen können und, daß die CPU von möglichst vielen Netzwerkoperation befreit wird.

Betrachtet man die Programmierung auf einem Cluster, lassen sich die beiden Modelle *Message Passing* (Nachrichtenaustausch) und *Shared Memory* (Gemeinsamer Speicher) unterscheiden. Beim *Message Passing* koordiniert der Programmierer die Aktionen der Instanzen eines Programmes (Prozesse), die auf verschiedenen Rechnerknoten ausgeführt werden, durch expliziten Nachrichtenaustausch, daß heißt, er selbst fügt Operationen für den Nachrichtenaustausch in den Quelltext ein. Wird *Shared Memory* eingesetzt, so braucht sich der Programmierer nicht selbst um den Austausch gemeinsamer Daten kümmern. Diese Aufgabe wird von spezieller Hardware oder Software übernommen.

Beide Programmiermodelle haben Vor- und Nachteile. Ein Nachteil des Modells *Message Passing* ist die aufwendige Programmierung. Da der Programmierer den Nachrichtenaustausch selbst koordinieren muß, liegt hier aber auch ein großes Optimierungspotential. In der Regel führt dies zu effizienteren Anwendungen als bei Einsatz des *Shared Memory*-Modells. Programmbibliotheken wie MPI [10] und PVM [11] erleichtern das Programmieren im *Message Passing*-Modell. Ein wichtiger Vorteil des anderen Modells ist die Einfachheit der Programmierung. Die Konsistenz gemeinsamer Daten wird durch die Hardware oder Software gewährleistet. Daher besitzt der Programmierer aber auch weniger Optimierungsmöglichkeiten, was im allgemeinen zu geringeren Leistungswerten führt. In der Vergangenheit wurde das *Shared Memory*-Modell hauptsächlich durch spezielle Hardware [12] realisiert. Die Entwicklung dieser Hardware ist aufwendig. Dadurch besitzen solche Systeme ein schlechteres Preis-Leistungs-Verhältnis. Hinzu kommt, daß speziell entwickelte Hardware schlecht skalierbar ist.

Eine Alternative sind Systeme, die das *Shared Memory*-Modell durch Software realisieren. Diese Software wird auch als DSM-System (*Distributed Shared Memory*-System; Verteiltes System mit virtuellem gemeinsamem Speicher) bezeichnet. Da diese Systeme auf dem Austausch von Nachrichten beruhen, wirken sich die geringe Bandbreite und die hohe Latenzzeit der verwendeten Kommunikationsarchitektur nachteilig auf die Leistungsfähigkeit aus. Eine Kommunikationsarchitektur besteht aus einem Netzwerk und einer Software, die ein Kommunikationsprotokoll implementiert. Möglichkeiten, die hohen Latenzzeiten zu tolerieren, sind der Einsatz von Multi-Threading (MT) und die Verbesserung der Responsivität eines DSM-Systems. Multi-Threading ermöglicht die Überlappung von Kommunikation und Berechnungen innerhalb eines Prozesses. Ziel der Verbesserung der Responsivität ist es, daß wichtige Nachrichten so schnell wie möglich bearbeitet werden.

In den letzten Jahren wurden beachtliche Fortschritte in der Netzwerktechnologie gemacht. Die vorliegende Arbeit beschäftigt sich damit, die gesteigerte Leistungsfähigkeit von Netzwerken in einem *Shared Memory*-System zu nutzen. Das zugrundegelegte *Shared Memory*-System ist DSM-Threads. Der Name DSM-Threads steht für *Distributed Shared Memory-Threads*. Es handelt sich um ein System, das gemeinsamen Speicher in einem Netzwerk von Rechnern, d.h. deren Speicher auf die einzelnen Knoten verteilt ist, simuliert. Es ist in der Programmiersprache *C* implementiert und unterstützt Multi-Threading nach dem POSIX-Thread-Standard 1003.1c [13].

DSM-Threads gewährleistet die Konsistenz gemeinsamer Daten durch nachrichtenbasierte Protokolle. Die ursprüngliche Implementierung von DSM-Threads verwendet das Netzwerkprotokoll TCP basierend auf dem Netzwerk Ethernet. Die Kombination TCP/Ethernet zeichnet sich durch Robustheit und große Verfügbarkeit aus, hat aber auch den Nachteil einer geringen Leistungsfähigkeit. Neue Kommunikationarchitekturen können diesen Nachteil mindern oder sogar ganz aufheben. Die neuen Architekturen erfordern auch die Überarbeitung der Netzwerkprotokolle. Dies leistet im Fall von Myrinet die Bibliothek BIP (*Basic Interface for Parallelism*) [9]. In dieser Arbeit wird BIP durch die Anwendung von MADELEINE [14] für die Kommunikation eingesetzt. MADELEINE ist eine Kommunikationsbibliothek, die paralleles Rechnen und Multi-Threading unterstützt. Sie ermöglicht dem Anwender, zwischen verschieden Kommunikationsarchitekturen zu wählen, ohne den Quelltext eines Programmes zu ändern. Die Ausführungsgeschwindigkeit eines Programmes kann nicht nur durch die Wahl einer leistungsfähigeren Netzwerkarchitektur sondern auch durch das spezielle Design von Nachrichten in MADELEINE, wodurch sich Optimierungsmöglichkeiten beim Nachrichtentransfer ergeben, gesteigert werden.

Ein weiteres Ziel der Arbeit war es, der Frage, wie von der einzelnen Kommunikationshardware abstrahiert werden kann, nachzugehen. Dazu wurde die Kombination von DSM-Threads und MADELEINE studiert. MADELEINE eignet sich für diese Studie, da es von den Spezifika einzelner Kommunikationsarchitekturen abstrahiert und eine einheitliche Schnittstelle bereitstellt. Bei der Kombination sollen beide Systeme so wenig wie möglich geändert werden. Programme, die bisher das DSM-System benutzen, sollen ohne Anpassungen weiterhin funktionieren.

1.1 Allgemeine Ergebnisse

In diesem Abschnitt werden allgemeine Ergebnisse dieser Arbeit vorgestellt (siehe auch [15]). Während im Hauptteil der Arbeit an einem konkreten Beispiel erläutert wird, wie asynchrone Kommunikation auf Grundlage einer modernen Kommunikationsarchitektur mit Multi-Threading verbunden werden kann, sollen hier die Erfahrungen verallgemeinert werden. Die folgenden Kapitel können auch als Untermauerung der aufgestellten Thesen verstanden werden.

Mit der Studie wurde untersucht, welche Anforderungen ein DSM-System mit bestimmten Eigenschaften an die einzusetzende Kommunikationssoftware stellt. Zunächst werden die im Kontext der Arbeit relevanten Eigenschaften eines DSM-Systems aufgezählt. Das DSM-System verwendet bzw. erfordert:

- Multi-Threading - ermöglicht ein besseres Design von Anwendungen und die Kompensierung von Verzögerungen beim Austausch von Nachrichten,
- striktes Prioritätenscheduling,
- indirekte Kommunikation - Nachrichten werden an einer Stelle zentral empfangen und anschließend an die Adressaten verteilt (Demultiplexing),
- MT-sichere Kommunikation - Kommunikationsoperationen blockieren nur den aktuellen Thread,
- asynchrone Kommunikation - Kommunikation kann zu einem beliebigen Zeitpunkt stattfinden *und*
- Responsivität - wichtige Nachrichten müssen beim Empfänger zügig, möglichst sofort, bearbeitet werden.

Andere wichtige Eigenschaften eines DSM-Systems werden in Kapitel 2 genannt. Im folgenden werden die Erfahrungen, die bei der Kombination von DSM-Threads und MADELEINE gesammelt wurden, verallgemeinert.

Kommunikationssoftware, die effizient in DSM-Systemen mit den beschriebenen Eigenschaften eingesetzt werden soll, muß folgendes leisten:

- **Unterstützung indirekter Kommunikation**,
- **Interruptunterstützung** und
- **Trennung von verteilter Ausführung und Kommunikation**.

Die ersten beiden Punkte wirken sich besonders auf die Responsivität des Systems aus. Der letzte Punkt hat Auswirkungen auf die Einfachheit des Integrationsprozesses und die Flexibilität des Systems.

Unterstützung indirekter Kommunikation

Dieser Punkt berücksichtigt das Problem, wie Nachrichten empfangen werden. Zwei Modelle können unterschieden werden: (A) direkte Kommunikation und (B) indirekte Kommunikation. Direkte Kommunikation bedeutet, daß jeder Thread eines Prozesses Nachrichten an einem privaten Port empfängt. Unter der Voraussetzung, daß Multi-Threading mit striktem Prioritätenscheduling eingesetzt wird, kann dieses intuitive Modell die Responsivität des Systems deutlich reduzieren. Die indirekte Kommunikation, bei der alle Nachrichten eines Prozesses an einem zentralen Port, der von einem ausgezeichneten Thread überwacht wird, empfangen werden löst dieses Problem. In Abschnitt 2.2.2 wird detailliert auf dieses Problem eingegangen.

Moderne Kommunikationsarchitekturen, wie z.B. BIP/Myrinet, bauen darauf auf, daß sich Sender und Empfänger synchronisieren, wenn die Nachrichten eine bestimmte Größe übersteigen. Dies vermeidet eine zusätzliche Zwischenspeicherung von Daten beim Empfänger. Die Synchronisation verwendet ein einfaches *Handshaking*. Das bedeutet, daß der sendende Thread auch eine Bestätigung vom Empfänger bekommt. Da aber jeder Thread Nachrichten senden kann, würde dies der Forderung nach indirekter Kommunikation widersprechen, denn mehrere Threads müßten Nachrichten empfangen können. Bei der Integration von Kommunikationssoftware, die auf Bibliotheken wie z.B. BIP aufbaut, muß dieses Problem berücksichtigt werden. In der Regel werden Änderungen im Quelltext notwendig. Anstelle von BIP könnten auch Bibliotheken, die die Kommunikation à la ActiveMessages [7] oder FastMessages [16] implementieren, eingesetzt werden. Mit diesen könnte die Synchronisierung vermieden werden. Unter bestimmten Voraussetzungen wäre aber wieder eine Zwischenspeicherung notwendig. In Abschnitt 3.2.2 wird detailliert auf diese Problematik eingegangen.

Interruptunterstützung

Dieser Punkt beruht auf der Beobachtung, daß die Kommunikation asynchron und Polling nicht effizient einsetzbar ist. Polling wird in modernen Kommunikationsarchitekturen durch nicht-blockierende Testoperationen unterstützt. Die wichtige Forderung nach guter Responsivität läßt sich aber nicht in allen Systemen erfüllen. Dies gilt insbesondere für Systeme, die Multi-Threading mit striktem Prioritätenscheduling einsetzen, da die Polling-Frequenz systembedingt nie 1/10 ms übersteigt. Interrupts werden nur durch wenige Kommunikationssyteme wie z.B. LCI (Low-level Communication Interface) [17] unterstützt. In vielen Implementierungen (z.B. BIP [9] und GM [18]) wurde auf die Implementierung von Interrupts zur asynchronen Notifikation, mit der Begründung, daß Polling effizienter als Interrupts sei, verzichtet. Diese Aussage gilt aber nicht allgemein. Systeme die Multi-Threading einsetzen und auf Ansätze, wie z.B. dem Einfügen von Pollingtests in den Binärcode des ausführbaren Programms [19] verzichten wollen, sind auf Interrupts angewiesen. Interrupts benötigen nur einen Bruchteil der kleinsten möglichen Warteperiode (10 ms). Selbst wenn diese Zeitspanne deutlich reduziert werden könnte (etwa um den Faktor 100 - 1000), wären Interrupts die interessantere Alternative, da viele unnötige Kontextwechsel den geringen Vorteil eines Pollingtests überkompensieren würden.

Trennung von verteilter Ausführung und Kommunikation

Die untersuchte Kommunikationssoftware MADELEINE/BIP stellt verteilte Ausführung und Kommunikation zur Verfügung. Beide Komponenten sind eng miteinander verknüpft, insbesondere setzt die Kommunikation ein bestimmtes Modell der verteilten Ausführung voraus. Für eine einfachere Integration solcher Kommunikationssoftware in DSM-Systeme wäre eine klare Trennung dieser Komponenten wünschenswert.

Einerseits müßte das Modell der verteilten Ausführung des DSM-Systems nicht verändert werden, um die Kommunikationskomponente einzusetzen. Andererseits reduziert das Modell die Flexibilität von DSM-Systemen während der Laufzeit in Bezug auf die Menge der verfügbaren Rechnerknoten. Das Modell geht davon aus, daß zu Beginn einer Anwendung alle Prozesse auf den Knoten gestartet werden. Während der Laufzeit eines Programmes können Knoten nicht dynamisch hinzugefügt werden.

Die weitere Arbeit ist wie folgt aufgebaut. In den Kapiteln 2 und 3 werden die Systeme DSM-Threads und MADELEINE erläutert. In Kapitel 4 werden die Methodik der Kombination und das Design und die Implementierung eines Zwischenschrittes beschrieben. Das Design des kombinierten Systems wird in Kapitel 5 dargestellt. Daran schließt sich eine Erläuterung der Implementierung des neuen Systems in Kapitel 6 an. Das Kapitel 7 beschäftigt sich mit Funktionalitätstests und Leistungsmessungen. In Kapitel 8 werden verwandte Arbeiten vorgestellt. Zuletzt werden die Ergebnisse der Arbeit in Kapitel 9 zusammengefaßt.

Kapitel 2

DSM-Threads

DSM-Threads [20, 21] vereint zwei wesentliche Merkmale verteilten/parallelen Rechnens in einem System. Zum einen simuliert es gemeinsamen Speicher in einem Netzwerk von Rechnern und gewährleistet die Konsistenz der Daten. Zum anderen stellt es Mittel zur Erzeugung von entfernten Threads (Parallelität) bereit. Es stellt dem Benutzer eine Bibliothek von Pthreads-ähnlichen Funktionen zur Verfügung. Pthreads steht für den POSIX-Thread-Standard 1003.1c [13], der Threads und eine Schnittstelle zu ihrer Manipulierung definiert.

Bei der Entwicklung von DSM-Threads waren die folgenden Punkte wichtig:

- Portabilität (Realisierung als Laufzeitsystem),
- Kompatibilität zu POSIX-Threads (Pthreads),
- effiziente Kommunikation,
- Erweiterbarkeit,
- Modularität (Komponenten, Protokolle),
- Verwendung existierender Compiler und Betriebssysteme *und*
- Skalierbarkeit.

Zunächst werden die grundlegenden Eigenschaften und Konzepte von DSM-Threads dargestellt. Anschließend werden die Komponenten des Systems, die von der Kombination betroffen waren, detailliert erläutert.

2.1 Grundlegende Eigenschaften und Konzepte

DSM-Threads verbindet die verteilte Ausführung von Threads und die Gewährleistung der Konsistenz von gemeinsamen Daten in einem System. Es unterstützt Multi-Threading nach dem POSIX-Thread-Standard 1003.1c (Pthreads). Ein Thread kann auf einem beliebigen Rechner aus einer definierten Menge von Rechnern eines Netzwerks gestartet werden. Threads sind priorisiert. Es wird striktes Prioritätenscheduling eingesetzt. Das bedeutet, daß immer derjenige Thread ausgeführt wird, der von allen aktiven Threads die höchste Priorität besitzt. Die Threads eines DSM-Prozesses lassen sich aufgrund ihrer Priorität in drei Gruppen zusammenfassen.

Benutzer-Threads: Sie besitzen die niedrigste Priorität. Diese Threads führen benutzerdefinierte Funktionen aus. Der Programmierer kann während der Laufzeit neue Benutzer-Threads erzeugen.

Kommunikationsserver: Es existiert immer ein Kommunikationsserver (CS) pro Prozeß. Er hat die höchste Priorität. Der Kommunikationsserver empfängt alle Nachrichten. Da er die höchste Priorität hat, werden eintreffende Nachrichten sofort bearbeitet werden. Dies bewirkt eine hohe Responsivität des Systems.

Worker-Threads: Sie haben eine höhere Priorität als die Benutzer-Threads, aber eine geringere als der Kommunikationsserver. Ihre die Aufgabe ist, den Kommunikationsserver bei der Bearbeitung von Nachrichten zu unterstützen.

Die Konsistenz von gemeinsamen Daten kann durch den Einsatz unterschiedlicher Konsistenzprotokolle (z.B. sequentielle Konsistenz oder Eintrittskonsistenz) erreicht werden. Diese Protokolle sind nachrichtenbasiert. Die Kommunikation ist asynchron, das heißt es ist kein „Rendez-vous“ von Sender und Empfänger notwendig. Der sendende Thread wird nicht blockiert. Beim Empfangen einer Nachricht wird nur der empfangende Thread blockiert. Während jeder Thread Nachrichten senden kann, werden Nachrichten immer von einem ausgezeichneten Thread, dem Kommunikationsserver (CS) empfangen.

Der Kommunikationsserver ist der Thread mit der höchsten Priorität. Das stellt eine unverzügliche Bearbeitung einer Nachricht sicher und garantiert somit eine hohe Responsivität des Systems.

Nachrichten sind typisiert. Das Senden einer Nachricht an einen Prozeß entspricht einer Anfrage, die vom Empfänger bearbeitet wird. Der Sender erhält eine Antwort von einem Prozeß (nicht bei allen Nachrichtentypen), der auch verschieden zu dem ursprünglichen Empfänger sein kann. Die Struktur einer Nachricht wird im nächsten Abschnitt erläutert.

DSM-Threads stellt neben den üblichen Synchronisationsoperationen, wie Locks und Bedingungsvariablen, auch Barrieren bereit. Der gemeinsame Speicher kann statisch und dynamisch allokiert werden.

2.2 Komponenten

In diesem Abschnitt werden ausgewählte Komponenten von DSM-Threads vorgestellt. Die Auswahl beschränkt sich auf jene Komponenten, die von der Kombination von DSM-Threads und MADELEINE, betroffen waren. Zunächst wird die Initialisierung besprochen. Daran schließt sich die Beschreibung der Kommunikation an. Darauf folgt die Erläuterung der Erzeugung von entfernten Threads. Zuletzt wird die Terminierung beschrieben. Bei den Ausführungen werden auch Details der Implementierung genannt.

2.2.1 Initialisierung

Die Initialisierung beginnt mit dem Start eines Programmes und endet, wenn alle Systemkomponenten einsatzbereit sind. Beim Start eines Programmes wird nur ein Prozeß erzeugt. Während der Initialisierung werden Informationen der Konfigurationsdatei `.rhosts` verarbeitet. Diese Datei enthält die Namen derjenigen Rechner, auf welchen Threads gestartet werden können. Der Benutzer muß Zugriffsrechte auf diese Rechner besitzen. Die Datei muß sich im aktuellen Arbeitsverzeichnis (aktuelles Verzeichnis zum Zeitpunkt des Programmstarts) befinden.

Zuerst wird die Thread-Bibliothek von FSU-Pthreads [22] mittels `pthread_init` initialisiert. Der erste Thread eines Prozesses wird im folgenden als *Main-Thread* bezeichnet. Dieser Thread ruft die Funktion `dsm_init` mit den Parametern `argc` (Anzahl der Argumente beim Programmstart) und `argv` (Argumente beim Programmstart) auf.

Die Funktion `dsm_init` führt die Initialisierung durch. Die Initialisierung im ersten Prozeß bzw. Startprozeß - auch *Master* oder *Masterprozeß* genannt - und die Initialisierung der anderen Prozesse - auch *Slave* oder *Slaveprozeß* genannt - besitzen Gemeinsamkeiten und Unterschiede. In diesem Abschnitt werden nur die Gemeinsamkeiten (Initialisierung der Systemkomponenten) und die Initialisierung des *Masters* besprochen. Die Besonderheiten der Initialisierung eines *Slaveprozesses* werden in Abschnitt 2.2.3 erläutert.

Zunächst wird die Konfigurationsdatei verarbeitet. Die Rechnernamen werden in einer zirkulären Liste gespeichert. In Abbildung 2.1 wird die Deklaration eines Listenelementes gezeigt.

```
typedef struct dsm_rhosts {
  char *host;
  int  port;
  struct dsm_rhosts *next;
} dsm_rhosts_s;
```

Abbildung 2.1: Listenelement für die Verwaltung von rechnerbezogenen Informationen

In der Variable `host` wird der Rechnername gespeichert. Die Variable `port` speichert die Nummer des Ports, an welchem ein Prozeß ausgeführt wird, Verbindungen akzeptiert. Der Zeiger `next` dient zur einfachen Verkettung der Liste. Der `next`-Zeiger des letzten Elementes verweist auf das erste Element. Die prozeß-globale Variable `dsm_current` verweist immer auf das aktuelle Listenelement. Die prozeß-globale Variable `dsm_rhosts` verweist auf das erste Element der zirkulären Liste. Das erste Element ist jenes Element, welches zuerst in die Liste eingefügt wurde.

Nun werden die Systemkomponenten *Kommunikationsserver*, *Worker-Threads* und *Seitenverwaltung* initialisiert. Dazu wird die Funktion `dsm_protect` mit den Parametern `remote`, `host`, `port`, `father_host` und father_port aufgerufen. Der Parameter remote gibt an, ob der Prozeß der *Master* (remote == 0) oder ein *Slave* (remote == 1) ist. Die Werte der anderen Parameter werden in Abhängigkeit von remote gesetzt. Ist der Prozeß der *Master*, sind die Variablen host und father_host mit (char *) NULL gesetzt. Die Variablen port und father_port sind mit 0 belegt. Die Belegung der Variablen in einem *Slaveprozeß* wird in Abschnitt 2.2.3 angegeben. Die Funktion dsm_protect initialisiert die Kommunikationskomponenten durch Aufruf von dsm_comminit mit den Parametern remote, host und port. Zunächst wird ein Kommunikationsendpunkt, an dem Nachrichten empfangen werden, erstellt. Wird TCP als Netzwerkprotokoll eingesetzt, bedeutet dies einen Aufruf der Funktionen socket, bind und listen. Es wird versucht, den angegebenen Port mit dem Socket zu verbinden. Falls der Port bereits benutzt wird, wird der nächste freie Port verwendet. Anschließend werden ein Ringpuffer für Nachrichten angelegt und einige Worker-Threads gestartet. Zuletzt wird der Kommunikationsserver als neuer Thread mit höchster Priorität gestartet. Für alle Threads wird striktes Prioritätenscheduling verwendet. In Abbildung 2.2 sind alle Threads eines Prozesses nach der Initialisierung der Kommunikationskomponenten dargestellt.

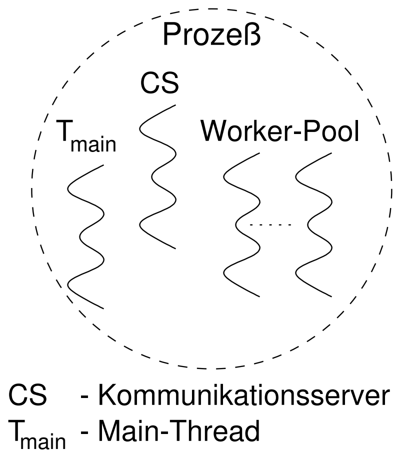

Abbildung 2.2: Threads in einem Prozeß nach der Initialisierung der Kommunikationskomponenten

Der *Main-Thread* ist T_{main}. Der Kommunikationsserver wird vom Thread CS ausgeführt. Außerdem gibt es einige Worker-Threads (default ist 5) die im Worker-Pool zusammengefaßt sind. Der *Main-Thread* hat unterschiedliche Aufgaben im *Master* und in den *Slaves*. Im *Masterprozeß* kann der *Main-Thread* alle Funktionen und Anweisungen ausführen. In einem *Slaveprozeß* werden nur die Funktionen und Anweisungen, bis einschließlich des Aufrufs von dsm_init, ausgeführt. Die Initialisierung eines *Slaveprozesses* wird in Abschnitt 2.2.3 genauer beschrieben.

Nun werden die Seitenverwaltung und mit ihr die Seitentabelle initialisiert. Für jede Seite wird der initiale Besitzer, der immer der *Masterprozeß* ist, festgelegt. Auf weitere Details der Implementierung soll hier nicht eingegangen werden, da die Seitentabelle bei der Kombination von DSM-Threads und MADELEINE eine untergeordnete Rolle gespielt hat.

Als Letztes registriert die Funktion dsm_protect eine Behandlungsroutine für Seitenzugriffsfehler. Mit Hilfe dieser Routine wird ein Nachrichtenaustausch ausgelöst, nachdem ein Seitenzugriffsfehler aufgetreten ist. Auf diese Weise kann die sequentielle Konsistenz von gemeinsamem Speicher eingehalten werden. Ein Prozeß kann Lese- und/oder Schreibrechte oder gar keine Zugriffsrechte auf eine Seite besitzen. Der *Masterprozeß* besitzt zunächst Lese- und Schreibrechte für alle Seiten. Ein *Slaveprozeß* besitzt am Anfang weder Schreibrechte noch Leserechte.

Nachdem alle Systemkomponenten initialisiert wurden, kehrt die Ausführung im *Masterprozeß* zur Hauptfunktion (main) zurück. Im *Slaveprozeß* wird die Initialisierung mit weiteren Anweisungen, die zur Erzeugung eines neuen Threads führen, fortgesetzt. Diese Anweisungen werden in Abschnitt 2.2.3 beschrieben.

2.2.2 Kommunikation

Die Kommunikation basiert auf Nachrichten. Konsistenzprotokolle und Synchronisationsoperationen werden durch den Austausch von Nachrichten realisiert. Die Implementierung zeichnet sich durch einen schichtenartigen Aufbau aus. Dies ermöglicht den Einsatz verschiedener Kommunikationsarchitekturen. Kommunikationsarchitekturen sind z.B. TCP/Ethernet und CML/SCI[1].

[1] CML steht für *Common Message Layer* [23]. SCI steht für *Scalable Coherent Interface* [24].

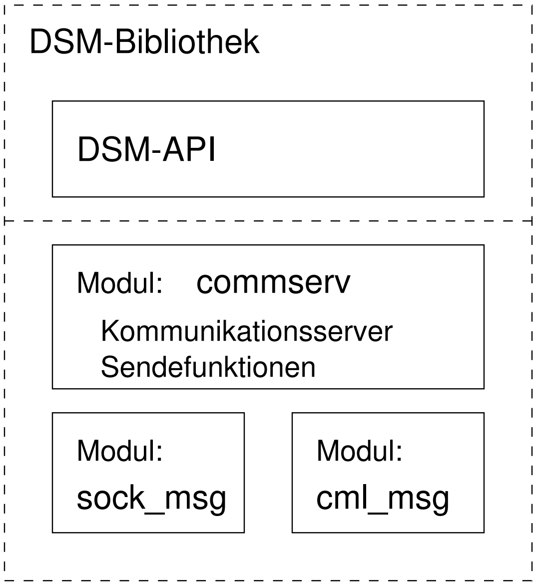

Abbildung 2.3: Modularisierung der Kommunikation in DSM-Threads

Der schichtenartige Aufbau der Kommunikationskomponenten ist in Abbildung 2.3 dargestellt. Die Kommunikationsmodule werden nur intern in DSM-Threads verwendet. Dies wird durch die oberste Schicht DSM-API, die nicht Teil der Kommunikation ist, dargestellt. Die nächste Schicht (Modul commserv) enthält Funktionen, die unabhängig vom verwendeten Netzwerkprotokoll sind. Dazu gehören der Kommunikationsserver (schließt das Empfangen von Nachrichten ein), Funktionen für den Aufbau von Verbindungen zwischen Prozessen und Funktionen für das Zusammenstellen und Senden von Nachrichten. Die Funktionen dieser Schicht greifen auf Basisfunktionen zurück, die in Abhängigkeit der verwendeten Kommunikationsarchitektur implementiert sind. In Tabelle 2.1 wird die Semantik der Basisfunktionen beschrieben. Das Modul sock_msg implementiert die Basisfunktionen für die Netzwerkprotokolle TCP und UDP. Für CML/SCI wird diese Aufgabe von dem Modul cml_msg übernommen.

Funktion	Beschreibung
`dsm_server_open`	Initialisiert einen Kommunikationsendpunkt (KEP, z.B. Port), an welchem der Kommunikationsserver Nachrichten empfängt.
`dsm_server_accept`	Wartet auf das Eintreffen einer Nachricht am KEP.
`dsm_read`	Liest einen Teil der Nachricht in den Speicher.
`dsm_client_open`	Baut eine Verbindung zu dem KEP eines anderen Prozesses auf.
`dsm_write`	Übergibt einen Teil der Nachricht an das Netzwerk.
`dsm_close`	Beendet eine Verbindung zu einem anderen Prozeß.

Tabelle 2.1: Funktionen für den internen Nachrichtenaustausch in DSM-Threads

Im folgenden werden zunächst Nachrichten - Typen und Aufbau - besprochen. Danach wird das Senden einer Nachricht erläutert. Schließlich werden das Empfangen von Nachrichten und die beteiligten Komponenten Kommunikationsserver und Worker-Threads beschrieben.

Nachrichten

Eine Nachricht in DSM-Threads - auch DSM-Nachricht genannt - ist eine endliche Folge von Bytes. Sie ist typisiert und besteht aus einem Kopf, einem Datenteil und einem Anhang. Der Typ wird durch einen Wert aus der Aufzählung, deren Definition in Abbildung 2.4 dargestellt ist, angegeben. Die einzelnen Typen sollen hier nicht besprochen werden. Die Aufzählung kann um weitere Typen ergänzt werden. Handelt es sich bei einer DSM-Nachricht um Nachricht vom Typ X, so wird diese Nachricht als X-Nachricht bezeichnet.

```
typedef enum dsm_Message {
  MSG_ACK, ACK, READ_REQ, WRITE_REQ, INVAL, READ_PAGE, WRITE_PAGE,
  COND_TOKEN, COND_WAIT, COND_SIGNAL, COND_TOKEN_BROADCAST, COND_BROADCAST,
  MUTEX_TOKEN, MUTEX_REQ, MUTEX_KILL, BARRIER_WAIT, BARRIER_RELEASE,
  BARRIER_REGISTER, BARRIER_COUNTDOWN
} dsm_Message_t;
```

Abbildung 2.4: Definition einer Aufzählung für Nachrichtentypen

Der Typ einer Nachricht wird im Kopfteil der Nachricht gespeichert. In Abbildung 2.5 ist der Kopfteil einer Nachricht mit seinen Elementen und deren Größe in Bytes dargestellt. In den Feldern ist angegeben, welche Information dort gespeichert wird. Die Werte darunter geben die Größe eines Feldes in Bytes an.

Typ	Länge	Sequenznummer	Port	Integer	Float	Seitenindex	Senderport	Sendername
1	4	4	4	4	4	4	4	MAXHOSTNAMELEN

Abbildung 2.5: Elemente des Kopfteils einer DSM-Nachricht

`Typ` speichert den Typ der Nachricht.

`Länge` speichert die Länge der gesamten Nachricht (Kopf, Daten und Anhang).

`Sequenznummer` speichert eine Sequenznummer. Diese wird verwendet, wenn UDP als Netzwerkprotokoll eingesetzt wird, um eine Flußkontrolle zu realisieren.

`Port` speichert die eigene Portnummer. Diese Information wird nur bei Verwendung von UDP benutzt.

`Integer` speichert den Integer-Wert 1. Dieses Feld und das nächste werden benutzt, um festzustellen, ob die Datenrepräsentationen auf zwei Rechnern (also zwischen Sender und Empfänger) miteinander kompatibel sind.

`Float` speichert den Fließkommawert 1.0. Zur Verwendung vgl. man die Beschreibung zu `Integer`.

`Seitenindex` speichert den Index eines Eintrages in der Seitentabelle.

`Senderport` speichert die Nummer des Ports, an welchem der Sender Verbindungen akzeptiert.

`Sendername` speichert den Rechnernamen des Senders.

An den Kopfteil schließen sich die Daten, z.B. eine Speicherseite, an. Nach dem Datenteil folgt der Anhang, der Informationen für die Heapspeicherverwaltung enthält. Eine Nachricht wird mit der Funktion `dsm_composeMsg` zusammengestellt.

Senden einer Nachricht

Eine Nachricht wird mit der Funktion `dsm_commSend` an einen anderen Prozeß gesendet. Nachrichten dürfen nicht an den eigenen Prozeß gesendet werden. Wie Nachrichten an den eigenen Prozeß „gesendet“ werden können, wird im Zusammenhang mit dem Empfangen von Nachrichten erläutert. Die Funktion erwartet neben der Nachricht (Folge von Bytes) die Angabe eines Empfängerknotens. Der Empfängerknoten wird durch die Struktur `dsm_Node_t`, deren Deklaration in Abbildung 2.6 dargestellt ist, übergeben. Die Variable `Hostname` speichert den Namen des Rechners, auf dem der Empfängerprozeß ausgeführt wird. Die Variable `Port` speichert die Nummer des Ports, an welchem der Empfänger auf Nachrichten wartet. Diese Struktur wird eingesetzt, um einen Prozeß eindeutig zu identifizieren.

Das Senden einer Nachricht ist asynchron, das heißt, es ist kein „Rendez-vous“ zwischen Sender und Empfänger notwendig. Dies gilt, wenn man die Implementierung des Nachrichtenaustausch in der DSM-Bibliothek betrachtet. Auf der Ebene der Systemfunktionen kann dennoch eine Synchronisation zwischen Sender und Empfänger notwendig sein. Die Implementierung der DSM-Bibliothek bleibt davon jedoch unberührt. Daher wird auf diese Problematik hier nicht weiter eingegangen.

Wird TCP als Netzwerkprotokoll eingesetzt, so wird für jede zu sendende Nachricht eine Verbindung zum Empfänger aufgebaut. Nachdem die Nachricht übertragen wurde, wird die Verbindung abgebrochen.

```
typedef struct dsm_Node {
  char Hostname[MAXHOSTNAMELEN];
  int  Port;
} dsm_Node_t;
```

Abbildung 2.6: Deklaration der Struktur für die Adressierung eines Empfängers

Implementierung des Sendens

Das Senden einer Nachricht wird durch die Funktion `dsm_commSend` implementiert. Diese Funktion ist Bestandteil des Moduls commserv. Sie greift auf Funktionen der untersten Schicht zu. Diese Funktionen sind in Abhängigkeit der verwendeten Kommunikationsarchitektur durch die Module sock_msg oder cml_msg implementiert. Von der tatsächlich verwendeten Kommunikationsarchitektur wird an dieser Stelle abstrahiert.

Zunächst wird mit der Funktion `dsm_client_open` eine Verbindung zum Empfänger aufgebaut. Im Anschluß wird die Nachricht mit der Funktion `dsm_WriteMsg` an den Empfänger übertragen. Nach Beendigung der Übertragung wird die Verbindung mit der Funktion `dsm_close` abgebaut.

Empfangen einer Nachricht

Dieser Abschnitt beschreibt das Empfangen einer Nachricht. Zunächst wird die Frage, ob jeder Thread eines Prozesses Nachrichten an einem privaten Kommunikationsendpunkt (Port) oder alle Threads Nachrichten an einem gemeinsamen Kommunikationsendpunkt (Port) empfangen, diskutiert. Anschließend werden die Komponenten Kommunikationsserver und Worker-Threads erläutert. Danach wird das Konzept der Behandlungsroutinen, mit denen Nachrichten unterschiedlichen Typs bearbeitet werden können, erläutert. Im Anschluß wird erläutert, wie Nachrichten an den eigenen Prozeß gesendet werden können.

Direkte und indirekte Kommunikation

Es gibt zwei Möglichkeiten, Nachrichten zu empfangen. Erstens kann jeder Thread selbst Nachrichten an einem privaten Kommunikationsendpunkt empfangen (siehe Abbildung 2.7a). Ein Kommunikationsendpunkt kann z.B. ein Port oder ein Dateideskriptor sein. Zweitens können alle Nachrichten an einem zentralen Kommunikationsendpunkt empfangen werden (siehe Abbildung 2.7b). Der tatsächliche Empfänger (Thread) wird bei dieser Variante durch den Kommunikationsserver über den Eingang der Nachricht informiert. Die erste Variante führt zu einer einfacheren Programmierung, da bei der zweiten eine zusätzliche Komponente für den Empfang der Nachrichten benötigt wird. Konzeptionell ist dennoch die zweite Variante vorzuziehen, da es bei der ersten Variante zu Blockierungen beim Empfangen von Nachrichten kommen kann.

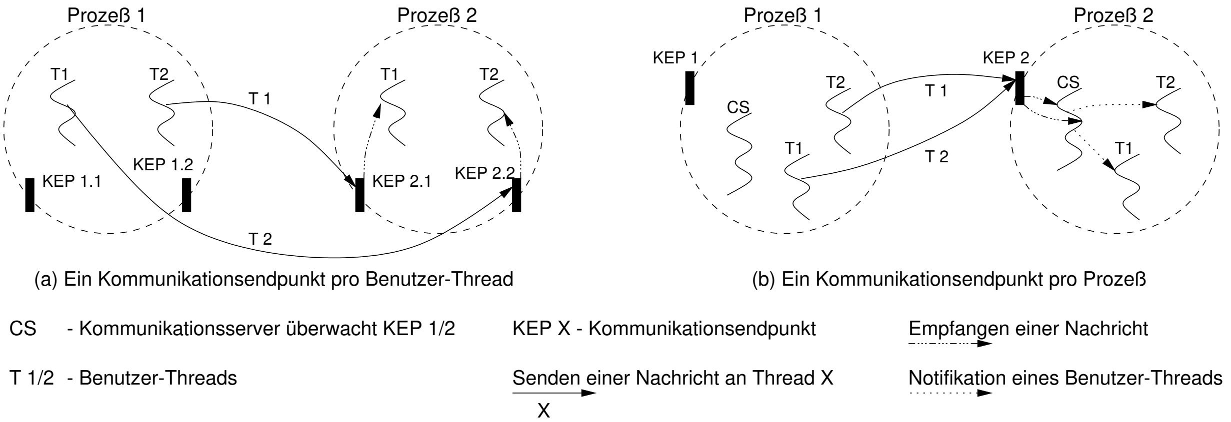

Abbildung 2.7: Empfangen von Nachrichten an privatem oder zentralem Kommunikationsendpunkt

Man stelle sich vor, daß eine Nachricht für einen Thread mit niedriger Priorität eintrifft. Nach dieser Nachricht folgt eine Nachricht für einen Thread mit höherer Priorität. Da striktes Prioritätenscheduling eingesetzt wird, kann es nun passieren, daß der höherpriorisierte Thread ausgeführt wird, da das Betriebssystem ihn über den Eingang einer neuen Nachricht informiert hat. Dennoch kann er nicht mit dem Empfangen der Nachricht beginnen, da zunächst die Nachricht für den Thread mit niedrigerer Priorität gelesen werden muß. Der Thread mit niedrigerer Priorität kann aber nicht ausgeführt werden, da es einen aktiven Thread mit höherer Priorität gibt. Eine Lösung dieses Problems wäre die Änderung der Prioritäten während der

Laufzeit. Die Priorität eines Threads, für den eine Nachricht eingetroffen ist, wird für die Dauer des Nachrichtenempfangens erhöht. Diese Möglichkeit wird in DSM-Threads nicht eingesetzt.

Der Kommunikationsserver

Der Kommunikationsserver empfängt alle Nachrichten. Er wird durch einen Thread mit höchster Priorität ausgeführt. Die Arbeitsweise des Kommunikationsservers ist in Abbildung 2.8 skizziert. Nach dem Start wartet er auf das Eintreffen einer Nachricht. Ist eine Nachricht eingetroffen, wird zunächst der Kopf der Nachricht bearbeitet. Anschließend werden Datenteil und Anhang vom Kommunikationsmedium gelesen. Nun wird die Nachricht bearbeitet. Die Bearbeitung wird von einer Behandlungsroutine vorgenommen. Für jeden Nachrichtentyp existiert eine Behandlungsroutine. Eine Behandlungsroutine kann entweder von dem Kommunikationsserver selbst oder von einem Worker-Thread ausgeführt werden. Nach der Behandlung der Nachricht - Ausführung der Routine durch den Kommunikationsserver oder Weitergabe der Nachricht an einen Worker - fährt der Kommunikationsserver fort, das heißt, er wartet auf die nächste Nachricht.

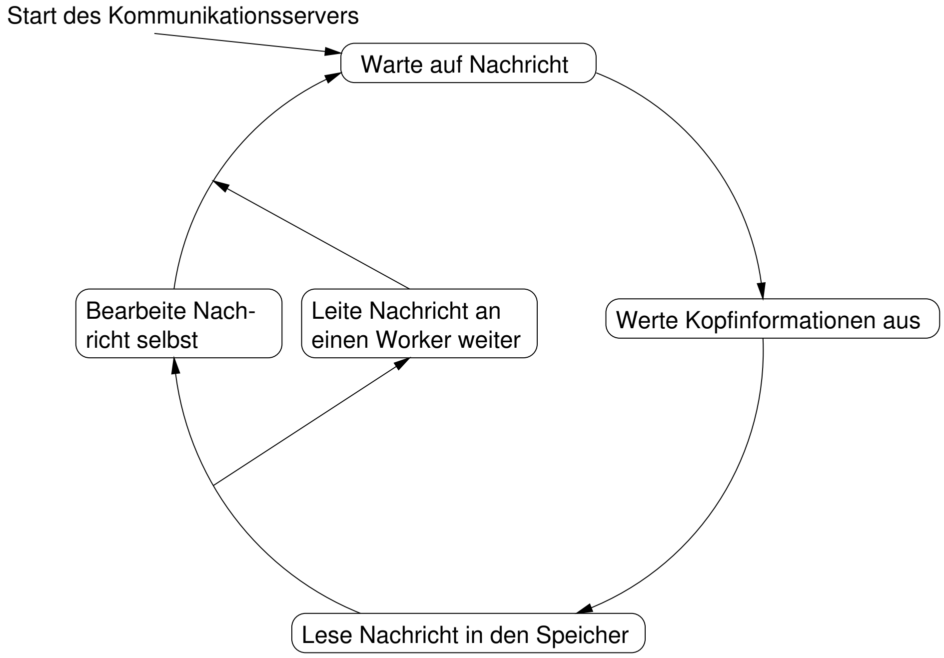

Abbildung 2.8: Arbeitsschema des Kommunikationsservers

Worker-Thread

Ein Worker-Thread (kurz Worker) ist ein Thread des DSM-Systems. Während der Initialisierung wird ein Pool an Workern gestartet. Nachdem der Kommunikationsserver eine Nachrichten empfangen hat entscheidet er, ob er die Nachricht selbst bearbeitet oder einen Worker mit der Bearbeitung beauftragt. Die Nachrichten, die durch einen Worker bearbeitet werden, werden zunächst in einem Ringpuffer zwischengespeichert. Jeder Worker wartet mit Hilfe einer Bedingungsvariable darauf, daß der Kommunikationsserver eine Nachricht in den Ringpuffer einfügt. Der Kommunikationsserver signalisiert die Worker-Threads, wenn er eine Nachricht in den Ringpuffer eingefügt hat. Ein Worker entnimmt eine Nachricht aus dem Puffer, bearbeitet sie und wartet anschließend wieder mit der Bedingungsvariable. Für den Fall, daß viele Nachrichten in den Puffer gestellt werden, existiert ein Pool an Workern. Falls sich die Bearbeitung einer Nachricht verzögert, kann ein anderer Worker die anderen Nachrichten bearbeiten.

Behandlungsroutinen für Nachrichten

Für jeden Nachrichtentyp gibt es eine Behandlungsroutine. Der Kommunikationsserver oder ein Worker ermittelt den Typ einer Nachricht und kann so die entsprechende Routine ausführen.

Diese Behandlungsroutinen erwarten einen Parameter des Typs `dsm_Msg_t`. Die Deklaration dieses Typs ist auszugsweise in Abbildung 2.9 dargestellt.

```
typedef struct dsm_Msg {
  long          PageIndex;
  dsm_Node_t    Sender;
  int           Len;
  char          *Buf;
  char          *Page;
} dsm_Msg_t;
```

Abbildung 2.9: Deklaration (auszugsweise) des Datentyps `dsm_Msg_t`

Der Kommunikationsserver erstellt eine Instanz dieses Typs, nachdem er die ganze Nachricht vom Kommunikationsmedium gelesen hat. Die Bedeutung der einzelnen Elemente des Datentyps `dsm_Msg_t` wird nun beschrieben.

`PageIndex` Index eines Eintrages in der Seitentabelle.

`Sender` Paar (Rechnername,Portnummer), das den Absender eindeutig identifiziert.

`Len` Die Länge der Nachricht.

`Buf` Die Nachricht als Folge von Bytes, wie sie durch `dsm_composeMsg` erstellt wurde.

`Page` Verweist auf den Datenteil der Nachricht.

Senden einer Nachricht an den eigenen Prozeß

Will ein Prozeß eine Nachricht an sich selbst senden, so darf er dazu nicht die Funktion `dsm_commSend` verwenden. Er hat aber die Möglichkeit, die entsprechende Behandlungsroutine direkt aufzurufen. Dazu muß er nur eine Instanz des Typs `dsm_Msg_t` erstellen und diese mit geeigneten Werten initialisieren. Ist der sendende Thread der Kommunikationsserver, muß zusätzlich berücksichtigt werden, ob die Nachricht durch den Kommunikationsserver oder einen Worker bearbeitet werden soll. Im zweiten Fall darf der Kommunikationsserver nicht die Behandlungsroutine aufrufen, sondern muß die Nachricht an einen Worker übergeben.

Anfrage-Antwort-Schema

Das Senden einer Nachricht an einen Prozeß kann auch als Anfrage an diesen Prozeß aufgefaßt werden. In den meisten Fällen - bei den meisten Nachrichtentypen - erwartet der sendende Thread eine Antwort vom Empfänger. Diese Antwort wird mit einer Nachricht übermittelt. In Abbildung 2.10 ist eine solche Abfolge von Nachrichten dargestellt. Ein Benutzer-Thread des Prozesses Y sendet eine Nachricht (Anfrage) an den Prozeß X und wartet mit Hilfe einer Bedingungsvariablen indirekt auf eine Antwort. Prozeß X sendet eine Nachricht (Antwort) zurück. Die Antwort wird vom Kommunikationsserver empfangen und zur Bearbeitung an einen Worker weitergeleitet. Die Behandlungsroutine informiert den wartenden Thread durch Signalisierung der Bedingungsvariable über den Erhalt der Antwort.

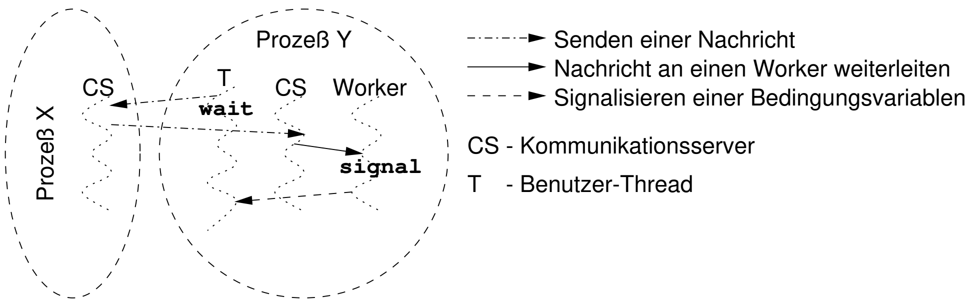

Abbildung 2.10: Anfrage-Antwort-Schema

Da nur der Kommunikationsserver Nachrichten empfängt, sind zwei Punkte zu berücksichtigen. Erstens muß der sendende Thread mit Hilfe einer lokalen Bedingungsvariable auf die Antwort warten. Die Bedingungsvariable wird durch die Behandlungsroutine, die die eingetroffene Antwort bearbeitet, signalisiert.

Zweitens darf der Kommunikationsserver keine Behandlungsroutinen ausführen, die auf eine Antwort warten. Dies würde zu einer Blockierung führen, da er nicht gleichzeitig an einer Bedingungsvariablen warten und eine Nachricht empfangen kann. Solche Behandlungsroutinen müssen immer von einem Worker ausgeführt werden.

Implementierung des Empfangens einer Nachricht

An dieser Stelle wird nur auf die Implementierung des Kommunikationsservers eingegangen. Dieser leistet die Hauptarbeit des Empfangens. Wie bei der Erläuterung der Implementierung des Sendens wird von speziellen Kommunikationsarchitekturen abstrahiert.

Der Kommunikationsserver wird durch die Funktion `dsm_commServer` realisiert. Diese Funktion ist Bestandteil des Moduls `commserv`. Während der Initialisierung wird mittels `pthread_create` ein neuer Thread erzeugt, der die Funktion ausführt.

Der Kommunikationsserver arbeitet in einer Schleife. An deren Beginn wartet er mit der Funktion `dsm_server_accept` auf eine Nachricht. Ist eine Nachricht eingetroffen, so wird der Kopf ausgewertet. Für den Datenteil und den Anhang wird ein Speicherbereich angelegt. Der Datenteil und der Anhang werden mit der Funktion `dsm_ReadMsg` in diesen Speicherbereich gelesen. Anschließend wird eine Instanz vom Typ `dsm_Msg_t` erzeugt und mit den Informationen der Nachricht initialisiert. Nun wird die Nachricht von ihrer Behandlungsroutine bearbeitet. Die Behandlungsroutine wird entweder vom Kommunikationsserver oder von einem Worker ausgeführt. Bevor der Kommunikationsserver am Anfang der Schleife weiterarbeitet, wird die temporäre Verbindung zum Sender mit `dsm_close` abgebaut.

Bei der Implementierung der Funktion `dsm_server_accept` muß beachtet werden, daß nur der Kommunikationsserver blockiert wird, solange keine Nachricht eingetroffen ist. Dies gilt insbesondere für die Unterstützung weiterer Kommunikationsarchitekturen.

2.2.3 Erzeugung entfernter Threads

Die parallele Ausführung von Programmteilen bzw. Funktionen basiert im DSM-System auf der dynamischen Erzeugung von Threads. Das System stellt eine Schnittstelle zur Erzeugung entfernter Threads bereit. Ein entfernter Thread wird nicht im selben Prozeß erzeugt. Stattdessen wird auf einem Rechner ein neuer Prozeß gestartet, der den neuen Thread erzeugt. Die verfügbaren Rechner eines Netzwerkes sind in der Datei `.rhosts` aufgeführt.

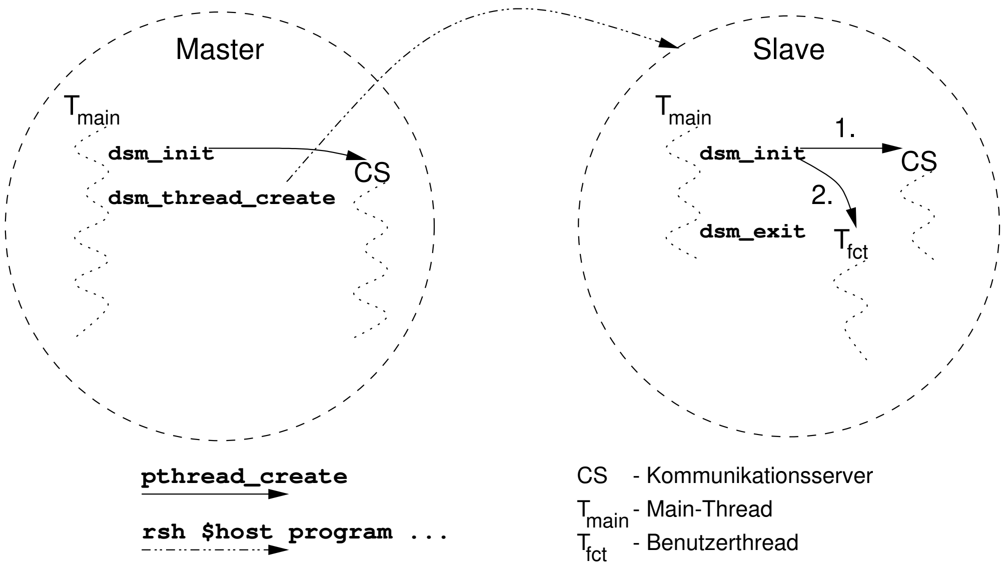

Abbildung 2.11: Erzeugung eines entfernten Threads

In Abbildung 2.11 ist Erzeugen eines entfernten Threads dargestellt. Der erzeugende Thread öffnet eine Shell (`rsh`) auf dem Zielrechner. In dieser Shell wird das Programm mit dem Namen der auszuführenden Funktion als Parameter gestartet. Die Initialisierungsroutine (`dsm_init`) erzeugt den neuen Thread.

Das DSM-System stellt die Funktion `dsm_thread_create` für die Erzeugung eines entfernten Threads bereit. Sie besitzt dieselbe Schnittstelle wie die Pthreads-Funktion `pthread_create`. Wie bereits angedeutet unterscheiden sich beide Funktionen in ihrer Semantik. Während mit `pthread_create` ein neuer Thread im selben Prozeß erzeugt wird, erzeugt `dsm_thread_create` zunächst einen neuen Prozeß, in dem der Thread erzeugt wird. Da der neue Prozeß auch auf einem anderen entfernten Rechner gestartet werden kann, spricht man auch von der Erzeugung eines entfernten Threads.

Im folgenden wird detailliert auf die Implementierung der Funktionen `dsm_thread_create` und `dsm_init` (Teil der die Erzeugung eines entfernten Threads betrifft) eingegangen.

Implementierung von `dsm_thread_create`

Die Funktion ist wie folgt deklariert.

```
int dsm_thread_create(dsm_thread_t *thread, dsm_threadattr_t *attr,
                      void *(*func)(void*), void *arg)
```

Die Variable `thread` verweist nach erfolgreichem Funktionsende auf eine Identifikationsstruktur für den erzeugten Thread. Mit `attr` können Eigenschaften des zu erzeugenden Threads, z.B. der Name des Rechners auf dem der Thread erzeugt werden soll, angegeben werden. Mit dem Parameter `func` wird die auszuführende Funktion angegeben. Der Parameter `arg` kann Argumente für die Funktion aufnehmen.

Nach einigen Vorbereitungen wird eine Shell auf dem Zielrechner geöffnet. In dieser Shell wird dasselbe Programm mit zusätzlichen Parametern ausgeführt. Sei

```
> programm arg1 ...  argN
```

das Kommando, mit dem der *Masterprozeß* gestartet wurde. Dann entspricht der Start eines *Slaveprozesses* dem Kommando

```
> rsh host programm {DSM} param1 ... paramM arg1 ... argN
```

Nun werden die Vorbereitungen im Detail erläutert. Zunächst werden die zusätzlichen Parameter generiert. Anhand der Adresse der auszuführenden Funktion (Parameter `func`) wird der Name dieser Funktion ermittelt. Dazu wird die Funktion `dsm_bfd_find`, die auf Basis der BFD Bibliothek[2] implementiert ist, verwendet. Neben der Portnummer des eigenen Kommunikationsendpunktes wird auch die Portnummer des Startprozesses in die Parameterliste übernommen. Desweiteren werden die Argumente `thread`, `attr` und `arg` in geeignete Formate übersetzt.

Der Name des Rechners, auf welchem der Prozeß gestartet werden soll, kann auf unterschiedliche Art und Weise festgestellt werden. Der Name kann explizit als Attribut (Parameter `attr`) angegeben werden. Wurde der Name nicht angegeben, ermittelt das System einen Rechner, auf dem Prozesse gestartet werden können. Dazu wird die zirkuläre Liste verwendet, die während der Initialisierung durch Auswertung der Datei `.rhosts` erstellt wurde. Die Auswahl aus dieser Liste erfolgt nach dem Round-Robin-Prinzip.

Zuletzt wird das Programm gestartet, indem mittels Systemfunktion `fork` ein neuer Prozeß generiert wird und die Shell auf dem Zielrechner durch Aufruf der Funktion `execlp` geöffnet wird.

In der Shell wird dasselbe Programm mit zusätzlichen Parametern ausgeführt. Die Ausführung des Programmes beginnt mit dem ersten Teil der Initialisierung, der in Abschnitt 2.2.1 erläutert wurde. Am Ende der Initialisierung wird der neue Thread erzeugt. Dies wird im nächsten Abschnitt erläutert.

Implementierung von `dsm_init`

Die Implementierung von `dsm_init` wurde bereits in Abschnitt 2.2.1 beschrieben. Hier werden nur Unterschiede zur Initialisierung des Startprozesses und die Erzeugung des neuen Threads erläutert, d.h., die hier beschriebene Initialisierung wird in allen *Slaveprozessen* durchgeführt.

Zunächst werden die Werte der Variablen `host`, `port`, `father_host` und `father_port` anhand der zusätzlichen Programmparametern gesetzt. Die Variablen `host` und `port` identifizieren den *Masterprozeß*. Der erzeugende Prozeß wird mit dem Paar (`father_host`,`father_port`) angegeben. Diese Variablen werden für die Initialisierung der Seitenverwaltung verwendet.

Nachdem die Systemkomponenten initialisiert wurden, wird nun der neue Thread erzeugt. Dazu wird zunächst die Adresse der auszuführenden Funktion mit Hilfe der Funktion `dsm_bfd_find` und des Funktionsnamens ermittelt. Anschließend werden die Argumente `thread`, `attr` und `arg` in ihre ursprünglichen Formate überführt.

Schließlich wird der neue Thread durch Aufruf der Funktion `pthread_create` erzeugt. Danach wartet die Routine mittels `pthread_join` auf die Beendigung des Threads. Nachdem der Thread beendet wurde, ruft sie die Funktion `dsm_exit` auf und kehrt nicht in die Hauptroutine (`main`) zurück.

[2] BFD steht für *Binary File Descriptor*.

2.2.4 Terminierung

Bevor ein Prozeß tatsächlich beendet wird, müssen sich alle Prozesse synchronisieren. Dadurch wird verhindert, daß durch das Beenden eines einzelnen Prozesses Daten verloren gehen. Die Synchronisation wird durch eine Baumbarriere realisiert. Die Baumstruktur wird während der Erzeugung entfernter Threads aufgebaut. Sie entspricht den Verwandtschaftsbeziehungen zwischen den Prozessen. Der *Masterprozeß* ist die Wurzel des Baumes. Alle Prozesse, die von einem Prozeß erzeugt wurden, befinden sich in der nächsttieferen Ebene. Diese Prozesse werden auch Kinder genannt, während der erzeugende Prozeß als Vater bezeichnet wird. Die Blätter sind Prozesse, die keine entfernten Threads erzeugt haben.

Die Barriere wird durch den Austausch von Nachrichten implementiert. Erreicht ein Prozeß die Barriere, so überprüft er, ob er Kinder hat, die die Barriere noch nicht erreicht haben. Ist dies der Fall, wartet er solange bis seine Kinder die Barriere erreicht haben. Haben alle Kinder die Barriere erreicht oder hatte der Prozeß gar keine Kinder sendet er eine Nachricht an seinen Vater und wartet auf eine Antwort.

Der Vater erhält die Terminierungsnachricht und merkt sich, daß das Kind (der Absender der Nachricht) und alle Kindeskinder die Barriere erreicht haben. Dies setzt sich fort, bis die Wurzel von allen Kindern eine solche Nachricht erhalten hat. Wenn alle Prozesse die Barriere erreicht haben, sendet der *Master* (Wurzel) eine Antwort an seine Kinder. Die Kinder verteilen diese Antwort weiter an ihre Kinder u.s.w. . Nachdem ein Prozeß die Antwort an seine Kinder weitergeleitet hat, ist die Barriere für ihn beendet.

Implementierung der Baumbarriere

Die Baumbarriere ist Teil der Funktion `dsm_exit`. In Abbildung 2.12 ist der Ablauf einer Barriere anhand eines Beispiels dargestellt.

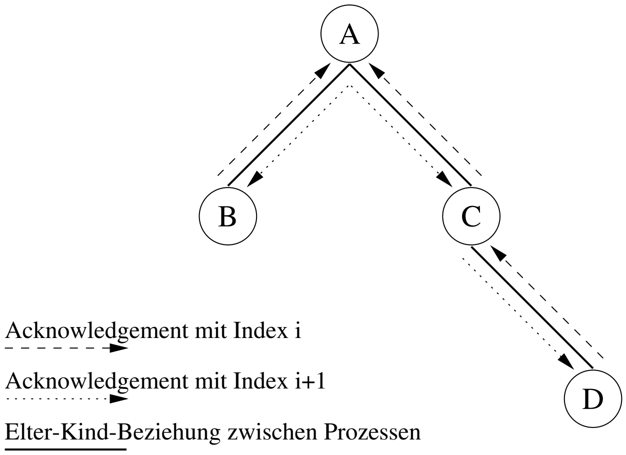

Abbildung 2.12: Synchronisation aller Prozesse während der Terminierung mittels Baumbarriere

Der Prozeß A hat die Kinder B und C und besitzt selbst keinen Vater. Prozeß B hat A als Vater und keine Kinder. Prozeß C hat ebenfalls A als Vater und D als Kind. Der Prozeß D hat keine Kinder und C als Vater. Diese Struktur ist in den Verwaltungsdaten einer speziellen Seite mit Index i abgelegt. Neben dieser wird noch eine weitere Seite mit dem Index $i + 1$ für die Terminierung verwendet.

Ein Prozeß verwaltet in den Informationen zu den Seiten i und $i+1$ jeweils eine Liste mit seinen Kindern. Zu Beginn enthält die Liste der Seite i alle Kinder, während die Liste der Seite $i+1$ leer ist. Außerdem wird der Vaterprozeß in den Informationen zu den Seiten gespeichert. Erhält der Prozeß eine Terminierungsbenachrichtigung (Nachricht vom Typ ACK, auch Acknowledgement genannt) von einem Kind (z.B. Prozeß C von Prozeß D), wird es aus der Liste der Seite i entfernt und in die Liste der Seite $i + 1$ eingefügt. Dies leistet die Behandlungsroutine für den Nachrichtentyp ACK. Gelangt ein Prozeß an die Terminierungsbarriere (durch Aufruf von `dsm_exit`), wartet er an einer Bedingungsvariablen, solange bis die Liste seiner Kinder (Seite i) leer ist. Anschließend überprüft er, ob er selbst einen Vaterprozeß besitzt. In diesem Fall, sendet er Acknowledgement an seinen Vaterprozeß (z.B. Prozeß C an Prozeß A) und wartet an einer Bedingungsvariablen der Seite $i + 1$. Besitzt er keinen Vaterprozeß, d.h. er ist die Wurzel der Baumstruktur (Prozeß A), dann schickt er an alle Kinder (Prozesse B und C) ein Acknowledgement für die Seite $i + 1$. Die Kinder werden aus der Liste der Seite $i + 1$ ermittelt. Empfängt ein Kind dieses Acknowledgement, so reicht es dieses an seine eigenen Kinder weiter (Prozeß C an Prozeß D).

2.3 Anwendungsbeispiel

Dieser Abschnitt zeigt mit einem kurzen Beispiel, wie die DSM-Bibliothek verwendet werden kann. Der Startprozeß gibt zunächst die Zeichenkette `Hello World!` aus. Nachdem ein entfernter Thread erzeugt wurde, wird der Startprozeß kurz suspendiert. Schließlich wird der Inhalt einer gemeinsamen Variablen ausgegeben. Der entfernte Thread gibt den Inhalt einer gemeinsamen Variablen aus und ändert danach ihren Wert.

Das Beispiel zeigt das Erzeugen von Parallelität und die transparente Herstellung der Konsistenz gemeinsamer Daten. Im unteren Teil der Abbildung 2.13 ist der kommentierte Quelltext des Beispiels abgebildet.

Im oberen Teil der Abbildung 2.13 sind zwei mögliche Abläufe angegeben. Sie unterscheiden sich in der zweiten Ausgabe der gemeinsamen Variable x durch den *Masterprozesses* (`weide:1277`). Im ersten Beispiel ist der Wert der Variable 15, da der entfernte Thread (`birke:1344`) den Wert noch nicht geändert hat. Im zweiten Beispiel hat der entfernte Thread den Wert der Variable geändert, bevor der *Masterprozeß* sie zum zweiten Mal ausgibt. Diese unterschiedlichen Abläufe können durch unterschiedlich lange Pausen des *Masterprozesses* entstehen (`sleep(random()%10)`).

1. Ausgabe

```
weide:1277 - Hello World!
weide:1277 - x=15 (1)
weide:1277 - x=15 (2)
birke:1344 - x=15
```

2. Ausgabe

```
weide:1277 - Hello World!
weide:1277 - x=15 (1)
birke:1344 - x=15
weide:1277 - x=45 (2)
```

```c
#include <dsm.h>
```

Stellt Schnittstelle zur DSM-Bibliothek bereit.

```c
char *welcome = "Hello World!";

DSM_DATA_BEGIN
  int x = 15;
  dsm_thread_t thread = {};
  dsm_threadattr_t attr = {};
```

Beginn der Deklaration von gemeinsamen Variablen
Initialwert der Variable x ist 15.
Die Variablen thread und attr werden für die Erzeugung eines entfernten Threads benötigt.

```c
DSM_DATA_END
```

Ende der Deklaration von gemeinsamen Variablen

```c
void RemoteThread(void *arg) {
  printf("%s:%d - x=%d\n",
    dsm_selfhost(),dsm_selfport(),x);
```

Funktion, die der entfernte Thread ausführen wird.
Ausgabe der gemeinsamen Variable x. Da der Thread bisher keine Zugriffsrechte für die Variable x besaß, beantragt das DSM-System Leserechte und ermittelt den aktuellen Wert.

```c
  x = 45;
}
```

An dieser Stelle beantragt das DSM-System Schreibrechte für die Variable x.
Nach Beendigung des Threads die Funktion dsm_exit durch die Initialisierungsroutine aufgerufen.

```c
main(int argc, char *argv[]) {
```

Funktion, die der Masterprozeß ausfhrt.

```c
  printf("%s:%d - %s\n",dsm_selfhost(),
         dsm_selfport(), welcome);
```

Ausgabe einer globalen nicht-gemeinsamen Variable.

```c
  pthread_init();
```

Initialisierung der Pthread-Bibliothek

```c
  dsm_init(argc,argv);
```

Initialisierung der DSM-Bibliothek

```c
  printf("%s:%d - x=%d (1)\n",
    dsm_selfhost(),dsm_selfport(),x);
```

Erste Ausgabe der gemeinsamen Variablen

```c
  dsm_thread_create(thread,attr,
                    RemoteThread,NULL);
```

Erzeugung eines entfernten Threads auf einem Rechner, der durch das DSM-System ausgewählt wird.

```c
  sleep(random()%10);
```

Thread wird für 0-9 Sekunden suspendiert.

```c
  printf("%s:%d - x=%d (2)\n",
    dsm_selfhost(),dsm_selfport(),x);
```

Zweite Ausgabe der gemeinsamen Variable. Falls der entfernte Thread zwischenzeitlich Schreibrechte erangt hat, beantragt das DSM-System Leserechte und ermittelt den aktuellen Wert der Variable.

```c
  dsm_exit();
}
```

Leitet die Terminierung des DSM-Systems ein (Baumbarriere).

Abbildung 2.13: Anwendungsbeispiel für das DSM-System

Kapitel 3

Madeleine

Madeleine [14] ist ein Kommunikationspaket, das den Austausch von Nachrichten in einem Netzwerk unterstützt. Es definiert eine einheitliche Schnittstelle für verschiedene Netzwerkprotokolle. So kann ein Programm ohne Änderungen des Quelltextes auf verschiedenen Kommunikationsarchitekturen eingesetzt werden. Es muß lediglich neu übersetzt werden, wenn eine andere Kommunikationsarchitektur eingesetzt werden soll. Es stellt Funktionen zur Initialisierung, zum Zusammenstellen und Senden von Nachrichten sowie zum Empfangen und Aufteilen von Nachrichten und zur Terminierung bereit. Nachrichten können auf unterschiedliche Art und Weise übertragen werden. Dies kann der Programmierer ausnutzen, um den Kommunikationsaufwand einer Anwendung zu verringern. Madeleine unterstützt bedingt Multi-Threading.

Im folgenden wird zunächst das Design von Madeleine beschrieben. Anschließend wird die Implementierung erläutert. Von den zahlreichen Kommunikationsarchitekturen, die Madeleine unterstützt, werden TCP/Ethernet und BIP/Myrinet bei den Erläuterungen berücksichtigt, da nur sie bei der Kombination von DSM-Threads und Madeleine eine Rolle gespielt haben. Zuletzt soll ein kleines Beispiel die Benutzung von Madeleine illustrieren.

3.1 Design von Madeleine

Dieser Abschnitt beschreibt das Design von Madeleine. Nach einer kurzen Einleitung folgt die Besprechung der Punkte Start/Initialisierung, Nachrichten, Senden und Empfangen.

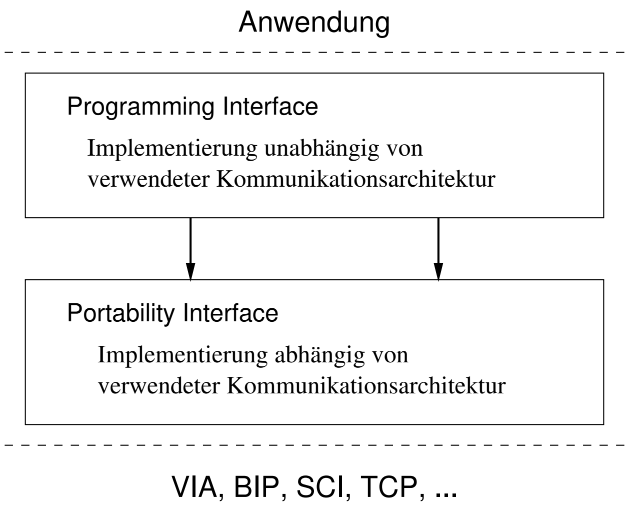

Abbildung 3.1: Schichtenaufbau von Madeleine

Die Madeleine-Bibliothek ist in zwei Schichten aufgebaut. In Abbildung 3.1 ist der Schichtenaufbau dargestellt. Die Schicht *Programming Interface* (Programmierschnittstelle) dient dem Programmierer als Schnittstelle. Sie stellt netzwerk- und protokollunabhängige Funktionen bereit. Die Schicht *Portability Interface* stellt Basisfunktionen für die obere Schicht bereit. Ihre Implementierung ist von der gewählten Kommunikationsarchitektur abhängig.

3.1.1 Start und Initialisierung

Vor dem Starten eines Programmes muß der Anwender festlegen, auf welchen Rechnern Prozesse gestartet werden sollen. Während der Start- und Initialisierungsphase werden gemäß der Festlegung des Anwenders die Prozesse auf den Rechnern gestartet. Der genaue Zeitpunkt des Startens der Prozesse hängt von der Wahl der Kommunikationsarchitektur ab. Dies wird in Abschnitt 3.2 näher erläutert. Nachdem alle Prozesse gestartet wurden, werden ihnen Identifikationsnummern zugewiesen. Zuletzt bauen die Prozesse Kommunikationsverbindungen untereinander auf.

3.1.2 Nachrichten

In MADELEINE ist eine Nachricht eine endliche Folge von Bytes. Sie wird im folgenden auch als MADELEINE-Nachricht bezeichnet. Eine MADELEINE-Nachricht ist aus Vektoren aufgebaut. Ist eine Nachricht aus n Vektoren aufgebaut, so werden die Vektoren von 0 bis $n-1$ nummeriert. Der Vektor 0 besitzt eine feste Länge. In ihm werden Kopfinformationen aber auch Daten abgelegt. Die Länge der anderen Vektoren ist nicht beschränkt.

Nachrichten können aus beliebig vielen Teilstücken zusammengesetzt sein. Teilstücke werden entweder im *Header* (Vektor 0) oder in einem anderen Vektor übertragen. Die Plazierung der Daten im *Header* oder in einem anderen Vektor kann der Benutzer durch Angabe eines Übertragungsmodus für ein Teilstück angeben. Im folgenden werden die Übertragungsmodi erläutert.

MAD_IN_HEADER erzwingt, daß MADELEINE die Daten in den *Header* der Nachricht kopiert. Das garantiert, daß die Daten mit dem Eintreffen des Vektors 0 der Nachricht beim Empfänger verfügbar sind.

MAD_IN_PLACE gibt an, daß MADELEINE die Daten ohne Kopien transportieren soll. Die Daten werden erst beim Senden direkt aus dem Speicher gelesen. Das bedeutet, daß jede Änderung der Daten zwischen dem Einpacken der Daten und dem Senden der Nachricht den Inhalt der Nachricht verändert. Bei der Angabe des Teilstückes werden nur seine Adresse und die Anzahl der Werte des Typs gespeichert.

MAD_BY_COPY kann benutzt werden, wenn die Daten nach dem Einpacken verändert werden sollen, aber diese Änderungen nicht den Inhalt der Nachricht beeinflussen dürfen. Der Unterschied zu MAD_IN_HEADER besteht darin, daß die Daten nur dann in den *Header* kopiert werden, wenn dieser noch genügend Aufnahmekapazität besitzt. Andernfalls werden die Daten in einen Vektor, dessen Speicher dynamisch angelegt wird, kopiert.

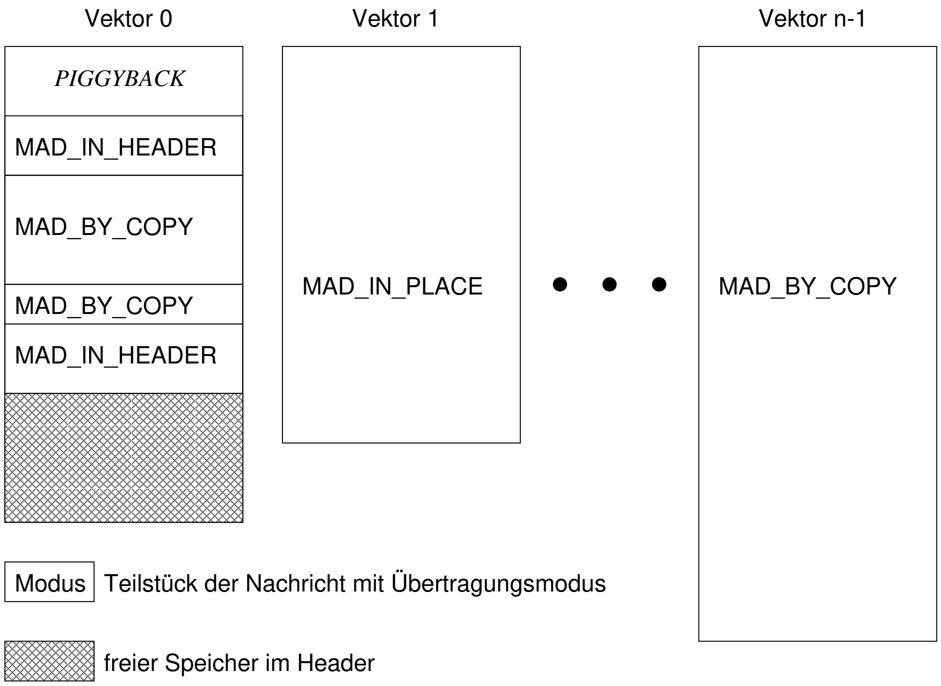

Abbildung 3.2: Aufbau einer MADELEINE-Nachricht

In Abbildung 3.2 ist der Aufbau einer Nachricht dargestellt. Der Vektor 0 speichert den *Header* und besitzt eine feste Länge. An seinem Anfang befindet sich der *piggyback*-Bereich. Hier wird die Gesamtlänge aller Vektoren gespeichert. Der Rest des Vektors 0 steht für Teilstücke der Nachricht, die mit einem der Modi MAD_IN_HEADER oder MAD_BY_COPY übertragen werden, zur Verfügung. Der Speicherplatz im *Header* muß nicht ausgeschöpft werden.

Die Vektoren 1 bis $n-1$ nehmen jeweils ein Teilstück der Nachricht auf. Ihre Länge ist unbeschränkt. Die Teilstücke werden entweder mit dem Modus MAD_BY_COPY oder MAD_IN_PLACE übertragen. Ein Teilstück, das mit dem Modus MAD_BY_COPY übertragen wird, wird in einem Vektor gespeichert, wenn der verfügbare Speicher des *Headers* zu klein ist.

3.1.3 Senden einer Nachricht

Das Senden einer Nachricht erfordert die folgenden drei Schritte.

1. Vorbereiten eines Sendepuffers
2. Zusammenstellen der Nachricht aus einem oder mehreren Teilstücken
3. Senden der Nachricht

Der Empfänger wird im ersten Schritt durch seine Identifikationsnummer angegeben. Ein Prozeß darf zu einem Zeitpunkt nur eine Nachricht senden. Dies muß nur beim dritten Schritt, dem eigentlichen Senden, berücksichtigt werden. Falls Multi-Threading eingesetzt wird, es also mehrere Threads geben kann, die zum gleichen Zeitpunkt eine Nachricht senden wollen, stellt MADELEINE den wechselseitigen Ausschluß sicher.

3.1.4 Empfangen einer Nachricht

Wie das Senden erfolgt das Empfangen einer Nachricht in drei Schritten.

1. Warten auf eine Nachricht und Empfangen des *Headers* (Vektor 0)
2. Benennung der Teilstücke einer Nachricht
3. Empfangen der Vektoren 1 bis $n-1$

Beim Empfangen wird nicht zwischen verschiedenen Sendern unterschieden. Benötigt der Empfänger die Information, welcher Prozeß die Nachricht gesendet hat, muß der Sender diese Information, z.B. seine Identifikationsnummer, in einem Teilstück der Nachricht übertragen.

Die Teilstücke, die im zweiten Schritt benannt werden, müssen in Größe und Übertragungsmodus mit den Teilstücken, die beim Zusammenstellen der Nachricht angegeben wurden, übereinstimmen. Zusätzlich muß die Reihenfolge der Benennung beim Empfangen und beim Senden übereinstimmen. Das ist notwendig, da in der Nachricht selbst keine Informationen über ihren Aufbau gespeichert werden.

Ein Prozeß darf immer nur eine Empfangsoperation, bestehend aus allen drei Schritten, ausführen. MADELEINE schützt diesen kritischen Bereich nicht. Können mehrere Threads gleichzeitig eine Empfangsoperation ausführen, muß der Programmierer den wechselseitigen Ausschluß sicherstellen.

3.2 Implementierung von Madeleine

MADELEINE ist in zwei Schichten aufgebaut (vgl. Abbildung 3.1). Dem Benutzer stehen die Funktionen des *Programming Interface* (Programmierschnittstelle) zur Verfügung. Diese Funktionen sind auf Basis von netzwerkspezifischen Modulen der Schicht *Portability Interface* implementiert. Dabei greifen die Funktionen der Programmierschnittstelle auf Funktionen zurück, die für jede Kommunikationsarchitektur in einem separaten Modul implementiert sind. Zunächst wird die Implementierung der Schicht *Programming Interface* erläutert. Anschließend wird im Detail auf die netzwerkspezifischen Implementierungen der Schicht *Portability Interface* eingegangen, wobei der Schwerpunkt auf dem Kommunikationspaket BIP liegt. BIP wird in einem eigenen Abschnitt vorgestellt.

3.2.1 Programmierschnittstelle

In diesem Abschnitt wird die Implementierung der Funktionen `mad_init`, `mad_sendbuf_init`, `mad_pack_`*type*, `mad_sendbuf_send`, `mad_receive`, `mad_unpack_`*type*, `mad_recvbuf_receive` und `mad_exit` beschrieben.

`void mad_init(int *argc, char **argv, int nbp, int *tids, int *nb, int *me)`

Diese Funktion ruft die netzwerkspezifische Initialisierungsfunktion auf. Mit den Parametern `argc` und `argv` werden die Argumente des Programmstarts übergeben. `nbp` gibt die Anzahl der Prozesse an. Wird an dieser Stelle die Konstante `NETWORK_ASK_USER` verwendet, wird der Benutzer aufgefordert, die Anzahl der Prozesse einzugeben. In dem Feld `tids` werden die Identifikationsnummern der Prozesse gespeichert. Der Parameter `nb` dient zur Rückgabe der Anzahl der Prozesse. In `me` wird die eigene Identifikationsnummer zurückgegeben.

`void mad_sendbuf_init(int dest_node)`

Diese Funktion implementiert den ersten Schritt des Sendens einer Nachricht. Zu Beginn wird ein Puffer, der zur internen Darstellung einer Nachricht dient, aus einem Pool ausgewählt. Anschließend wird der Puffer in einen Initialzustand versetzt. Die Identifikationsnummer des Empfängers ((dest_node) wird im Puffer gespeichert.

`void mad_pack_`***type***`(madeleine_part where, char *data, size_t nb)`

Diese Funktionen fügen der Nachricht ein Teilstück, das beginnend an der Adresse `data` aus einer Anzahl `nb` von Elementen des Typs *type* besteht, hinzu. Die Funktion `mad_pack_byte` ist die Basisfunktion für alle anderen Typen. Es existieren Funktionen für elementare Basistypen, wie z.B. `float`, `double` oder `int`. Neben den Angaben über die Daten (Speicheradresse und Anzahl der Elemente eines Typs) ist der Übertragungsmodus `where` ein Parameter der Funktion. Die Basisfunktion `mad_pack_byte` wird nun für jeden Übertragungsmodus beschrieben.

MAD_IN_HEADER Zunächst wird getestet, ob noch genügend Platz im *Header* der Nachricht vorhanden ist. Ist dies der Fall, werden die Daten dorthin kopiert. Der Zähler für die Länge des *Headers* wird um die Anzahl der kopierten Bytes inkrementiert. Danach wird der Zähler noch auf die nächstgrößere 32-Bit-Zahl erhöht. Reicht die Kapazität des *Headers* nicht für die Aufnahme der Daten aus, wird eine Fehlermeldung generiert.

MAD_IN_PLACE Zunächst wird überprüft, ob noch ein Vektor zur Verfügung steht. Wenn dies nicht der Fall ist, wird eine Fehlermeldung generiert. Andernfalls werden die Speicheradresse und die Anzahl der Bytes des Teilstücks im nächsten unbenutzten Vektor notiert. Anschließend wird vermerkt, daß der Speicher für die Daten, deren Adresse im Vektor angegeben ist, nicht freigegeben werden muß, da er beim Aufruf der Einpackfunktion nicht dynamisch angelegt wurde. Zuletzt wird der Zähler für die Anzahl der benutzten Vektoren inkrementiert.

MAD_BY_COPY Zu Beginn wird überprüft, ob die Daten noch Platz im *Header* haben. Ist dies der Fall, wird wie bei dem Modus MAD_IN_HEADER verfahren. Andernfalls werden die Daten in einen Vektor kopiert. Dazu wird ein entsprechend großer Speicherbereich angelegt. Die Daten werden in diesen Speicherbereich kopiert. Die Größe des Bereiches wird im Vektor notiert. Außerdem wird vermerkt, daß der Speicherbereich dynamisch angelegt wurde und nach der Benutzung, daß heißt nach dem Absenden der Nachricht, wieder freigegeben werden muß. Zuletzt wird der Zähler für die Anzahl der benutzten Vektoren inkrementiert. Werden bereits alle Vektoren benutzt, wird eine Fehlermeldung generiert.

Die Funktionen dieser Gruppe arbeiten unabhängig von der Kommunikationsarchitektur.

`void mad_sendbuf_send(void)`

Diese Funktion überträgt zunächst den *Header* aus dem Puffer in den Vektor 0. Anschließend werden alle Vektoren an die netzwerkspezifische Sendefunktion übergeben. Nachdem alle Vektoren gesendet wurden, werden die Speicherbereiche, die während des Zusammenstellens der Nachricht dynamisch angelegt wurden, freigegeben.

`void mad_receive(void)`

Diese Funktion ruft die netzwerkspezifische Methode zum Empfangen des *Headers* auf. Anschließend wird die Anzahl der bereits verarbeiteten Bytes des *Headers* auf die nächstgrößere 32-Bit-Zahl aufgerundet und in einer Variable gespeichert. Die Anzahl der zu lesenden Vektoren wird mit 0 initialisiert. Diese Daten (*Header*, Anzahl der gelesenen Bytes und Vektorenanzahl) werden in globalen Variablen der Schicht Programmierschnittstelle gespeichert. Daher darf zu jedem Zeitpunkt höchstens ein Thread eine Empfangsoperation durchführen, da diese Daten sonst überschrieben werden könnten.

`void mad_unpack_`***type***`(madeleine_part where, char *data, size_t nb)`

Diese Funktionen beschreiben den Aufbau der Nachricht auf der Empfängerseite. Mit jedem Aufruf einer Funktion dieser Gruppe wird ein Teilstück angegeben. Die Funktion `mad_unpack_byte` ist die Basisfunktion für alle anderen Typen (z.B. `float`, `double` oder `int`). Neben den Angaben über die Daten (Speicheradresse `data` und Anzahl `nb` der Elemente eines Typs *type*) ist der Übertragungsmodus `where` ein Parameter der Funktion. Die Basisfunktion `mad_unpack_byte` wird nun für jeden Übertragungsmodus erläutert.

MAD_IN_HEADER Die Daten werden aus dem *Header* an die angegebene Speicheradresse kopiert. Der Zähler für die verarbeiteten Bytes des *Headers* wird entsprechend inkrementiert und an der nächstgrößeren 32-Bit-Zahl ausgerichtet.

MAD_IN_PLACE Die Speicheradresse und die Größe des Teilstücks werden im aktuellen Vektor notiert. Anschließend wird der nächste Vektor zum aktuellen. Die Daten werden erst an ihre Stelle kopiert, wenn sie durch die Funktion `mad_recvbuf_receive` empfangen wurden.

MAD_BY_COPY Zunächst wird anhand der Anzahl bereits verarbeiteten Bytes des *Headers* und der zu entpackenden Bytes ermittelt, ob das Teilstück im *Header* oder in einem Vektor übermittelt wurde. Im ersten Fall wird wie bei dem Modus MAD_IN_HEADER verfahren. Andernfalls wird wie bei MAD_IN_PLACE verfahren.

Die Funktionen dieser Gruppe arbeiten unabhängig von der Kommunikationsarchitektur.

`void mad_recvbuf_receive(void)`

Die Funktion ruft die netzwerkspezifische Funktion, die die Vektoren 1 bis $n-1$ der Nachricht empfängt, auf. Der netzwerkspezifischen Funktion werden Informationen über den Aufbau der Nachricht (Adresse des ersten Vektors und Anzahl der Vektoren) als Parameter übergeben.

`void mad_exit(void)`

Diese Funktion ruft die parameterlose netzwerkspezifische Terminierungsfunktion auf.

3.2.2 Portabilitätsschicht

In diesem Abschnitt werden ausgewählte Funktionen der Schicht *Portability Interface* erläutert. Zunächst wird auf die Implementierung für die Unterstützung von TCP/IP eingegangen. Anschließend wird die Kommunikationsbibliothek BIP beschrieben. Zuletzt wird die Implementierung auf Basis von BIP erläutert.

Implementierung für TCP/IP

Dieser Abschnitt erläutert Funktionen der Schicht *Portability Interface*, die für das Protokoll TCP/IP implementiert sind. Im einzelnen sind dies die Funktionen `mad_tcp_network_init`, `mad_tcp_network_send`, `mad_tcp_network_receive`, `mad_tcp_network_receive_data`.

`void mad_tcp_network_init(int *argc, char **argv, int nbp, int *tids, int *nb, int *me)`

Diese Funktion implementiert die Initialisierung der TCP-basierten Komponenten von MADELEINE. Zuerst wird beschrieben, zu welchem Zeitpunkt die Funktion aufgerufen wird. Anschließend wird die Initialisierung erläutert.

Das Startskript `madload` erzeugt nur den *Masterprozeß*. In diesem wird zu Beginn die Funktion `mad_init`, die schließlich die hier beschriebene Funktion aufruft, ausgeführt. Zu diesem Zeitpunkt existiert also lediglich ein Prozeß auf dem Knoten, dessen Name als erster in der Konfigurationsdatei `.madconf` steht. Alle weiteren Prozesse werden durch `mad_tcp_network_init` gestartet. Neben dem Starten aller Prozesse werden während der Initialisierung Kommunikationsverbindungen zwischen allen Prozessen aufgebaut.

Zunächst überprüft die Initialisierungsroutine anhand der Umgebungsvariable `MAD_MY_NUMBER`, ob der aktuelle Prozeß der *Master* oder ein *Slave* ist. Im Falle des *Masters* ist diese Variable nicht gesetzt. Ist der Prozeß ein *Slave*, so steht in dieser Variable die Identifikationsnummer des Prozesses. Der *Master* ermittelt die Anzahl der verfügbaren Knoten (entspricht der Anzahl der Wörter in der Konfigurationsdatei minus 1). Anschließend wird die Anzahl der zu startenden Prozesse ermittelt. Diese kann als Parameter `nbp` der Initialisierungsfunktion angegeben oder vom Benutzer erfragt werden. Sie kann sich von der Anzahl der Knoten unterscheiden. Ist die Anzahl der Prozesse größer als 1, wird die Funktion `master` mit der Anzahl der Prozesse als Parameter aufgerufen. Der *Slave* ermittelt seine Identifikationsnummer und die Anzahl der Knoten anhand von Umgebungsvariablen. Anschließend ruft er die Funktion `slave` mit dem Namen des *Masters* und einer Portnummer auf.

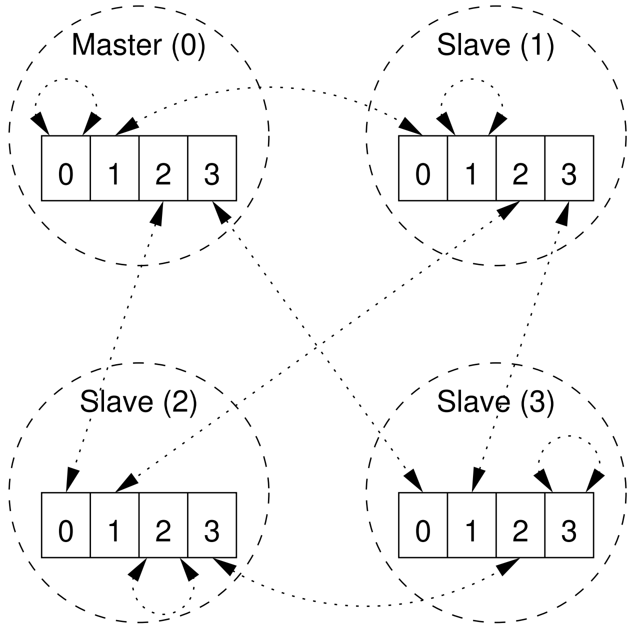

Abbildung 3.3: Kommunikationsverbindungen zwischen den Prozessen

Die Funktion `master` erstellt einen Kommunikationsendpunkt und startet $n-1$ Prozesse. Das Starten der Prozesse leistet die Funktion `spawn_procs`, die ihrerseits einige Skripte benutzt. In den Skripten werden die benötigten Umgebungsvariablen gesetzt. Auf diese Skripte soll hier nicht weiter eingegangen werden. Nach dem Starten der anderen Prozesse werden die Kommunikationsverbindungen initialisiert. Die Kommunikationsstruktur ist ein vollständiger Graph von Duplexverbindungen. In Abbildung 3.3 ist die Kommunikationsstruktur für 4 Prozesse dargestellt. Die Funktion `slave` erstellt einen Kommunikationsendpunkt und leistet den Aufbau der Kommunikationsverbindungen auf der Seite der *Slaves*.

Der Parameter `tids` dient zur Rückgabe der Identifikationsnummern. Die Prozesse werden entsprechend der Reihenfolge, in welcher die Rechnernamen in der Datei `.madconf` aufgeführt sind, nummeriert. Die Anzahl der gestarteten Prozesse wird in `*nb` gespeichert. Die eigene Identifikationsnummer wird mit `me` zurückgegeben.

`void mad_tcp_network_send(int dest_node, struct iovec *vector, size_t count`

Diese Funktion sendet eine Nachricht, die aus mehreren Vektoren bestehen kann, an einen Prozeß. Die Identifikationsnummer des Empfängers `dest_node`, die Adresse des Vektors 0 `vector` und die Anzahl der Vektoren `count` werden als Parameter übergeben. Zu Beginn wird mit einem Lock sichergestellt, daß kein anderer Thread gleichzeitig eine Nachricht an den Empfänger sendet. Dadurch wird garantiert, daß die Vektoren einer Nachricht nicht mit Vektoren einer anderen vermischt werden. Anschließend werden die Vektoren der Nachricht in aufsteigender Reihenfolge übertragen. Dafür wird die Funktion `writev` verwendet, daß heißt die ganze Nachricht kann mit einem Funktionsaufruf übertragen werden. Andernfalls wird die Nachricht inkrementell durch mehrmaliges Aufrufen von `writev` gesendet. Nachdem alle Vektoren übertragen wurden, wird der Lock wieder freigegeben.

`void mad_tcp_network_receive(char **head)`

Diese Funktion wartet auf das Eintreffen einer Nachricht von einem Prozeß[1]. Dazu wird die Funktion `select` verwendet. Ist eine Nachricht eingetroffen, wird der *Header* bzw. der Vektor 0 mit der Funktion `readv` in einen (statischen) Puffer `*head` gelesen. Die Identifikationsnummer des sendenden Prozesses wird in einer globalen Variable gespeichert. Sie darf bis zur Beendigung der Funktion `mad_tcp_network_receive_data` nicht erneut gesetzt werden. Daher ist das Empfangen einer Nachricht ein kritischer Bereich. Für dessen Schutz muß der Anwender selbst sorgen, da dies in MADELEINE nicht getan wird.

`void mad_tcp_network_receive_data(struct iovec *vector, size_t count)`

Diese Funktion liest die Vektoren 1 bis $n-1$ der Nachricht. Der Funktion werden Angaben über

[1] Der sendende Prozeß kann auch derselbe sein. Für die Kommunikation mit dem eigenen Prozeß wird eine *Pipe* verwendet.

die Teilstücke (Zieladresse im Speicher und Größe), die in Vektoren (Adresse des Vektors 1 `vector` und Anzahl der Vektoren `count`) gespeichert sind, übergeben. Die Nachricht wird durch die Funktion `readv` gelesen. So können mehrere Vektoren mit einem Aufruf bearbeitet werden. Falls nicht alle Vektoren mit einem Aufruf bearbeitet wurden, wird die Nachricht durch mehrmaliges Aufrufen von `readv` inkrementell gelesen.

Basic Interface for Parallelism (BIP)

BIP [9] ist eine Bibliothek, die ein einfaches Protokoll für das Netzwerk Myrinet implementiert. Grundlage dieser Arbeit ist die Version 0.98a. BIP stellt Funktionen zum Senden und Empfangen von Nachrichten bereit. Parallele Anwendungen werden durch Skripte, die die Konfiguration eines virtuellen Parallelrechners und das Starten von Prozessen auf den Knoten dieses Rechners ermöglichen, unterstützt.

Vor dem Starten einer parallelen Anwendung muß der Benutzer die Rechner, auf welchen ein Prozeß gestartet werden soll, mit Hilfe des Skriptes `bipconf` angeben. Anschließend kann eine Anwendung mit Hilfe des Skriptes `bipload` gestartet werden. Dieses Skript startet alle Prozesse auf den angegebenen Knoten. Dabei ist zu berücksichtigen, daß die verwendete Version von BIP genau einen Prozeß pro spezifiziertem Knoten erlaubt. Jeder Prozeß wird mit denselben Parametern und derselben Umgebung gestartet.

Zu Beginn führt jeder Prozeß die Funktion `bip_init` aus, um die Bibliothek zu initialisieren. Jedem Prozeß wird eine Identifikationsnummer zugeordnet. Für das Senden und Empfangen stehen blockierende und nicht-blockierende Aufrufe zur Verfügung. Hier sollen nur die nicht-blockierenden Funktionen erläutert werden.

Der Anwender von BIP muß zwischen kurzen und langen Nachrichten unterscheiden. Werden lange Nachrichten ausgetauscht, müssen die Aufrufe zum Senden und Empfangen eine Rendez-vous-Semantik implementieren. Der Aufruf der Empfangsoperation muß vor dem korrespondierendem Aufruf der Sendeoperation gestartet werden. Für kurze Nachrichten hält jeder Knoten einige Puffer bereit, so daß das Senden nicht blockiert, falls kein Aufruf zum Empfangen gestartet wurde. Dies gilt nur solange freie Puffer auf der Empfängerseite vorhanden sind. Die Größe kurzer Nachrichten wird durch `BIPSMALLSIZE` (254 Wörter à 4 Bytes) festgelegt.

Jede Nachricht wird an einen bestimmten Kanal eines Empfängers gesendet. Ein Kanal wird durch ein TAG identifiziert. Dieses muß sowohl beim Senden als auch beim Empfangen angegeben werden.

Zu einem Zeitpunkt kann nur eine Sendeoperation und je eine Empfangsoperation pro Kanal ausgeführt werden. Diese Operationen können in verschiedenen Threads ausgeführt werden und sind *thread-safe*. Allerdings muß beachtet werden, daß blockierende Aufrufe nicht automatisch einen Kontextwechsel hervorrufen.

Das nicht-blockierende Senden erfordert zwei Schritte. Zuerst wird die Sendeoperation mit der Funktion `bip_tisend(int dest, int tag, int *buf, int len)` eingeleitet. Der Empfänger wird mit `dest` angegeben. Der Kanal, an den die Nachricht geschickt werden soll, wird mit `tag` übergeben. Die Parameter `buf` und `len` geben den Inhalt der Nachricht an. Die Beendigung der Funktion garantiert nicht, daß die Nachricht bereits übertragen wurde. Der Rückgabewert der Funktion dient als *Handle*, das im zweiten Schritt benötigt wird. Im zweiten Schritt wird mit der Funktion `bip_test(int handle)` überprüft, ob die Sendeoperation beendet ist. Ist das Senden der Nachricht abgeschlossen, so kehrt sie mit einem Wert ungleich -1 zurück. Der zweite Schritt muß gegebenenfalls wiederholt werden, um auf den Abschluß der Sendeoperation zu warten. Beide Aufrufe sind nicht-blockierend. Die Methode des wiederholten Testens wird auch als *Polling* bezeichnet.

Das nicht-blockierende Empfangen erfordert ebenfalls zwei Schritte. Im ersten Schritt wird die Empfangsoperation mit der Funktion `bip_tirecv(int tag, int *buf, int len)` eingeleitet. Dabei bezeichnet `tag` den Kanal von dem eine Nachricht empfangen werden soll. Mit dem Parameter `buf` wird der Speicherbereich, in den die Nachricht gelesen werden soll, angegeben. Dessen Größe wird durch `len` spezifiziert. Der Rückgabewert ist ein *Handle*, das im zweiten Schritt Verwendung findet. Mit der Funktion `bip_rtestx(int handle, int *node)` überprüft, ob die Empfangsoperation beendet ist. Der Parameter `handle` ist der Rückgabewert des ersten Schrittes. Der zweite Parameter `node` dient zur Aufnahme eines Rückgabewertes. Dort wird die Identifikationsnummer des sendenden Prozesses gespeichert. Wenn die Nachricht vollständig empfangen wurde, gibt die Funktion die Anzahl der empfangenen Wörter zurück. Andernfalls kehrt sie sofort mit dem Wert -1 zurück. Beide Funktionen sind nicht-blockierend. Der zweite Schritt muß gegebenenfalls mehrfach wiederholt werden. Beim Empfangen muß also auch *Polling* eingesetzt werden.

BIP unterstützt nicht die Notifikation durch Signale für I/O-Ereignisse, z.B. Eintreffen einer Nachricht. Die Auswirkungen dieser Einschränkung von BIP bei der Unterstützung von Multi-Threading werden in Abschnitt 6.2.2 diskutiert.

Wird ein Programm beendet, so wird automatisch der während der Initialisierung registrierte *Exit-Handler* `bip_exit` aufgerufen.

Implementierung für BIP

Dieser Abschnitt erläutert Funktionen der Schicht *Portability Interface*, die für die Verwendung von BIP implementiert sind. Im einzelnen sind dies die Funktionen `mad_bip_network_init`, `mad_bip_network_send`, `mad_bip_network_receive` und `mad_bip_network_receive_data`.

`void mad_bip_network_init(int argc, char **argv, int nbp, int *tids, int *nb, int *me)`

Diese Funktion ruft die Initialisierungsfunktion von BIP `bip_init` auf und initialisiert interne Zustandsvariablen, z.B. für den *Credits*-Mechanismus, der später in diesem Abschnitt beschrieben wird.

`void mad_bip_network_send(int dest_node, struct iovec *vector, size_t count)`

Die Parameter dieser Funktion sind der Empfänger (`dest_node`) und die zu sendenden Vektoren der Nachricht (Adresse des Vektor 0 `vector` und Anzahl der Vektoren `count`). Zu Beginn wird mit einen Lock verhindert, daß andere Threads parallel eine Nachricht an den Empfänger senden.

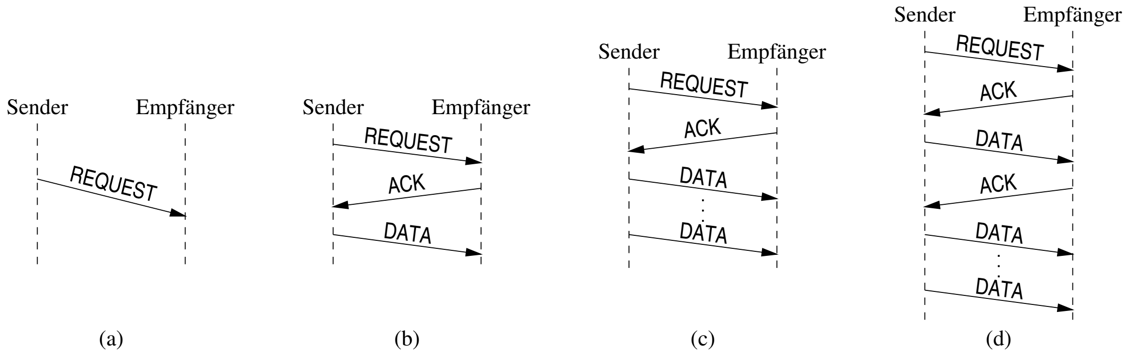

Abbildung 3.4: Schemata für das Senden einer Nachricht in Madeleine auf Basis von BIP

Die Sendeprozedur arbeitet in Abhängigkeit der Länge und Anzahl der Vektoren einer Madeleine-Nachricht. In Abbildung 3.4 ist für jeden Fall ein Nachrichtenschema dargestellt. Die möglichen Fälle sind: (a) die Nachricht besteht nur aus einem Vektor und ist kurz, (b) die Nachricht besteht aus einem langen Vektor, (c) die Nachricht besteht aus mehreren Vektoren, deren Nullter kurz ist und (d) die Nachricht besteht aus mehreren Vektoren, deren Nullter lang ist. Die Unterscheidung zwischen kurzen und langen Vektoren ist wichtig, da ein Empfänger statisch angelegte Puffer für die Aufnahme von kurzen Nachrichten bereithält (vgl. Erläuterungen zu BIP in diesem Kapitel). Nachrichten werden immer an einen bestimmten Kanal gesendet. Der Typ einer Nachricht bezeichnet den Kanal, an welchen die Nachricht geschickt wird. Die Implementierung benutzt die Typen REQUEST, ACK, DATA und CREDITS. In der Abbildung steht der Typ oberhalb des Pfeiles, der das Senden einer Nachricht symbolisiert. Im folgenden Text wird der Typ in Klammern angegeben.

Ein kurzer Vektor kann in einer Nachricht (REQUEST) gesendet werden, und der Empfänger muß noch nicht zum Empfangen bereit sein (Abb. 3.4a). Wird ein langer Vektor gesendet, muß der Empfänger die Empfangsoperation bereits gestartet haben. Für das Empfangen ist ein ausreichend dimensionierter Puffer notwendig. Da lange Vektoren beliebig groß werden können, muß zunächst die Länge dem Empfänger mitgeteilt werden. Daraus ergibt sich das Senden des Vektors in drei Schritten (Abb. 3.4b). Zunächst wird ein Teil des Vektors, der in einen Puffer auf der Empfängerseite paßt, gesendet (REQUEST). Der Empfänger bestätigt diese Nachricht und signalisiert die Bereitschaft zum Empfangen (ACK) des zweiten Teils. Der Sender übermittelt nach Erhalten der Bestätigung den zweiten Teil des Vektors (DATA). Besteht die Nachricht aus mehreren Vektoren, so wartet der Sender nach dem Senden des Vektors 0 auf eine Bestätigung (ACK). Anschließend wird jeder Vektor in einer eigenen Nachricht (DATA), deren Länge beliebig sein kann, übertragen. Der Empfänger sendet nach Erhalt des Vektors 0 eine Bestätigung (ACK) und führt für jeden Vektor eine Empfangsoperation aus (DATA). Dies ist in den Abbildungen 3.4c und 3.4d dargestellt. Nach dem Senden aller Vektoren wird der Lock freigegeben.

Für kurze Nachrichten wird ein Puffermanagement durchgeführt. Ein Prozeß besitzt je einen *Credit* für einen freien Puffer bei einem anderen Prozeß. Da jeder Prozeß zehn Puffer für jeden Sender bereithält, kann ein Sender maximal zehn *Credits* pro anderem Prozeß besitzen. In Abbildung 3.5 ist das Puffermanagement dargestellt. Der Initialzustand für zwei Prozesse ist im Bild 3.5a zu sehen. Der linke Stapel in einem Prozeß steht für die Anzahl der *Credits*. Außerdem zählt ein Prozeß die Anzahl der kurzen Nachrichten, die er erhält. Diese Zahl wird durch den rechten Stapel gezeigt.

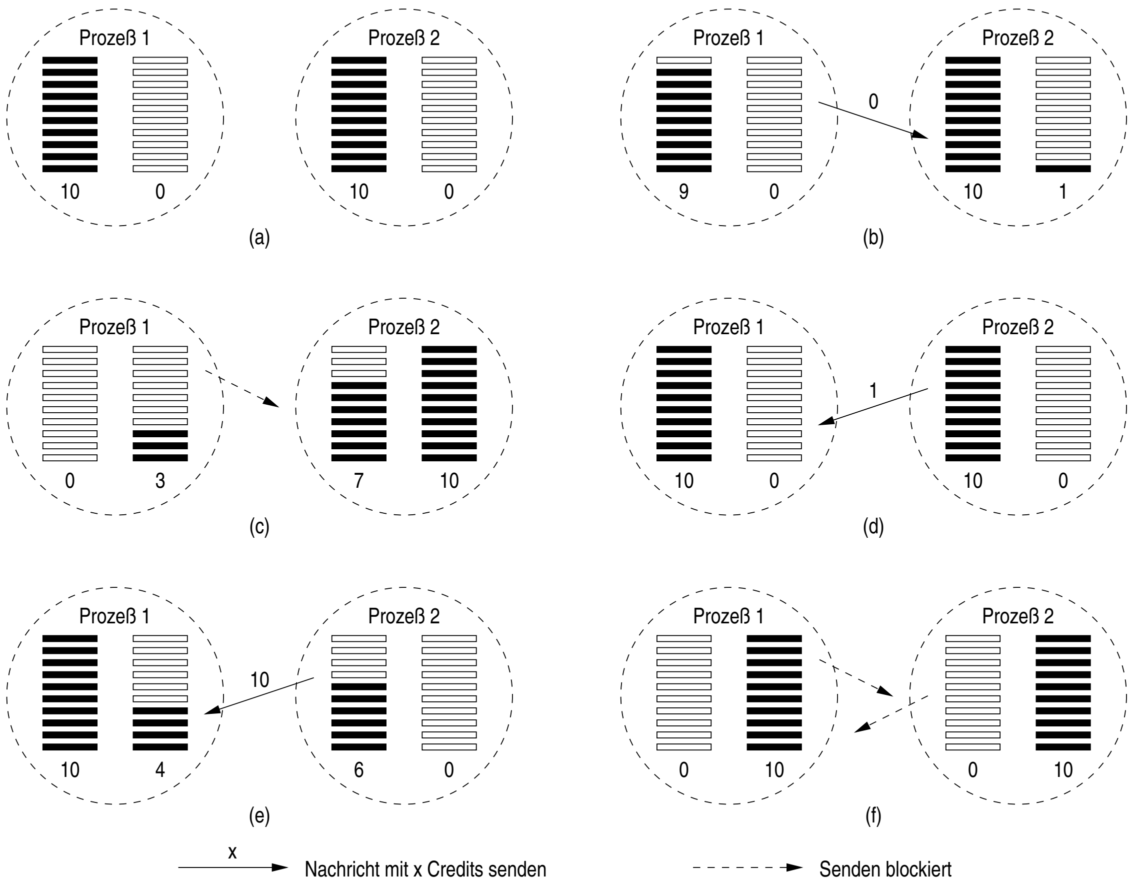

Abbildung 3.5: Puffermanagement durch Verwaltung von *Credits*

Sendet ein Prozeß eine kurze Nachricht, so wird die Anzahl der *Credits* für den Empfängerprozeß um eins dekrementiert (in Abbildung 3.5 von (a) zu (b)). Hat der Prozeß keinen *Credit* für den Empfänger, so blockiert der Sender vor dem Senden (Bild 3.5c). Gibt es mehrere Threads in dem Prozeß, so blockiert nur der sendende Thread. Die Blockierung ist beendet, wenn der Prozeß *Credits* von dem Empfänger zurückbekommen hat. Ein Prozeß sendet *Credits* zurück, wenn er selbst eine kurze Nachricht sendet (in Abbildung 3.5 von (b) zu (d)) oder der andere Prozeß keine *Credits* mehr besitzt (in Abbildung 3.5 von (c) zu (e)). Letzteres wird durch Vergleichen der Pufferanzahl mit der Anzahl der empfangenen Nachrichten festgestellt. Die Anzahl der zurückgesendeten *Credits* entspricht der Anzahl der empfangenen kurzen Nachrichten. Im ersten Wort einer kurzen Nachricht wird die Anzahl der zurückgesendeten *Credits* notiert. Nachdem *Credits* zurückgesendet wurden, wird der Zähler für die empfangenen Nachrichten auf Null zurückgesetzt.

Erhält ein Prozeß eine kurze Nachricht vom Typ REQUEST, ACK oder CREDITS, wird die Anzahl der verfügbaren *Credits* entsprechend dem übermittelten Wert (zurückgesendete *Credits*) inkrementiert (Bilder 3.5c und 3.5d).

Abschließend wird die Frage, ob ein Deadlock auftreten kann, diskutiert. O.B.d.A. gilt: Eine derartige Blockierung zwischen zwei Prozessen würde vorliegen, wenn beide Prozesse keine *Credits* mehr besitzen, aber eine kurze Nachricht senden wollen (Bild 3.5f). Diese Situation könnte nur dann eintreten, wenn beide Prozessoren alle ihre *Credits* verbrauchen (d.h. kurze Nachrichten senden), ohne zwischenzeitlich bzw. vor dem Senden der letzten kurzen Nachricht noch keine Nachricht des anderen Prozesses empfangen hat. Dann könnte in der letzten Nachricht kein *Credit* zurückgesendet werden. Für die Praxis hat diese Situation keine Bedeutung, wenn man dafür sorgt, daß ständig überprüft wird, ob neue Nachrichten eingetroffen sind und diese vorrangig bearbeitet. Allerdings muß es mindestens zwei Puffer geben, sonst ist ein Deadlock möglich.

`void mad_bip_network_receive(char **head)`

Diese Funktion wartet auf das Eintreffen einer neuen Nachricht und liest deren *Header* in einen vorbereiteten Puffer.

Das Empfangen einer BIP-Nachricht wird durch die Funktion `receive_from_network` realisiert. Diese hat die Argumente `sender`, `tag`, `buf` und `len`. An der Adresse `sender` wird der Sender der BIP-Nachricht gespeichert. Mit `tag` gibt man den Typ der erwarteten Nachricht an. Die Parameter `buf` und `len` geben die Adresse und die Größe des Puffers, in dem die Nachricht gespeichert wird, an. Die Funktion wartet auf eine Nachricht des angegebenen Typs und bearbeitet parallel eingehende Nachrichten des Typs CREDITS.

Zu Beginn der Funktion `mad_bip_network_receive` wird die Empfangsoperation mit der Funktion `bip_tirecv(tag,buf,len)` initialisiert. Der Rückgabewert der Funktion ist ein *Handle*, das beim Aufruf der Funktion `bip_rtestx(handle)` verwendet wird. Mit dieser Testfunktion wird überprüft, ob eine Nachricht eingetroffen ist. Ist dies der Fall, wird die Funktion `receive_from_network` beendet. Andernfalls arbeitet sie in einer Schleife weiter. In dieser Schleife wird mit `bip_rtestx(credits_status)` überprüft, ob zwischenzeitlich eine Nachricht vom Typ CREDITS eingetroffen ist. Eine eingetroffene CREDITS-Nachricht wird sofort bearbeitet, d.h. die *Credits* für den Sender der Nachricht werden inkrementiert. Anschließend wird mit `bip_tirecv(CREDITS,credits,3)` eine neue Empfangsoperation für eine CREDITS-Nachricht eingeleitet. Mit dem Paar (`credits`, `3`) wird ein Puffer, in dem die erwartete CREDITS-Nachricht gespeichert werden soll, angegeben. Falls keine CREDITS-Nachricht eingetroffen ist, wird getestet, ob nun die Nachricht des mit `tag` angegebenen Typs eingetroffen ist. War der Test nicht erfolgreich, wird die Kontrolle mit `marcel_givehandback`[2] für eine kurze Zeit abgegeben. Andernfalls kehrt die Funktion `receive_from_network` mit der Anzahl der empfangenen Bytes zurück.

Die Funktion `mad_bip_network_receive` ruft zu Beginn die Funktion `receive_from_network` auf, um eine REQUEST-Nachricht zu empfangen. Nachdem eine REQUEST-Nachricht eingetroffen ist, wird entsprechend dem Aufbau der MADELEINE-Nachricht fortgefahren. Hierfür sind die vier Fälle (vgl. Abbildung 3.4) (a) Nachricht mit einem kurzen Vektor, (b) Nachricht mit einem langen Vektor, (c) Nachricht bestehend aus mehreren Vektoren, deren erster kurz ist und (d) Nachricht bestehend aus mehreren Vektoren, deren erster lang ist, zu berücksichtigen.

Zuerst wird der Vektor 0 einer Nachricht betrachtet. Handelt es sich um einen kurzen Vektor (Fälle (a) und (c)), kann die Funktion `mad_bip_network_receive` beendet werden. Andernfalls (Fälle (b) und (d)) wird eine Nachricht vom Typ ACK an den Sender geschickt und mit `receive_from_network` auf eine Nachricht vom Typ DATA gewartet. Diese Nachricht enthält den zweiten Teil des Vektors 0.

`void mad_bip_network_receive_data(struct iovec *mvec, size_t count)`

Diese Funktion realisiert das Empfangen der Vektoren 1 bis $n-1$ einer Nachricht. Der Parameter `mvec` ist die Adresse des Vektors 1. Mit `count` wird die Anzahl der Vektoren angegeben. Besteht die Nachricht nur aus dem Vektor 0 (Fälle (a) und (b)), wird die Funktion sofort beendet. Ist die Nachricht aus mehreren Vektoren zusammengesetzt (Fälle (c) und (d)), wird zunächst eine Nachricht vom Typ ACK an den Sender, der durch die Funktion `mad_bip_network_receive` in einer globalen Variable gespeichert wurde, gesendet. Daher müssen die Funktionen `mad_bip_network_receive` und `mad_bip_network_receive_data` hintereinander aufgerufen werden. Beide Funktionen bilden einen kritischen Bereich. Nachdem die ACK-Nachricht gesendet wurde, ruft die Funktion für jeden Vektor einmal die Funktion `receive_from_network` mit dem Tag DATA auf.

3.3 Kritik an Madeleine

MADELEINE vereint verschiedene Kommunikationsarchitekturen unter einer Schnittstelle. Dieser Vorteil kann weiter ausgebaut werden: Es gibt mindestens zwei Verbesserungsmöglichkeiten in MADELEINE.

Erstens sind die Konzepte der verteilten Ausführung und der Kommunikation eng miteinander gekoppelt. MADELEINE kann nur verwendet werden, wenn alle Prozesse bereits zu Beginn der Programmausführung gestartet werden. Diese Einschränkung ist zumindest für TCP nicht notwendig (vgl. Erzeugung von entfernten Threads in DSM-Threads in Abschnitt 2.2.3). Hingegen setzt BIP voraus, daß alle Prozesse zu Beginn gestartet werden. Hier mußte MADELEINE an diese Einschränkung angepaßt werden.

Zweitens unterstützt MADELEINE nur bedingt Multi-Threading. Einerseits wird MARCEL für die Unterstützung von Threads angewendet. MARCEL bietet aber nicht alle Möglichkeiten, die im POSIX-Thread-Standard 1003.c festgelegt sind. Andererseits werden priorisierte Threads und der Einsatz von striktem Prioritätenscheduling nicht unterstützt, da MADELEINE die Funktion `marcel_givehandback` benutzt, um Kontextwechsel durchzuführen. An einem kleinen Beispiel soll illustriert werden, daß diese Funktion für priorisierte Threads nicht ihren Zweck erfüllt.

[2] MADELEINE verwendet MARCEL für die Unterstützung von Multi-Threading (vgl. Erläuterungen in Abschnitt 3.3).

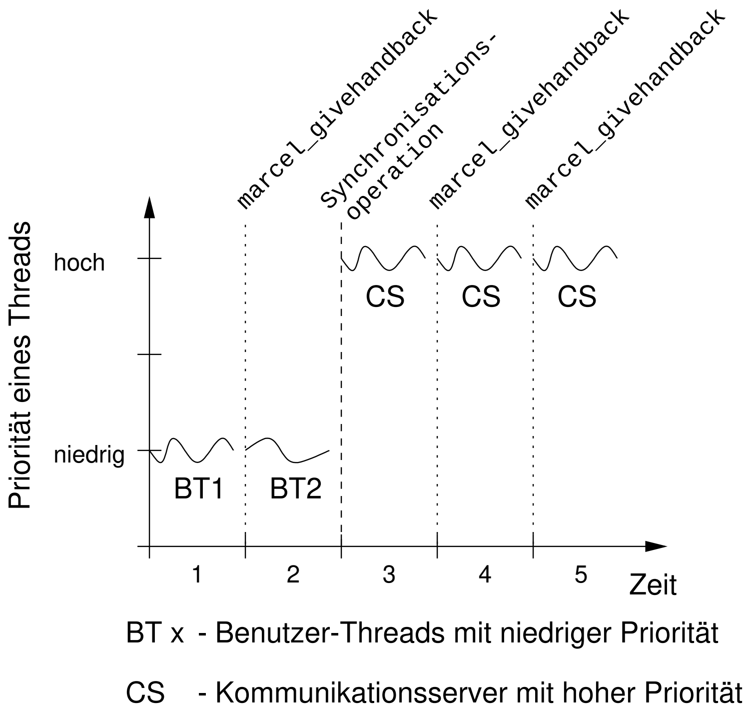

Abbildung 3.6: Auswirkungen von `marcel_givehandback` auf priorisierte Threads

In Abbildung 3.6 ist zu sehen, daß die Funktion `marcel_givehandback` nicht geeignet ist, einen Kontextwechsel herbeizuführen, wenn die Threads unterschiedliche Prioritäten haben. Es gibt zwei Benutzer-Threads BT1 und BT2 mit niedriger Priorität. Ein anderer Thread CS - der Kommunikationsserver - besitzt eine höhere Priorität. Angenommen der Kommunikationsserver könnte eine Nachricht empfangen, aber zu Beginn ist der Benutzer-Thread BT1 aktiv. Irgendwann führt BT1 einen Kontextwechsel mit `marcel_givehandback` durch. Nun ist der Benutzer-Thread BT2 aktiv. An diesem Kontextwechsel sieht man schon, daß der Kommunikationsserver nicht aktiv wird, obwohl dies sinnvoll wäre. Schließlich führt der Benutzer-Thread eine Synchronisationsoperation aus. Dies führt dazu, daß der Kommunikationsserver aktiv wird. Führt dieser nun die Funktion `marcel_givehandback` aus, um einen Kontextwechsel zu einem Benutzer-Thread durchzuführen, erhält er wieder die Kontrolle. Das liegt daran, daß die Funktion `marcel_givehandback` nur Kontextwechsel unter Threads mit gleicher Priorität durchführt. Die einzige Möglichkeit, daß ein Benutzer-Thread die Kontrolle bekommt, ist der Aufruf einer Synchronisationsoperation wie z.B. `cond_wait`.

Das wirkt sich nachteilig auf die Responsivität des Systems aus. Der Nachteil wird deutlich, wenn man bedenkt, daß der Kommunikationsserver *Polling* verwenden muß, um festzustellen, ob eine Nachricht eingetroffen ist. Zwischen zwei Pollingtests kann er nur mit Synchronisationsoperationen die Kontrolle an die Benutzer-Threads abgeben. Die Zeit zwischen zwei Tests liegt unter Linux (Kernel 2.2.12) bei etwa 10 ms. So läßt sich eine Polling-Frequenz von etwa 100 Hz erreichen. Wie man höhere Polling-Frequenzen erreichen kann, wird in Abschnitt 6.2.2 beschrieben.

3.4 Anwendungsbeispiel

In diesem Abschnitt wird die Programmierung mit MADELEINE anhand eines kurzen Beispiels gezeigt. Die Anwendung besteht aus zwei Prozessen. Der *Masterprozeß* öffnet eine große Datei und sendet sie an den zweiten Prozeß. Der *Slaveprozeß* empfängt die Nachricht und speichert die Datei in einem lokalen Verzeichnis. In Abbildung 3.8 wird der kommentierte Quelltext des Beispiels gezeigt.

Die Nachricht, die der *Master* an den anderen Prozeß sendet, besteht aus 3 Teilstücken. Das Zusammenstellen der Nachricht im Speicher ist in Abbildung 3.7 dargestellt. Auf der linken Seite sieht man die Funktionsaufrufe. Gegenüber ist der Aufbau der Nachricht im Speicher auszugsweise dargestellt. In dem Kasten `Header` bezeichnet `destination` den Empfänger der Nachricht, `header_size` gibt den Speicherplatz im Puffer `header_buffer` des *Headers*, der durch Teilstücke belegt ist, an. Mit `vectors` ist ein Feld von statisch angelegten Vektoren dargestellt. Zur besseren Übersichtlichkeit sind die Elemente dieses Feldes außerhalb des *Headers* plaziert.

Zunächst wird der Name der Datei im *Header* gespeichert. Anschließend wird dort die Länge der Datei vermerkt. Schließlich werden die Speicheradresse des Puffers, in den die Datei geladen wurde, und ihre Länge im Vektor 1 notiert. Der Puffer wird dabei nicht kopiert (Modus MAD_IN_PLACE).

Der Empfänger wertet zunächst die Information im *Header* der Nachricht aus (Dateiname und Dateilänge). Anschließend legt er einen Puffer für die Datei an. Dann wird die Datei in diesem Puffer gelesen. Die Datei wird dabei erst durch den Aufruf von `mad_recvbuf_receive` gelesen. Mit der Aufruf von `mad_unpack_byte` wird nur ihre Größe und die Adresse des Puffers angegeben. Schließlich wird eine Datei geöffnet. Der Inhalt des Puffers wird in diese Datei geschrieben.

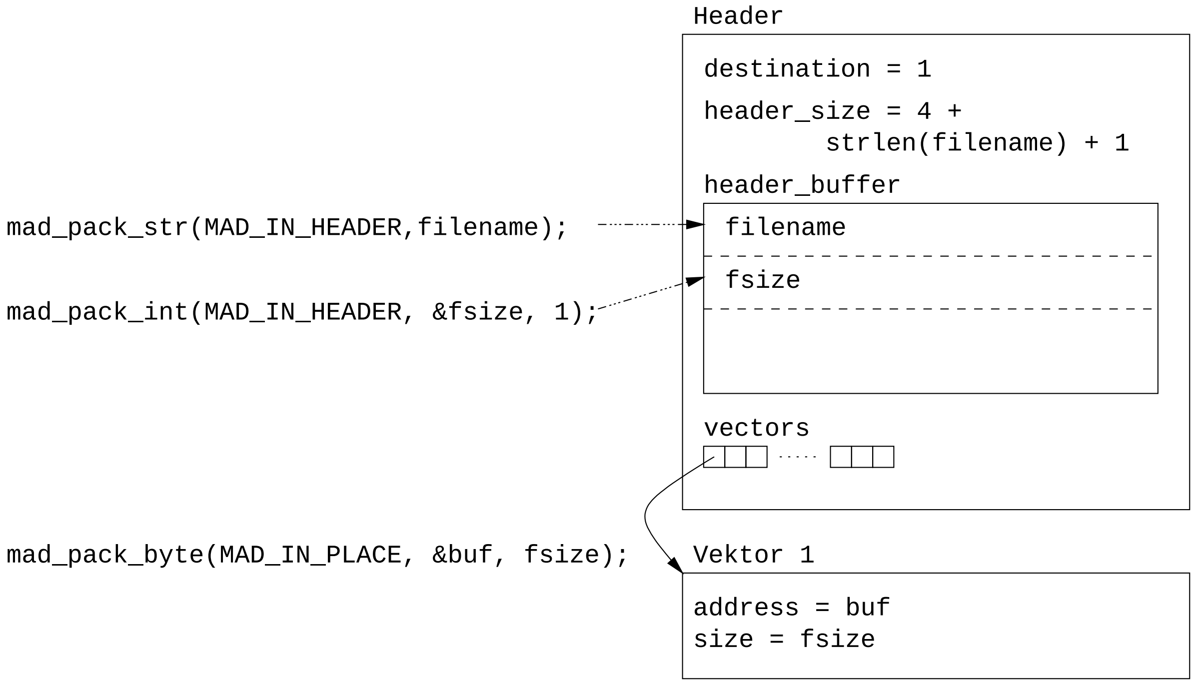

Abbildung 3.7: Zusammenstellen einer Nachricht in MADELEINE

```
#include <mad.h>                                 Stellt Schnittstelle zur Madeleine-Bibliothek bereit.
#include <stdio.h>

FILE *file = NULL;                               Deklaration von globalen Variablen
char *buf  = NULL;
int  fsize = 0;

void master(char *filename) {                    Funktion, die der Master ausführen wird.
  struct stat stats;

  file = fopen(filename,"r");                    Öffnen der Datei
  stat(filename,&stats);                         Ermittlung der Dateigröße
  fsize = stats.st_size;
  buf = malloc(fsize+1);                         Anlegen eines Puffers für die Datei
  fread(buf,1,fsize,file);                       Lesen der Datei in den Puffer
  fclose(file);

  mad_sendbuf_init(1);                           Vorbereiten eines Sendepuffers
  mad_pack_str(MAD_IN_HEADER,filename);          Kopieren des Dateinamens in den Header
  mad_pack_int(MAD_IN_HEADER,&fsize,1);          Speichern der Dateigröße im Header
  mad_pack_byte(MAD_IN_PLACE,buf,fsize);         Speichern der Adresse des Puffers und der
                                                 Dateigröße in einem Vektor

  mad_sendbuf_send();                            Absenden der Nachricht an den Prozeß 1
}
void slave(void) {                               Funktion, die der Prozeß 1 ausführen wird.
  char filename[256];

  mad_receive();                                 Warten auf eine Nachricht und Lesen des Headers
  mad_unpack_str(MAD_IN_HEADER,filename);        Kopieren des Dateinamens aus dem Header
  mad_unpack_int(MAD_IN_HEADER,&fsize,1);        Kopieren der Dateigröße aus dem Header
  buf = malloc(fsize+1);                         Anlegen eines Puffers für die Datei
  mad_unpack_byte(MAD_IN_PLACE,buf,fsize);       Angabe der Pufferadresse und seiner Größe
  mad_recvbuf_receive();                         Lesen der Datei in den Puffer

  file = fopen(filename,"r+");                   Öffnen der Datei
  fwrite(buf,1,fsize,file);                      Schreiben des Puffers in die Datei
  fclose(file);
}
main(int argc, char *argv[]) {

  mad_init();                                    Initialisierung der Madeleine-Bibliothek

  if (my_number == 0)                            Fallunterscheidung in Abhängigkeit der
    master(argv[1]);                             Identfikationsnummer
  else                                             0 ==> Master führt die Funktion master aus
    slave();                                       1 ==> Slave führt die Funktion slave aus
}
```

Abbildung 3.8: Anwendungsbeispiel für MADELEINE

Kapitel 4

Kombination von DSM-Threads und Madeleine

Dieses Kapitel beschreibt die Kombination von DSM-Threads und MADELEINE. Zunächst wird das Vorgehen beim Zusammenführen diskutiert. Anschließend werden das Design und die Implementierung eines Zwischenschrittes beschrieben.

4.1 Methodik

Bei der Kombination gibt es vier Komponenten, die berücksichtigt werden müssen. Diese sind die Initialisierung, die Kommunikation bzw. die Kommunikationsaufrufe, die Erzeugung entfernter Threads und die Terminierung.

Die Kombination von DSM-Threads und MADELEINE ersetzt die ursprüngliche Kommunikation in DSM-Threads durch MADELEINE. Daher müssen alle kommunikationsbezogenen Aufrufe in DSM-Threads berücksichtigt werden. Um MADELEINE benutzen zu können, muß das Kommunikationspaket initialisiert werden. Da möglichst wenige Änderungen an MADELEINE vorgenommen werden sollen, muß die Initialisierung von DSM-Threads angepaßt werden. Das bedeutet, daß bereits während der Initialisierung alle Prozesse entsprechend der Konfiguration erzeugt werden. Diese Tatsache macht die Änderung des Erzeugens von (entfernten) Threads notwendig. Außerdem ist eine Überarbeitung der Terminierung erforderlich.

Die aufgezählten Punkte können in zwei Kategorien aufgeteilt werden. Eine Kategorie enthält die Initialisierung und die Kommunikation. Diese Aspekte müssen immer berücksichtigt werden, wenn ein Kommunikationspaket durch ein anderes ausgetauscht wird. In die andere Kategorie gehören die Erzeugung entfernter Threads und die Terminierung. Diese ergeben sich aus den Änderungen, die aufgrund der ersten Kategorie vorgenommen wurden.

Um die Komplexität der Durchführung der Kombination gering zu halten, wurde die Kombination in zwei Schritte unterteilt. Im ersten Schritt (Zwischenschritt) wurden die Initialisierung, die Erzeugung entfernter Threads und die Terminierung bearbeitet. Die Kommunikation wurde zunächst beibehalten, MADELEINE wurde also noch nicht als Kommunikationspaket eingesetzt. Im zweiten Schritt wurden die Kommunikationsaufrufe in *DSM-Threads* umgestellt. Für diesen Schritt waren auch umfangreiche Änderungen in MADELEINE notwendig. Außerdem mußte die Initialisierung erneut bearbeitet werden.

Der Zwischenschritt bereitet die Einbindung von MADELEINE vor. Nach der Implementierung des Zwischenschrittes konnte das System auf seine Funktion getestet werden. Im nächsten Abschnitt wird das Design des Zwischenschrittes dargestellt.

4.2 Design des Zwischenschrittes

Ausgangsbasis für den Zwischenschritt war die Implementierung von DSM-Threads, die auf dem Netzwerkprotokoll TCP/IP beruht. Im folgenden wird das Design für die Komponenten Start/Initialisierung, Erzeugung entfernter Threads und Terminierung erläutert.

4.2.1 Programmstart und Initialisierung

Der Programmstart muß so geändert werden, daß bereits alle Prozesse zu Beginn gestartet werden. Für den Programmstart wird ein Skript verwendet. Dieses Skript wertet die Konfigurationsdatei `.rhosts` aus. Es startet den ersten Prozeß auf dem Rechner, dessen Name als erster in der Konfigurationsdatei genannt wird.

Dieser Prozeß wird *Master* oder *Masterprozeß* genannt. Der *Master* initialisiert seine Komponenten, so auch den Kommunikationsserver, der an einem Port auf Nachrichten wartet. Die Nummer dieses Ports wird an das Startskript übergeben. Anschließend wartet der *Master* darauf, daß die anderen Prozesse ihre Portnummern mitteilen. Die anderen Prozesse, im folgenden auch als *Slaves* bezeichnet, werden vom Startskript nach Erhalt der Portnummer des *Masters* gestartet. Die *Slaves* initialisieren ebenfalls ihre Komponenten. Jeder Prozeß sendet eine Nachricht mit seiner Portnummer an den *Master* und wartet auf eine Antwort des *Masters*. Nachdem dieser von allen Prozessen deren Portnummer erhalten hat, teilt er jedem Prozeß die Portnummern aller Prozesse mit.

4.2.2 Erzeugung entfernter Threads

Ein entfernter Thread kann in jedem Prozeß erzeugt werden. Das Erzeugen eines entfernten Threads ist in Abbildung 4.1 dargestellt. Der erzeugende Thread (T in Prozeß X) schickt eine THREAD_CREATE-Nachricht an den Prozeß Y, in dem der neue Thread erzeugt werden soll. Der Kommunikationsserver des Zielprozesses empfängt die Nachricht und erzeugt den Thread. Anschließend sendet er eine Antwort (THREAD_CREATE_ACK-Nachricht) an den erzeugenden Prozeß (T in Prozeß X). Der Kommunikationsserver in X signalisiert den erzeugenden Thread, der auf das Ergebnis der Operation gewartet hat. Der erzeugende Thread informiert nun den *Master* darüber (THREAD_CREATED-Nachricht), daß ein neuer Thread erzeugt wurde. Diese Information ist für die Terminierung wichtig. Bevor der erzeugende Thread fortfahren kann, muß er noch auf eine Antwort (THREAD_CREATED_ACK-Nachricht) des *Masters* warten. Diese ist notwendig, um Fehler bei der Terminierung auszuschließen, falls Nachrichten beim *Master* nicht in der Reihenfolge des Absendens bearbeitet werden. Für eine detaillierte Darstellung dieses Problems wird auf den Abschnitt 5.4 verwiesen.

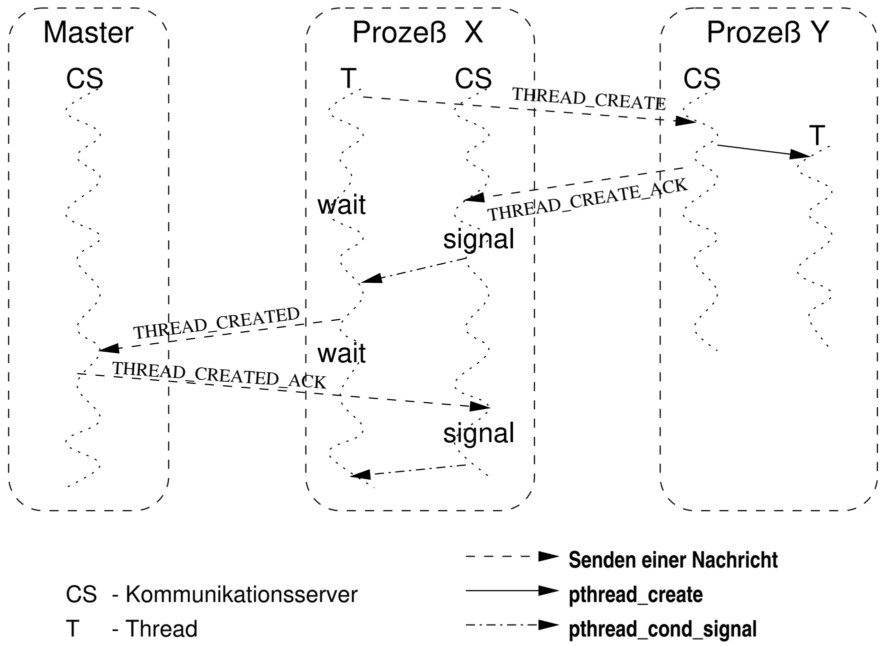

Abbildung 4.1: Modell der Erzeugung eines entfernten Threads

4.2.3 Terminierung

Wird ein mit obiger Methode erzeugter Thread beendet, wird eine THREAD_TERMINATED-Nachricht an der *Master* gesendet. Hiermit teilt der Thread dem *Master* mit, daß er terminiert. Der *Master* wartet bis alle Threads terminiert haben. Dann sendet er an alle Prozesse eine Nachricht, daß sie mit ihrer Terminierung beginnen bzw. fortfahren können. Dieses Verfahren ist notwendig, um zu erkennen, wann die Prozesse beendet werden können. In Abschnitt 5.4 wird die Terminierung detailliert besprochen.

4.3 Implementierung des Zwischenschrittes

In diesem Kapitel wird nur der Teil der Implementierung beschrieben, der im zweiten Schritt der Kombination überarbeitet wird. Dies betrifft das Startskript sowie den Austausch der Portnummern. Der andere Teil wird in Kapitel 6 detailliert erläutert.

Zur Illustration betrachte man die Abbildung 4.2, in der die Schritte der Initialisierung dargestellt sind. Das Startskript (0) ist in der Skriptsprache Perl geschrieben. Es öffnet die Konfigurationsdatei `.rhosts`, die sich im aktuellen Verzeichnis befinden muß. Anschließend liest es die Rechnernamen aus der Datei (1) und speichert sie in einem Feld ab. Rechnernamen, die mehrfach in der Datei aufgeführt sind, werden nur einmal in dem Feld gespeichert. Somit kann eine Konfigurationsdatei ohne Änderungen sowohl von der alten als auch von der neuen Version des Systems benutzt werden.

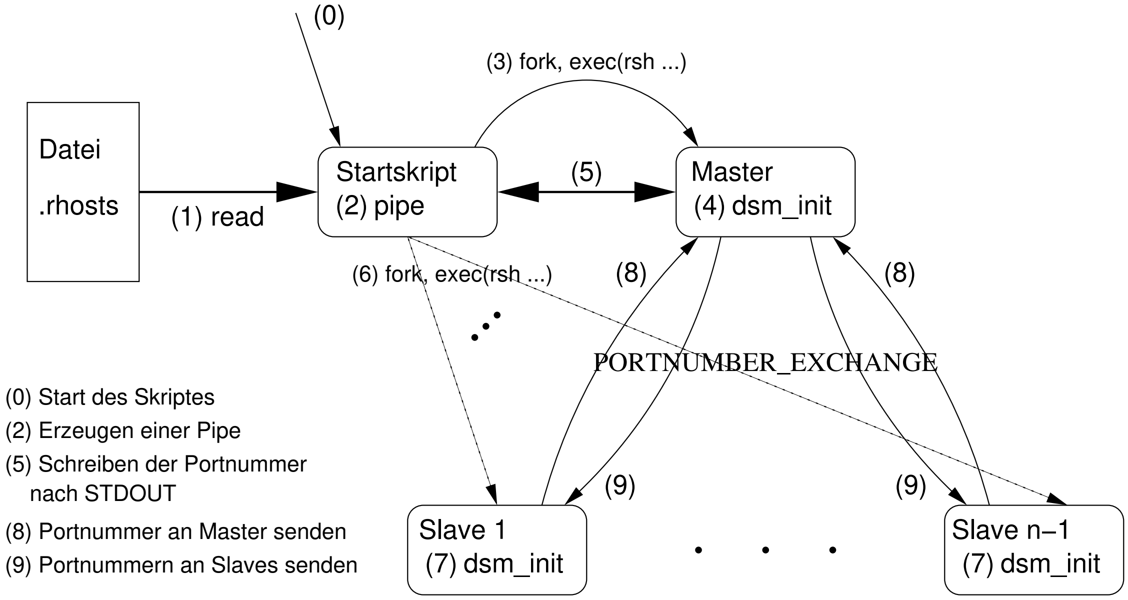

Abbildung 4.2: Implementierung der Initialisierung nach dem Design des Zwischenschrittes

Nun wird auf dem Rechner, der als erster in der Konfigurationsdatei steht, der *Masterprozeß* gestartet. Dazu ruft das Skript die Funktion `open` mit dem Parameter `program|` auf. Dies bewirkt, daß zunächst eine Pipe geöffnet wird (2). Sie dient zur Kommunikation zwischen dem Vaterprozeß und dem Kindprozeß, der im Anschluß erzeugt wird (3). So können Ausgaben des Kindprozesses in die Standardausgabe, beim Vaterprozeß aus der Standardeingabe gelesen werden. Der Kindprozeß öffnet mit dem Programm `rsh` eine Shell auf dem Zielrechner (3). In dieser Shell wird das auszuführende Programm gestartet. Der Vaterprozeß erwartet die Übergabe der Portnummer (5) an der Standardeingabe (STDIN). Das gestartete Programm führt seine Initialisierung durch (4).

Bevor die Portnummern ausgetauscht werden, wird noch eine Hilfsstruktur für das Erzeugen entfernter Threads angelegt. Diese Struktur wird in Abschnitt 6.3 erläutert.

Nachdem der Kommunikationsserver gestartet wurde, ist auch der Port für ankommende Nachrichten eingerichtet. Dessen Portnummer wird in die Standardausgabe geschrieben (5). Das Startskript liest die Portnummer von der Standardeingabe und startet die *Slaveprozesse* mit dem Namen des Rechners, auf welchem der *Master* gestartet wurde, und der Portnummer des *Masters* als Parameter (6). Die *Slaveprozesse* beginnen mit ihrer Initialisierung (7). Nachdem ihre Kommunikationskomponenten gestartet wurden, senden sie ihre Portnummer in einer PORTNUMBER_EXCHANGE-Nachricht an den *Master*. Dieser hat auf diese Nachrichten gewartet, seitdem er seine Portnummer dem Startskript mitgeteilt hatte. Das Warten wird von dem Main-Thread des *Masters* durchgeführt. Der Kommunikationsserver empfängt die Nachrichten der *Slaveprozesse* und speichert die Portnummern in einer zirkulären Liste, die von dem Modul rhosts implementiert wird. Jedes Listenelement speichert den Namen eines Rechners und die Portnummer des Prozesses, der auf diesem Rechner gestartet wurde.

Anhand der Anzahl der Elemente in dieser Liste und der Anzahl der empfangenen Nachrichten stellt der Kommunikationsserver fest, wann alle *Slaveprozesse* ihre Portnummer übermittelt haben. Dann signalisiert er den wartenden Main-Thread. Dieser stellt eine PORTNUMBER_EXCHANGE-Nachricht, die die Portnummern aller Prozesse enthält, zusammen und schickt eine Kopie an jeden *Slaveprozeß* (9). Diese warten, seitdem sie ihre Portnummer an den *Master* gesendet haben, auf diese Nachricht. Der Kommunikationsserver eines *Slaveprozesses* empfängt die Nachricht vom *Master* und speichert die Portnummern in den Listenelementen. Anschließend kann der Main-Thread des *Slaveprozesses* mit der Initialisierung fortfahren.

Der *Master* initialisiert zwei Seitenverwaltungsstrukturen, die für die Terminierungsbarriere benötigt werden. Die Implementierung der Terminierungsbarriere wird in Abschnitt 6.4 beschrieben.

Die *Slaveprozesse* starten am Ende der Initialisierungsprozedur die Funktion `dsm_exit`. Die Ausführung kehrt nie in die Hauptroutine `main` zurück. Die Anweisungen, die in der Hauptroutine `main` nach der Funktion `dsm_init` folgen, haben keine Bedeutung für die *Slaveprozesse*. Diese Anweisungen werden wie bisher nur vom *Master* ausgeführt. Am Ende der Initialisierungsphase ist auf jedem Rechner ein Prozeß gestartet, der alle Komponenten des DSM-Systems gestartet hat und zum Empfangen von Nachrichten bereit ist. Im *Masterprozeß* gibt es zu diesem Zeitpunkt den einzigen Benutzer-Thread, der mit der Ausführung des Programmes fortfährt.

Kapitel 5

Design des neuen Systems

In diesem Abschnitt wird das Design des neuen Systems beschrieben. Dabei werden nur die veränderten Komponenten berücksichtigt. Zu Beginn wird die Initialisierung erläutert. Daran schließt sich die Darstellung der Kommunikation an. Für die Erzeugung entfernter Threads wird eine weiterentwickelte Variante, die mit geringerem Nachrichtenaufwand auskommt, erläutert. Die Terminierung wurde in Abschnitt 4.2.3 grob skizziert. Sie wird hier detailliert dargestellt.

5.1 Initialisierung

Die Initialisierung umfaßt das Starten des Programmes und die Initialisierungsfunktion. Das Starten eines Programmes wird durch ein Skript realisiert. Dieses Skript hat die Aufgabe, in Abhängigkeit der verwendeten Kommunikationsarchitektur (TCP/Ethernet oder BIP/Myrinet), die Konfiguration zu überprüfen und das richtige Startskript auszuwählen. Es vereinfacht die Benutzung von DSM-Threads.

Zu Beginn der Initialisierungsfunktion wird die MADELEINE-Bibliothek initialisiert. Die Identifikationsnummern der Prozesse ersetzen die Portnummern. Dabei wird davon ausgegangen, daß die Nummer eines Prozesses der Position des Rechnernamens in der Konfigurationsdatei entspricht. Die Abbildung von Rechnernamen auf Identifikationsnummern ist für die Beibehaltung der Schnittstelle von `dsm_thread_create` notwendig. Dies gilt, wenn pro Rechner nur ein Prozeß gestartet werden darf (Einschränkung von MADELEINE/BIP). Können mehrere Prozesse pro Rechner gestartet werden, so muß nur die Abbildung von Rechnernamen auf Identifikationsnummern erweitert werden.

Anschließend werden die Systemkomponenten wie z.B. die Seitentabelle und der Kommunikationsserver initialisiert. Wird BIP als Kommunikationsbibliothek eingesetzt, muß ein zusätzlicher Server (als Thread) gestartet werden. Dieser behandelt Nachrichten, die aufgrund fehlender *Credits* nicht gesendet werden konnten. Bevor der *Master* aus der Initialisierungsfunktion zurückkehrt, initialisiert er die Terminierungsbarriere. Die anderen Prozesse kehren nie aus der Initialisierungsfunktion zurück. Sie warten darauf, daß der Prozeß beendet werden kann.

5.2 Kommunikation

Die Schnittstelle, die durch das Modul sock_msg bereitgestellt wird, wurde in dem neuen Modul mad_msg unverändert auf Basis der Funktionen von MADELEINE implementiert. Im folgenden werden Änderungen und Erweiterungen in MADELEINE und DSM-Threads erläutert, die für die Unterstützung von BIP vorgenommen werden mußten. Für die Unterstützung von TCP/IP waren zusätzlichen Änderungen notwendig.

Das Senden einer Nachricht in MADELEINE erfordert unter bestimmten Voraussetzungen eine Synchronisation zwischen Sender und Empfänger (vgl. Ausführungen zu BIP in Abschnitt 3.2.2). Wesentlich dabei ist, daß der sendende Thread selbst auf eine Antwort von dem Empfänger wartet. Er empfängt also eine BIP-Nachricht. Wie in Abschnitt 2.2.2 diskutiert wurde, darf in DSM-Threads immer nur ein Thread Nachrichten empfangen. Dies gilt auch für BIP-Nachrichten. In DSM-Threads empfängt der Kommunikationsserver alle Nachrichten. Daher wartet der sendende Thread an einer Bedingungsvariable, die der Kommunikationsserver nach dem Eintreffen der Antwort signalisiert.

Eine Erweiterung betrifft den *Credits*-Mechanismus. Besitzt ein Sender keinen *Credit* für einen Empfänger, so kann die Nachricht nicht gesendet werden. Der sendende Thread könnte nun einfach blockieren und somit warten, bis der Empfänger *Credits* zurückgesendet hat. Dies ist die bisherige Vorgehensweise von MADELEINE. Ist der Kommunikationsserver der sendende Thread, so würde er dauerhaft blockieren, da er die zurückgesendeten *Credits* nicht in Empfang nehmen könnte. Daher ist diese Vorgehensweise für die Verwendung mit DSM-Threads nicht geeignet. Betrachtet man das Problem genauer, so stellt man fest, daß es nicht notwendig ist, daß der sendende Thread blockiert wird. Es muß nur dafür gesorgt werden, daß die Nachricht zu einem späteren Zeitpunkt, wenn wieder *Credits* vorhanden sind, gesendet wird. Diese Methode wird durch Verwendung einer Warteschlange für „verzögerte“ Nachrichten und einen Server, der diese Warteschlange bearbeitet, angewendet. Erhält ein Prozeß *Credits* zurück, wird der Warteschlangenserver signalisiert.

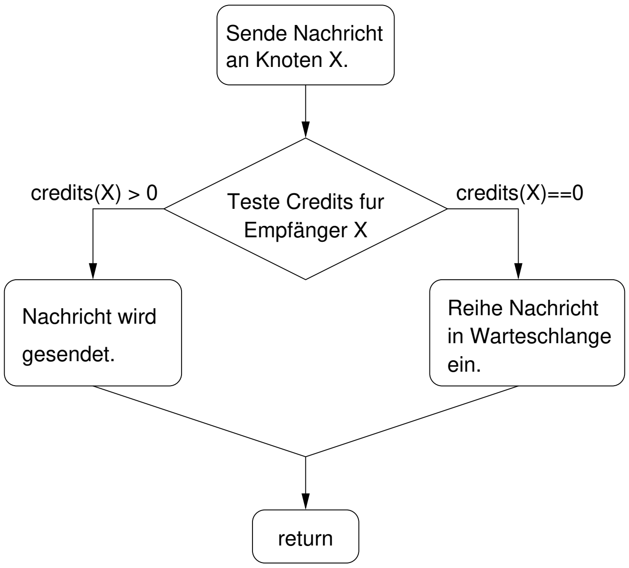

Abbildung 5.1: Erweiterter Algorithmus (Ausschnitt) zum Sendens einer DSM-Nachricht

Der Server kann sowohl in DSM-Threads als auch in MADELEINE integriert werden. Bei der Variante, in der er in DSM-Threads integriert wird, müssen aber DSM-Threads und MADELEINE eng miteinander verzahnt werden. Einerseits müßte DSM-Threads auf interne Strukturen von MADELEINE zugreifen. Andererseits müßte der Warteschlangenserver durch MADELEINE signalisiert werden. Daher ist es sinnvoller, den Server in MADELEINE zu integrieren. DSM-Threads bleibt bei dieser Variante unberührt. In Abbildung 5.1 wird ein erweiterter Algorithmus für das Senden einer Nachricht gezeigt.

5.3 Erzeugung entfernter Threads

Ein neuer Thread kann durch Aufruf der Funktion `dsm_thread_create` erzeugt werden. Dieser muß nicht in einem entfernten Prozeß erzeugt werden. Für das Verfahren spielt dies jedoch keine Rolle. In Abschnitt 4.2.2 wurde bereits ein Verfahren zur Erzeugung eines entfernten Threads vorgestellt. Dieses Verfahren wurde weiterentwickelt.

Das weiterentwickelte Modell der Erzeugung eines entfernten Threads ist in Abbildung 5.2 dargestellt. Es beruht auf dem Austausch von Nachrichten zwischen Prozessen, wobei hier davon ausgegangen werden soll, daß alle dargestellten Prozesse voneinander verschieden sind. Der erzeugende Thread (T in Prozeß X) sendet eine DSM-Nachricht vom Typ THREAD_CREATE an den Zielprozeß Y und wartet auf eine Antwort. Der Kommunikationsserver im Zielprozeß bearbeitet die Nachricht. Er erzeugt den neuen Thread und schickt eine Nachricht (THREAD_CREATED) an den *Masterprozeß*. Der *Masterprozeß* empfängt die Information und sendet eine Bestätigung an den erzeugenden Thread. Mit dieser Variante wird der *Masterprozeß* informiert, und es wird gleichzeitig eine Nachricht gegenüber der Variante in Abbildung 4.1 eingespart. Der *Masterprozeß* muß über die Erzeugung des neuen Threads informiert werden, bevor der erzeugende Thread weiterarbeiten kann. Sonst kann der *Master* die Terminierung des DSM-Systems auslösen, obwohl es noch Benutzer-Threads gibt. Wann dieser Fall auftreten kann, wird in Abschnitt 5.4 erläutert.

Werden mehrere Prozesse pro Rechner unterstützt, so muß das Verfahren nicht geändert werden. Lediglich die Schnittstelle der Funktion `dsm_thread_create` müßte angepaßt werden. Bisher wurde der Prozeß, in dem der neue Thread erzeugt werden soll, durch den Namen des Rechners, auf dem der Prozeß ausgeführt wird, identifiziert. Die Angabe des Rechnernamens reicht nun nicht mehr aus. Denkbar wäre, den Rechnernamen um eine Nummer zu erweitern. Diese Nummer könnte die Identifikationsnummer, die jedem Prozeß von MADELEINE zugewiesen wird, sein. Der Benutzer hätte somit drei Möglichkeiten, den Zielprozeß anzugeben.

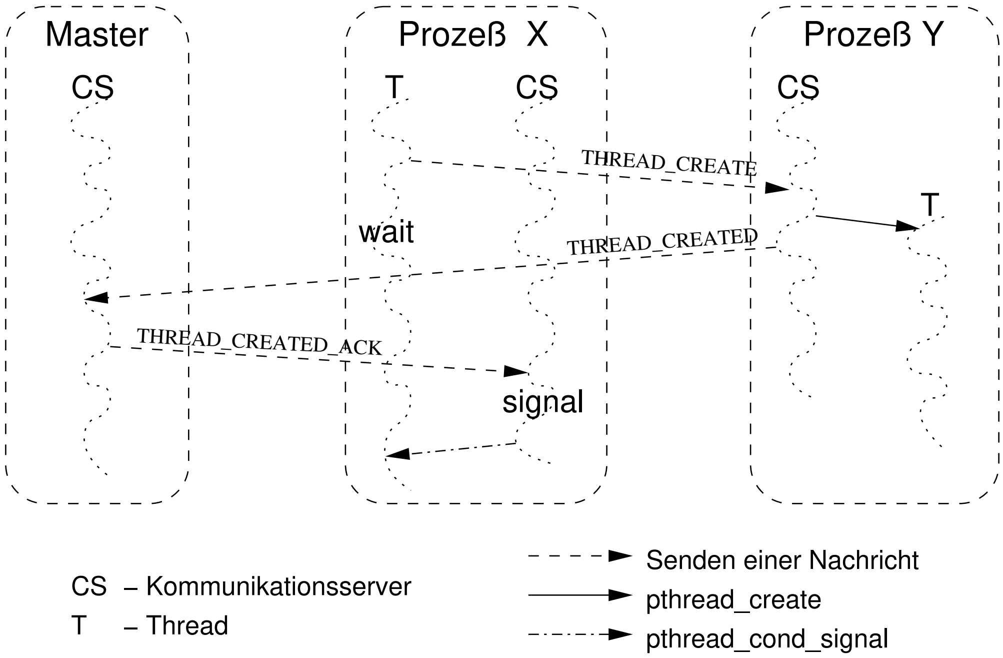

Abbildung 5.2: Modell der Erzeugung eines entfernten Threads

Automatische Auswahl Bei dieser Möglichkeit wird der Prozeß nicht explizit angegeben. Das System wählt nach dem Round-Robin-Verfahren den nächsten Prozeß aus.

Halbautomatische Auswahl Hier wird nur der Name eines Rechners angegeben. Gibt es mehrere Prozesse auf diesem Rechner, so wird der nächste gemäß Round-Robin ausgewählt. Andernfalls wird der einzige Prozeß ausgewählt.

Exakte Auswahl Diese Möglichkeit erfordert die Angabe des Rechnernamens und der Identifikationsnummer des Prozesses.

5.4 Terminierung

Bevor ein Prozeß terminiert, synchronisiert er sich mit allen anderen Prozessen des DSM-Systems. Für die Synchronisation wird eine Barriere verwendet. Das System muß erkennen, zu welchem Zeitpunkt alle Benutzer-Threads terminiert haben. In dieser Situation können keine neuen Threads erzeugt werden. Also können die Prozesse terminieren.

Der *Masterprozeß* verwaltet einen Zähler für die Anzahl der Benutzer-Threads. Dieser wird inkrementiert, wenn ein neuer Thread erzeugt wurde. Wenn ein Thread terminiert hat, wird er dekrementiert. Erreicht der Zähler den Wert Null, sendet der *Master* an alle Prozesse eine Terminierungsnachricht (FINISH_REMOTE). Dieses Verfahren wurde vor die bereits existierende Barriere gestellt. Nun werden also zwei Barrieren verwendet. Die ursprüngliche Barriere diente auch zum Sammeln von Status-Daten. Zu diesem Zweck wurde sie im neuen System beibehalten, obwohl sie für die eigentliche Terminierung nicht mehr notwendig ist. Nach den Barrieren wird die Terminierung des Systems abgeschlossen.

Wie im vorherigen Abschnitt erläutert, erfordert die Terminierungsroutine, daß der *Master* die Information, daß ein neuer Thread erzeugt wurde (Erzeugungsnachricht), bestätigt. Der erzeugende Thread wird bis zum Eingang dieser Bestätigung blockiert. Würde er nicht auf die Bestätigung warten, so könnte es zu einer inkorrekten Terminierung kommen. Dabei würde der *Master* die Terminierungsnachricht senden, obwohl noch Benutzer-Threads existieren. Diese Situation kann eintreten, wenn die Erzeugungsnachricht erst bearbeitet wird, nachdem alle anderen Threads, im besonderen der erzeugende Thread, ihre Terminierung dem *Master* mitgeteilt haben (THREAD_TERMINATED-Nachricht). Angenommen der erzeugende Thread sendet zunächst die Erzeugungsnachricht und terminiert anschließend (THREAD_TERMINATED-Nachricht), dann kann es für die umgekehrte Verarbeitungsreihenfolge zwei Ursachen geben.

1. Die Nachrichten kommen in anderer Reihenfolge beim *Master* an (vgl. Abbildung 5.3a).
2. Der Kommunikationsserver, der die Nachrichten empfängt, bearbeitet die Erzeugungsnachricht nicht selbst, sondern reiht sie in eine Warteschlange ein. Bevor ein *Worker* die Nachricht bearbeitet, erhält der Server die THREAD_TERMINATED-Nachricht und bearbeitet sie sofort (vgl. Abbildung 5.3b).

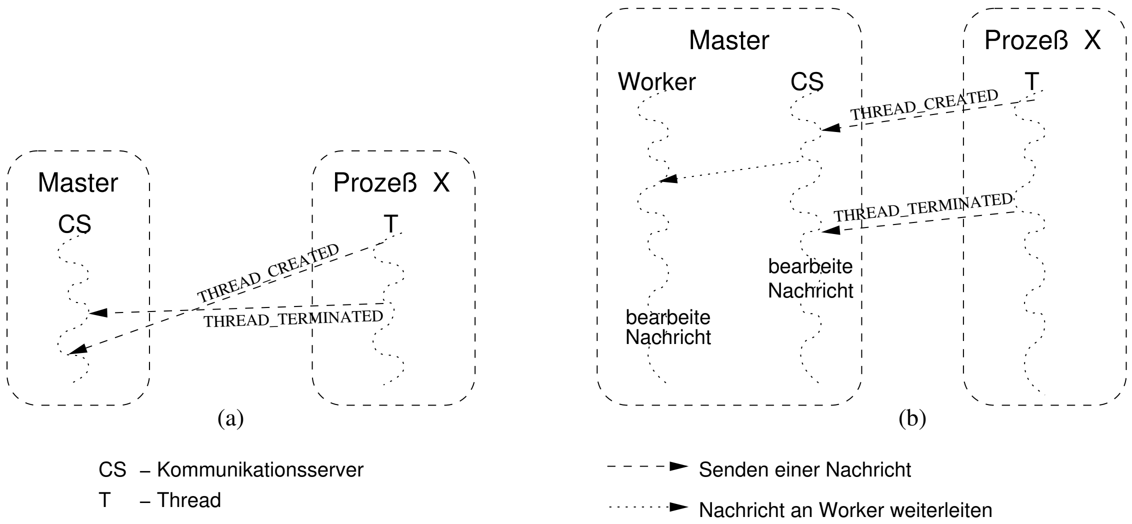

Abbildung 5.3: Nachrichten-Race bei der Terminierung

Beide Fälle werden durch das Warten auf Bestätigung (THREAD_CREATED_ACK) ausgeschlossen. Nun kann der erzeugende Thread erst nach Erhalt der Bestätigung terminieren (vgl. Erläuterungen in Abschnitt 5.3 sowie Abbildung 5.2). Der *Master* sendet die Bestätigung aber erst, nachdem die Erzeugungsnachricht bearbeitet wurde.

Kapitel 6

Implementierung des neuen Systems

Dieses Kapitel beschreibt die Implementierung des neuen Systems. Dabei werden nur jene Komponenten berücksichtigt, die verändert oder erweitert wurden. Zu Beginn wird auf die Initialisierung beschrieben. Anschließend wird die Implementierung der Kommunikation mit einer detaillierten Beschreibung der Behandlung von Nachrichten (Senden und Empfangen) erläutert. Der dritte Abschnitt beschäftigt sich mit der Erzeugung von entfernten Threads. Im letzten Teil wird die Implementierung der Terminierung erläutert.

6.1 Initialisierung

Dieser Abschnitt ist in die Teile **Startskript** und **Initialisierungsfunktion** unterteilt. Im ersten Teil wird beschrieben, wie die unterschiedlichen Methoden ein Programm unter MADELEINE zu starten, kombiniert werden. Anschließend werden Änderungen und Erweiterungen der Initialisierungsfunktion erläutert.

6.1.1 Startskript

Ein Programm wird mit Hilfe des Skriptes `dsmload` gestartet. Dieses Skript ermittelt, welches Netzwerkprotokoll (TCP) bzw. welche Bibliothek (BIP) von MADELEINE eingesetzt werden soll. Dazu wird die Ausgabe des Skriptes `madcustom` ausgewertet. Das Startskript kann dann die notwendigen Vorbereitungen (Konfigurationsdateien) für den Programmstart treffen und durch Aufruf des kommunikationsspezifischen Startskriptes - `madload` für TCP, `bipload` für BIP - das Programm starten. Der Ablauf des Startskriptes `dsmload` ist in Abbildung 6.1 dargestellt. Die Implementierung des Skriptes wird anhand der Abbildung erläutert. In diesem Beispiel soll das Programm `lu` (berechnet die LU-Zerlegung einer Matrix) mit den Parametern `-n512` und `-p2` gestartet werden.

Zunächst wertet das Skript die Ausgabe von `madcustom` aus. Die einzeilige Ausgabe

```
The current network interface is set to:  bip
```

enthält am Ende eine Zeichenkette, die das Netzwerkprotokoll (`tcp`) bzw. die Kommunikationsbibliothek (`bip`) bezeichnet. Die folgenden Arbeitsschritte werden zunächst für TCP und anschließend für BIP beschrieben.

Startskript für TCP

MADELEINE benötigt die Konfigurationsdatei `.madconf`, die sich in einem Verzeichnis, welches durch die Umgebungsvariable `MADELEINE_ROOT` angegeben wird, befindet. Diese Datei hat dieselbe Syntax wie die Konfigurationsdatei `.rhosts` von DSM-Threads. Das neue System benötigt beide Dateien. Ist nur eine vorhanden, wird eine Kopie mit dem Namen der anderen erzeugt. Falls beide Dateien fehlen, wird das Skript mit einer Fehlermeldung beendet.

Anschließend kann das Skript `madload` mit den Parametern durch Aufruf der Funktion `exec` gestartet werden. An diesem Skript wurden keine Veränderungen vorgenommen.

Startskript für BIP

MADELEINE benötigt die Konfigurationsdatei `.bipconf`, die sich im Verzeichnis `${HOME}/.bip` befindet. Die Umgebungsvariable `HOME` gibt das Hauptverzeichnis eines Benutzers an. Falls die Datei nicht existiert, wird das Skript mit einer Fehlermeldung beendet. Andernfalls wird die Datei geöffnet und gelesen. Die

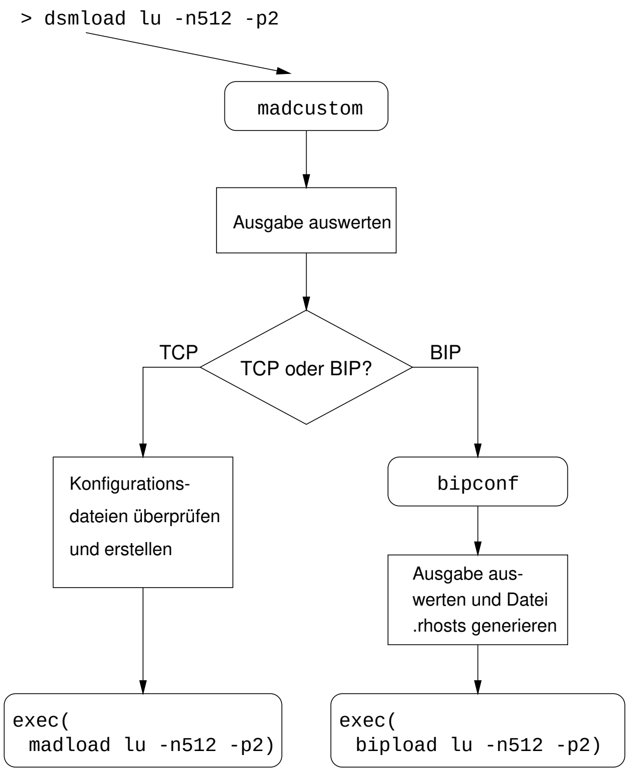

Abbildung 6.1: Arbeitsschritte des Startskriptes für TCP und BIP

Rechnernamen werden zeilenweise in die Datei `.rhosts.bip` im selben Verzeichnis geschrieben. Anschließend wird ein symbolischer Verweis (Link), der von der Datei `.rhosts` im Arbeitsverzeichnis auf die Datei `.rhosts.bip` verweist, generiert. Falls bereits eine Datei `.rhosts` im Arbeitsverzeichnis existiert, wird diese zuvor gelöscht. Das DSM-System, das die Konfigurationsdatei `.rhosts` benötigt, braucht so nicht geändert werden.

Anschließend wird das Skript `bipload` mit den Parametern durch Aufruf der Funktion `exec` gestartet. An diesem Skript wurden keine Veränderungen vorgenommen.

6.1.2 Initialisierungsfunktion

Am Anfang eines Programmes wird die Funktion `dsm_init` aufgerufen. Dieser Abschnitt beschreibt diese Funktion. Dabei wird detailliert auf die Veränderungen eingegangen. Eine Beschreibung der Originalfassung befindet sich in Abschnitt 2.2.1.

Zu Beginn wird das Modul rhosts, das die Namen aller verfügbaren Rechner in einer zirkulären Liste verwaltet, initialisiert. Die Funktion `dsm_hostinit` wertet dafür die Datei `.rhosts` aus. Während die Liste mit den Rechnernamen erstellt wird, wird jedem Rechnernamen auch seine Identifikationsnummer, die von Madeleine benutzt wird, zugewiesen. Dabei wird angenommen, daß Madeleine diese Nummern entsprechend der Reihenfolge der Rechnernamen in der Konfigurationsdatei `.rhosts` bzw. `.madconf` vergibt. Weiterhin wird vorausgesetzt, daß jeweils ein Prozeß auf einem Rechner gestartet wurde. Treten Rechnernamen mehrfach in der Konfigurationsdatei auf, wird für jede Nennung ein Listenelement erzeugt. Listenelementen mit dem gleichen Namen wird dieselbe Identifikationsnummer zugewiesen. In Abbildung 6.2 ist die Deklaration eines Elementes der Liste dargestellt. Die Variable `host` speichert den Rechnernamen, während die Variable `port` die Identifikationsnummer aufnimmt.

```
typedef struct dsm_rhosts {
  char *host;
  int  port;
  struct dsm_rhosts *next;
} dsm_rhosts_s;
```

Abbildung 6.2: Deklaration eines Listenelementes

Die gewählte Methode der Zuweisung von Identifikationsnummern zu Rechnernamen hat den Vorteil, daß sie einfach zu programmieren ist und keiner zusätzlichen Kommunikation der teilnehmenden Prozesse bedarf. Ein Nachteil der Methode ist, daß sie davon abhängig ist, wie MADELEINE die Nummern zuordnet. Wird das Verfahren in MADELEINE geändert, so muß auch die Initialisierungsroutine `dsm_hostinit` angepaßt werden.

Eine andere Methode kann analog zum Austausch der Portnummern (vgl. Erläuterungen in Abschnitt 4.3) implementiert werden. Während die zirkuläre Liste angelegt wird, werden nur die Rechnernamen gespeichert. Nachdem das Kommunikationspaket und der Kommunikationsserver initialisiert wurden, werden die Identifikationsnummern mittels Nachrichten ausgetauscht. Ein möglicher Algorithmus ist in Abbildung 6.3 dargestellt. Sei n die Anzahl der Prozesse. Dann sendet der *Masterprozeß* (ID 0) allen anderen Prozessen, die die Nummern 1 bis $n-1$ haben, eine Aufforderung (ID-REQ) ihre Daten (Rechnername und Identifikationsnummer) mitzuteilen. Die anderen Prozesse beantworten diese Anfrage mit einer ID-RESP–Nachricht. Anschließend warten sie auf eine ALL-IDs–Nachricht, die die Identifikationsnummern aller Prozesse enthält. Der *Masterprozeß* empfängt die Antworten (ID-RESP) der anderen Prozesse und schickt sämtliche Paare $(Rechnername, Identifikationsnummer)$ in einer ALL-IDs–Nachricht an alle Prozesse. Die anderen Prozesse empfangen diese Nachrichten (ALL-IDs) und speichern die Identifikationsnummern in den entsprechenden Elementen der Liste.

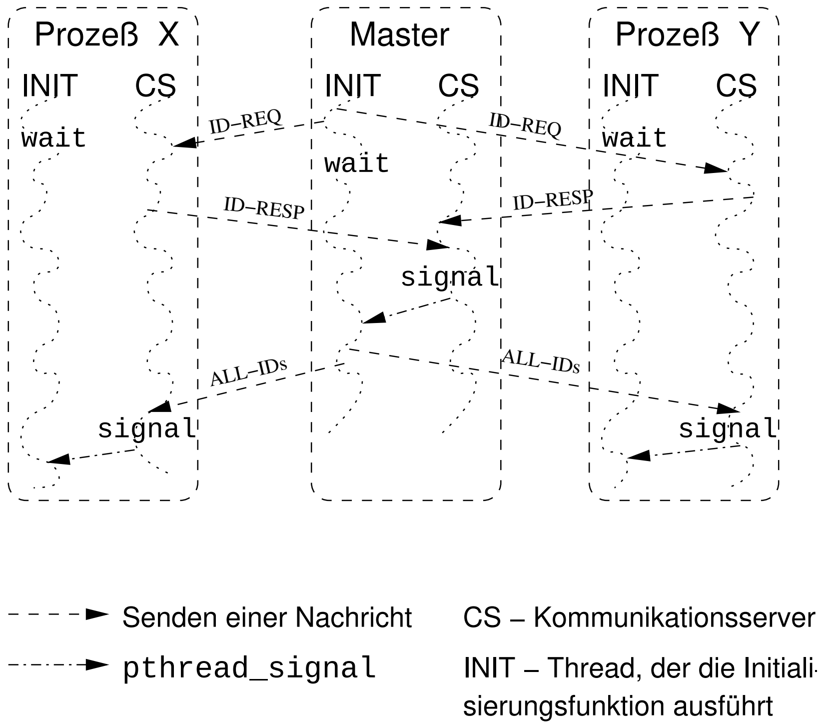

Abbildung 6.3: Algorithmus zum Austausch der Identifikationsnummern

Ein Vorteil dieser Methode ist, daß der Algorithmus nicht geändert werden muß, wenn sich die Nummern-Zuordnung von MADELEINE verändert. Dies gilt, solange der *Masterprozeß* die ID Null behält. Der komplexere Algorithmus ist ein Nachteil, zumindest im Vergleich mit der anderen Methode.

Nachdem die Initialisierung des Moduls `rhosts` abgeschlossen ist, wird das Kommunikationspaket MADELEINE durch Aufruf der Funktionen `mad_init` und `mad_buffers_init` initialisiert. Wird BIP als Bibliothek eingesetzt, wird durch `mad_init` ein zusätzlicher Thread - der Server für verzögerte Nachrichten, auch *Delayed Messages Server* genannt - gestartet. Dieser Thread bearbeitet die Warteschlange für Nachrichten, die aufgrund fehlender *Credits* nicht gesendet werden konnten. Eine detaillierte Beschreibung dieses Servers folgt in Abschnitt 6.2. Anschließend wird die prozeß-lokale Variable `dsm_UserThreadsCounter` gesetzt. Der *Masterprozeß* zählt mit dieser Variable die Anzahl der Benutzer-Threads. Der Thread, der die Initialisierung im *Masterprozeß* durchführt, ist ein Benutzer-Thread, daher wird die Variable mit dem Wert 1 initialisiert. In den anderen Prozessen wird die Variable mit 99 oder irgendeinem anderen Wert ungleich Null initialisiert. Diese Prozesse benutzen die Variable für die neue erste Terminierungsbarriere. Die Implementierung dieser Barriere wird in Abschnitt 6.4 beschrieben.

Es folgt der Aufruf der Funktion `dsm_protect`, die wichtige Systemkomponenten, wie z.B. der Kommunikationsserver, startet. Nun werden die prozeß-lokalen Variablen `dsm_main_thread` und `dsm_is_remote` initialisiert. Erstere speichert die ID des aktuellen Threads, der die Initialisierung durchführt. Diese Variable wird bei der Terminierung benötigt. Die zweite wird im *Master* mit Null und in allen anderen Prozessen mit Eins initialisiert. Sie wird an einigen Stellen benutzt, um festzustellen, ob der ausführende Thread zum *Master* oder zu einem anderen Prozeß gehört.

Anschließend wird eine Liste, die als Hilfsstruktur für die Erzeugung entfernter Threads dient, angelegt. Für jeden Benutzer-Thread wird ein Element in der Liste eingefügt. Da im *Masterprozeß* die Initialisierung von einem Benutzer-Thread durchgeführt wird, wird für den Main-Thread ein Element in die Liste eingefügt. In den anderen Prozessen bleibt die Liste zunächst leer, da die Initialisierung nicht von einem Benutzer-Thread vorgenommen wird. Die Deklaration eines Listenelementes wird in Abschnitt 6.3 beschrieben.

Für die zweite Terminierungsbarriere wird eine Hierarchie der Prozesse benötigt. Diese Hierarchie spiegelt die Verwandtschaft zwischen den Prozessen wieder (vgl. Erläuterungen in Abschnitt 2.2.4). Im neuen Design ist der *Masterprozeß* per Definition der Vater aller anderen Prozesse. Diese Beziehung wird in einer Liste, die für jeden anderen Prozeß ein Element enthält, gespeichert.

Zuletzt wird in den *Slaveprozessen* die Funktion `dsm_exit` aufgerufen. Der *Masterprozeß* hingegen kehrt aus der Initialisierungsfunktion zurück.

6.2 Kommunikation

Dieser Abschnitt beschreibt die Implementierung der Kommunikation. Zunächst wird erläutert, wie das Modul mad_msg implementiert wurde. Dieses Modul benutzt Funktionen von MADELEINE, um die Kommunikationsschnittstelle von DSM-Threads zu realisieren. Die Funktionen dieser Schnittstelle werden intern im DSM-System für den Nachrichtenaustausch verwendet. Im folgenden werden Änderungen in der Schicht *Portability Interface* von MADELEINE besprochen. Zunächst wird auf Änderungen der TCP-Unterstützung eingegangen. Anschließend werden umfangreiche Änderungen und Erweiterungen der Unterstützung von BIP erläutert. Es werden die beiden Verfahren *Polling* und *Notifikation durch Signale* für das Empfangen einer Nachricht diskutiert. Weiterhin werden die Behandlung langer Nachrichten und der *Delayed Messages Server* - eine Erweiterung von MADELEINE - besprochen.

Das Modul mad_msg

In der ursprünglichen Version von DSM-Threads, stellt das Modul sock_msg Funktionen für den Nachrichtenaustausch bereit. Die Schnittstelle dieser Funktionen, deren Semantik in der Tabelle 2.1 beschrieben wird, wurde beibehalten. So mußte nur die Implementierung geändert werden. Zusätzlich wurde die Schnittstelle um die Funktionen `dsm_server_close` und `dsm_client_close` erweitert. Im folgenden wird die Implementierung jeder Funktion im Modul mad_msg besprochen.

```
int dsm_server_open(int type, int *port, int broadcast,
                    struct sockaddr_in *gaddr, int bufsize)
```

MADELEINE baut bereits während der Initialisierung intern Verbindungen zwischen den einzelnen Prozessen auf. Daher gibt diese Funktion nur den Wert `*port` zurück. Die anderen Parameter werden nicht verwendet.

```
int dsm_server_accept(int s, int type, int port, int broadcast,
                      struct sockaddr_in *gaddr, int *msglen,
                      char **buf, int maxlen)
```

Empfängt den Kopf (*Header*) einer MADELEINE-Nachricht durch Aufruf von `mad_receive`. Diese Funktion blockiert nur den aufrufenden Thread. Somit zeigt die MADELEINE-basierte Funktion das gleiche Verhalten, wie die ursprüngliche Version. Die Funktion gibt den Wert des Parameters `port` zurück. Die anderen Parameter werden nicht verwendet.

```
int dsm_read(int s, int type, char *buf, int len, struct sockaddr *saddr)
```

Die Daten werden mit der Funktion `mad_unpack_byte` aus dem bereits empfangenen Kopf der Nachricht an die angegebene Stelle (`buf`,`len`) kopiert. Für diese Operation wird der Modus MAD_BY_COPY verwendet. Wurden die Daten nicht im Kopf übertragen, werden die Adresse (`buf`) und die Größe (`len`) in einer Empfangsstruktur gespeichert. Diese Informationen werden bearbeitet, wenn die Daten empfangen worden sind (siehe `dsm_server_close`). Rückgabewert ist der Parameter `len`. Die Parameter `s`, `type` und `saddr` werden nicht verwendet.

```
int dsm_server_close(int s)
```

Schließt das Empfangen einer Nachricht durch Aufruf von `mad_recvbuf_receive` ab. Diese Funktion empfängt den Teil der Nachricht, der nicht mit dem *Header* übertragen wurde und benutzt die Informationen vorangegangener `dsm_read`-Aufrufe, um die Daten an die richtigen Stellen im Speicher zu kopieren. Gibt immer den Wert 0 zurück. Der Parameter `s` wird nicht verwendet.

`int dsm_client_open(int type, int port, int broadcast, char *dest,`
`                    struct sockaddr_in *gaddr)`

Initialisiert einen Sendepuffer durch Aufruf der MADELEINE-Funktion `mad_sendbuf_init`. Die Identifikationsnummer des Empfängers (`port`) wird an die MADELEINE-Funktion übergeben. Der Rückgabewert der Funktion ist ebenfalls `port`. Die anderen Parameter werden nicht verwendet.

`int dsm_write(int s, int type, char *buf, int len, struct sockaddr *saddr)`

Die Daten des Puffers (`buf`,`len`) werden durch die Funktion `mad_pack_byte` in den Sendepuffer übertragen. Für diese Operation wird der Modus MAD_BY_COPY verwendet. Die Daten werden noch nicht zum Empfänger übertragen. Dies ermöglicht weitere Aufrufe von `dsm_write`, falls sich die Nachricht aus mehreren Teilen zusammensetzt. Der Rückgabewert ist der Parameter `len`. Die Parameter `s`, `type` und `saddr` werden nicht verwendet.

`int dsm_client_close(int s)`

Schließt das Senden einer Nachricht durch Aufruf der Funktion `mad_sendbuf_send` ab. Die Verbindung bleibt erhalten, da sie von MADELEINE verwaltet wird. Der Rückgabewert ist immer Null. Der Parameter `s` wird nicht verwendet.

`int dsm_close(int s)`

Diese Funktion wird nicht mehr benötigt, da ihre Aufrufe entweder durch `dsm_server_close` oder `dsm_client_close` ersetzt wurden.

6.2.1 TCP-Unterstützung

Für die TCP-Unterstützung waren nur geringfügige Modifikationen notwendig. Beim Senden und Empfangen von Nachrichten wird an einigen Stellen die MARCEL-Funktion `marcel_givehandback` verwendet, um ein *BusyWait* zu verhindern, wenn eine Operation mehrmals hintereinander fehlschlägt. Mit `marcel_givehandback` teilt der aktive Thread dem System mit, daß der Rest der Zeitscheibe von einem anderen Thread benutzt werden kann. Dies funktioniert nur mit präemptiven Scheduling. DSM-Threads benutzt striktes Prioritäten-Scheduling. Daher kann diese Funktion nicht einfach durch die Pthreads-Funktion `sched_yield` ersetzt werden. Eine einfache Lösung dieses Problems ist es, den aktiven Thread für eine kurze Zeitspanne zu suspendieren, daß heißt anderen Threads die Möglichkeit zu geben, aktiv zu werden.

Das Warten wird mit Hilfe der Funktion `pthread_cond_timedwait` implementiert. Da die Bedingungsvariable nie signalisiert wird, kehrt die Funktion zurück, wenn der angegebene Zeitpunkt erreicht bzw. überschritten ist. Der Zeitpunkt muß absolut angegeben werden. Bei der Berechnung des Zeitpunktes muß beachtet werden, daß das System eine geringere Zeitauflösung besitzt als gewünscht ist. Beträgt die Auflösung beispielsweise 10 ms, so wartet das System mindestens 10 ms, auch wenn eine geringere Wartezeit angegeben wurde. Sei AWZ die angegebene Wartezeit, dann gilt für die tatsächliche Wartezeit (TWZ):

$$TWZ \geq (n+1) \cdot 10 \qquad \text{mit} \quad n = \frac{AWZ}{10}, \quad n \text{ auf die nächstkleinere natürliche Zahl abgerundet}$$

6.2.2 BIP-Unterstützung

Für die BIP-Unterstützung waren mehrere umfangreiche Modifikationen notwendig. Ein Problem war die Verwendung von `marcel_givehandback`. Während diese Funktion in der TCP-Unterstützung nur aufgerufen wurde, wenn ein Fehler vorausgegangen war, wird sie hier im Zusammenhang mit Polling eingesetzt. Polling wird verwendet, um festzustellen, ob eine asynchrone Sende- oder Empfangsoperation beendet ist.

Ein weiteres Problem war der ursprüngliche Algorithmus für das Senden langer Nachrichten. Dieser erfordert eine Synchronisation zwischen Sender und Empfänger, wobei der Sender nach dem Absetzen einer REQUEST-Nachricht auf eine Bestätigung (ACK) vom Empfänger wartet, bevor er die restlichen Daten in einer DATA-Nachricht sendet (siehe Abbildung 3.4(b-d)). Der sendende Thread wartet dabei selbst auf die Bestätigung. Da in DSM-Threads nur ein Thread - der Kommunikationsserver - Nachrichten empfangen darf, mußte die Implementierung entsprechend angepaßt werden.

Schließlich konnte es zu einer Blockierung (*Deadlock*) kommen, wenn einem Sender *Credits* zum Senden fehlen (vgl. Erläuterung des *Credit*-Mechanismus in Abschnitt 3.2.2). Für die Lösung dieses Problems werden zwei Varianten vorgestellt.

Polling-Problem

In einer Multi-Threading-Umgebung kann man mit BIP nur *Polling* einsetzen, um nicht-blockierendes Senden und Empfangen zu implementieren. Die Betrachtungen beschränken sich an dieser Stelle auf das Empfangen von Nachrichten, sind aber auch auf das Senden übertragbar. Die blockierenden Empfangsoperationen halten den gesamten Prozeß an, d.h. alle Threads sind bis zum Abschluß der Operation blockiert, auch wenn ihre Ausführung gar nicht vom Ergebnis der Operation abhängig ist.

Um diese Art von blockierenden Zuständen zu vermeiden, muß der Thread, der eine Nachrichtenoperation durchführt, nicht-blockierende Funktionen verwenden. Mit der Funktion `bip_tirecv(tag,buf,len)` wird eine Empfangsoperation eingeleitet. Der Parameter `tag` gibt den Kanal, an welchem die Nachricht erwartet wird, an. Die Parameter `buf` und `len` bezeichnen einen Puffer, in den die Nachricht abgelegt werden soll. Die Funktion gibt ein *Handle* zurück, das im nächsten Schritt verwendet wird.

Das Ende einer Empfangsoperation muß durch Testen ermittelt werden. Hierzu wird das *Handle* benutzt. Die Funktion *bip_rtestx(handle)* realisiert das Testen. Das Testen muß solange wiederholt werden, bis es erfolgreich war, d.h. die Funktion wird in einer Schleife aufgerufen. Um ein *BusyWait* zu verhindern, muß der Thread zwischen zwei Testoperationen suspendiert werden. So können andere Threads aktiv werden. Die Suspendierung eines Threads kann nicht Funktionen, wie z.B. `sched_yield()` oder `marcel_givehandback()`, verwenden, da diese nur einen Kontextwechsel zwischen Threads gleicher Priorität auslösen. Da aber alle Nachrichten vom Kommunikationsserver, dem Thread mit der höchsten Priorität in DSM-Threads, empfangen werden, könnte die Kontrolle nie an einen Benutzer-Thread abgegeben werden. Daher muß die Funktion `pthread_cond_timedwait` benutzt werden. Die Funktion erlaubt die Suspendierung eines Threads für eine kurze Zeit und bewirkt einen Kontextwechsel zu Threads mit anderen Prioritäten.

Das Problem bei dieser Funktion ist, daß die Dauer der Suspendierung nicht beliebig klein gewählt werden kann. Unter Linux 2.2.12 konnte eine minimale Dauer von 10 ms eingestellt werden. Dies führt dazu, daß die niedrigen Latenzzeiten für kleine BIP-Nachrichten sehr gering, nicht an das DSM-System weitergegeben werden. Obwohl sich die Polling-Frequenz mittels der *Real Time Clock (RTC)* (Echtzeituhr) deutlich erhöhen lassen, können auch so die möglichen Latenzzeiten mittels Polling nicht erreicht werden.

Die RTC erlaubt die Generierung periodischer Interrupts. Die Frequenz kann auf einen Wert 2^k Hz mit $1 \leq k \leq 13,\ k \in \mathcal{N}$ $(2^{13} = 8192)$ eingestellt werden. Um Werte ab einer Frequenz von 128 Hz ($k = 7$) zu benutzen, muß der Anwender Superuser-Rechte besitzen. In Abbildung 6.2.2 wird die Anwendung der RTC mit einem kurzen Beispielprogramm illustriert.

Zunächst wird die Gerätedatei `/dev/rtc` geöffnet (Zeile 13). Anschließend wird die eingestellte Frequenz ermittelt (Zeile 15). Mit der äußeren Schleife (Zeilen 17 - 29) werden für jede Frequenz von 16 Hz bis 8192 Hz jeweils 40 Interrupts mit der inneren Schleife (Zeilen 21 - 28) abgefangen. Zu Beginn jedes Durchlaufs der äußeren Schleife wird die Frequenz neu gesetzt (Zeile 19). Die innere Schleife führt immer die drei Schritte

1. Einschalten der periodischen Interrupts (Zeile 23)

2. Warten bis zur Generierung des nächsten Interrupts mittels blockierender `read`-Funktion (Zeile 25). Gelesen wird aus der Gerätedatei, die in Zeile 13 geöffnet wurde.

3. Aussschalten der periodischen Interrupts (Zeile 27)

Die Version 1.09 des RTC-Treibers unterstützte nicht die Verwendung von FSU-Threads. Dank eines Patches für den Linux-Kernel (Version 2.2.12) und einer modifizierten Version der FSU-Pthreads, die von Frank Müller zur Verfügung gestellt wurden, konnte die RTC schließlich eingesetzt werden. In Abschnitt 7.2.2 wird der Nachrichtenaustausch analysiert. Trotz erhöhter Polling-Frequenz ist die erreichbare Leistungsfähigkeit enttäuschend.

Eine konsequente Lösung des Polling-Problems wäre die Unterstützung von Interrupts (Notifikation durch Signale) durch BIP.

Lange Nachrichten

Das Senden langer Nachrichten erfordert eine Synchronisation zwischen Sender und Empfänger (vgl. Erläuterungen in Abschnitt 3.2.2).

Gemäß dem Design des neuen Systems (siehe Abbildung 6.5), empfängt der sendende Thread nun nicht selbst die Bestätigung, es sei denn, er ist der Kommunikationsserver. Stattdessen wartet er an einer Bedingungsvariablen auf eine Bestätigung. Die Bedingungsvariable wird vom Kommunikationsserver signalisiert, sobald die Bestätigung bei ihm eingegangen ist. Jeder Thread benutzt eine eigene Bedingungsvariable für das Warten. Der sendende Thread sendet im Kopfteil der REQUEST-Nachricht seine Thread-ID, die vom Empfänger mit der Bestätigung (ACK) zurückgesendet wird. In jedem Prozeß wird eine Liste, die für jeden Thread ein Element enthält, verwaltet. Diese Elemente ermöglichen die Zuordnung einer Bedingungsvariable

```
#include <stdio.h>
#include <linux/mc146818rtc.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[]) {

  int i, fd;
  unsigned long tmp, data;

  fd = open ("/dev/rtc", O_RDONLY);

  ioctl(fd, RTC_IRQP_READ, &tmp);

  for (tmp=16; tmp<=8192; tmp*=2) {

    ioctl(fd, RTC_IRQP_SET, tmp);

    for (i=1; i<41; i++) {

      ioctl(fd, RTC_PIE_ON, 0);

      read(fd, &data, sizeof(unsigned long));

      ioctl(fd, RTC_PIE_OFF, 0);
    }
  }

  close(fd);
  return 0;
}
```

Abbildung 6.4: Anwendung der *Real Time Clock*

zu dem Thread, der sie für das Warten auf die Bestätigung einsetzt. Mit Hilfe dieser Angaben - Thread-ID im Nachrichtenkopf und Liste - kann der Kommunikationsserver genau diejenige Bedingungsvariable signalisieren, an welcher der Thread wartet.

Der Kommunikationsserver führt die Funktion `receive_from_network` aus, wenn er auf eine Nachricht wartet. Diese Funktion wurde so erweitert, daß periodisch überprüft wird, ob eine Bestätigung eingetroffen ist. Hat der Kommunikationsserver eine Bestätigung erhalten, ermittelt er, für welchen Thread sie bestimmt ist und signalisiert die entsprechende Bedingungsvariable, es sei denn, er selbst ist der Empfänger.

Besteht die MADELEINE-Nachricht aus mehreren Vektoren, so erwartet der sendende Thread eine Bestätigung, bevor der Vektor 1 gesendet wird. Diese Bestätigung wird auf die gleiche Weise wie die erste empfangen.

Fehlende Credits

Das Senden von Nachrichten in MADELEINE erfordert das Vorhandensein von *Credits* für den Empfänger (siehe Erläuterungen in Abschnitt 3.2.2). Sendet ein Thread viele Nachrichten in kurzer Zeit an denselben Prozeß, kann es vorkommen, daß der Sender keine *Credits* mehr besitzt. In diesem Fall blockiert MADELEINE den sendenden Thread und wartet solange, bis der Empfängerprozeß *Credits* zurückgeschickt hat. Für das kombinierte System würde dies bedeuten das, daß ein Aufruf der Funktion `dsm_commSend` blockiert. Dieses Verhalten könnte zu Verzögerungen bei der Bearbeitung von Nachrichten führen, wenn beispielsweise viele *Worker-Threads* blockiert werden, oder gar zu einem *Deadlock* führen, wenn der Kommunikationsserver selbst blockiert wird (vgl. Erläuterungen in Abschnitt 3.2.2). Das Problem kann dadurch gelöst werden, daß vor dem Senden einer Nachricht überprüft wird, ob noch ein *Credit* für den Empfänger vorhanden ist (vgl. Erläuterungen in Abschnitt 5.2 und Abbildung 5.1). Im folgenden werden zwei Möglichkeiten der Implementierung des veränderten Sendens vorgestellt. Zunächst wird eine Variante erläutert, die Änderungen

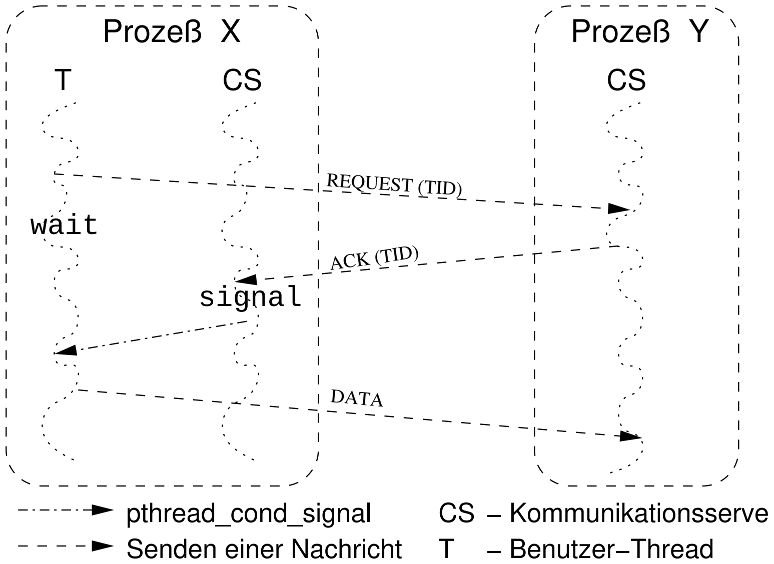

Abbildung 6.5: Austausch langer BIP-Nachrichten im neuen System

in DSM-Threads und MADELEINE erfordert. Für die zweite Variante waren nur Änderungen in MADELEINE notwendig, daher wurde diese Variante gewählt.

1. Variante: Die Funktion `dsm_commSend` testet zunächst, ob für den Empfänger noch mindestens ein *Credit* vorhanden ist. Ist dies der Fall, arbeitet die Funktion weiter wie bisher. Andernfalls wird die Nachricht durch Aufruf der Funktion `DelayMessage` in eine Warteschlange eingereiht. Die Funktion `dsm_commSend` wird anschließend sofort beendet, daß heißt, es wird so getan, als ob die Nachricht bereits gesendet wurde. Bei dem Test, ob noch *Credits* vorhanden sind, muß eine RACE-Bedingung berücksichtigt werden. Diese kann auftreten, wenn nach einem positivem Testergebnis der aktuelle Thread durch einen höherpriorisierten Thread (z.B. Kommunikationsserver) unterbrochen wird und der unterbrechende Thread den letzten *Credit* verbraucht.

Die Warteschlange ist durch eine doppelt verkettete Liste implementiert. In Abbildung 6.6 ist die Deklaration eines Listenelementes dargestellt. Die Variablen `buf` und `len` speichern den Inhalt der Nachricht. `receiver` ist ein Verweis auf den Empfänger der Nachricht.

```
typedef struct dsm_DelayedMessage {
  char *buf;
  int  len;
  dsm_Node_t *receiver;
  long index;
  struct dsm_DelayedMessage *prev;
  struct dsm_DelayedMessage *next;
} dsm_DelayedMessage_t;
```

Abbildung 6.6: Deklaration eines Listenelementes der Warteschlange für Nachrichten

Die Warteschlange wird von einem neuen Thread des DSM-Systems bearbeitet. Dieser Server (auch „Delayed Messages Server“ (DMS) genannt wartet auf das Eintreffen von *Credits* mit Hilfe einer Bedingungsvariablen. Erhält der Kommunikationsserver *Credits* zurück, signalisiert er den Warteschlangenserver. Dieser traversiert die Warteschlange und stellt dabei fest, ob für verzögerte Nachrichten *Credits* eingetroffen sind. Ist dies der Fall, werden soviele Nachrichten wie möglich gesendet. Gesendete Nachrichten werden aus der Warteschlange entfernt.

Die Signalisierung muß durch MADELEINE-Funktionen vorgenommen werden, da es keine Möglichkeit gibt, das Empfangen von *Credits* außerhalb von MADELEINE festzustellen.

2. Variante: Bei dieser Variante bleibt die Funktion `dsm_commSend` unverändert. Der Algorithmus - Testen, eventuelles Einfügen der Nachricht in eine Warteschlange und Signalisieren eines speziellen Threads - wird wie bei der ersten Variante implementiert. Alle Veränderungen werden aber in MADELEINE vorgenommen.

```
typedef struct mad_DelayedMessage {
  char *buf;
  int  len;
  int  receiver;
  struct mad_DelayedMessage *prev;
  struct mad_DelayedMessage *next;
} mad_DelayedMessage_t;
```

Abbildung 6.7: Deklaration eines Listenelementes der Warteschlange für Nachrichten

In Abbildung 6.7 ist die Deklaration für ein Element der Warteschlange dargestellt. Die Elemente `buf` und `len` wurden bereits bei der 1. Variante verwendet. Der Empfänger wird durch die Zahl `receiver` angegeben.

Die Warteschlange wird von einem neuen Thread, der während der Initialisierung von MADELEINE gestartet wird, bearbeitet. Der Thread führt den *Delayed Messages Server* aus. Dieser wartet auf das Eintreffen von *Credits* mit Hilfe einer Bedingungsvariablen. Erhält der Kommunikationsserver *Credits* zurück, wird diese Bedingungsvariable signalisiert. Dies wird von intern MADELEINE-Funktionen geleistet. Nach seiner Signalisierung traversiert der *Delayed Messages Server* die Liste der gespeicherten Nachrichten und sendet Nachrichten, für deren Empfänger *Credits* eingetroffen sind. Erfolgreich gesendete Nachrichten werden aus der Warteschlange entfernt.

Mit der zweiten Variante konnte eine enge Verzahnung des DSM-Systems mit MADELEINE, die durch Implementierung der ersten Variante notwendig geworden wäre, verhindert werden. Die vorgestellte Lösung fügt zusätzliche Funktionalität in MADELEINE ein.

Eine weitere Variante wäre, daß der Kommunikationsserver nur noch DSM-Nachrichten empfängt, aber nicht mehr selbst bearbeitet. Insbesondere dürfte er keine Nachrichten mehr senden, da dies (ohne die oben vorgestellte Lösung) zu einer Blockierung führen kann. Bei dieser Variante wären die oben besprochenen Änderungen nicht notwendig. Würde die Bearbeitung einer Nachricht das Senden einer Antwort erfordern, müßte die Bearbeitung dieser Nachricht von einem *Worker* übernommen werden. Diese Variante wurde dennoch nicht implementiert, weil das Problem sehr selten, selbst bei großer Anzahl nebenläufiger Benutzer-Threads, innerhalb eines Prozesses auftritt. Eine Änderung der Bearbeitung von Nachrichten wäre für diese wenigen Fälle eine zu große Einschränkung der Flexibilität des DSM-Systems. Im Gegensatz dazu vergleiche man die notwendigen Änderungen bei der Initialisierung und bei der Erzeugung von entfernten Threads, ohne die eine Anwendung von MADELEINE gar nicht möglich gewesen wäre.

6.3 Erzeugung entfernter Threads

In den Abschnitten 4.2.2 und 5.3 wurden zwei Varianten für die nachrichtenbasierte Erzeugung entfernter Threads erläutert. Hier wird die Implementierung der Variante aus Abschnitt 5.3, die mit drei Nachrichten auskommt, beschrieben.

Die Erzeugung eines entfernten Threads beruht auf dem Austausch von Nachrichten zwischen den Prozessen (vgl. Abbildung 5.2), die bereits seit der Startphase existieren. Die Funktion `dsm_thread_create` wurde an das neue Modell angepaßt. An dieser Stelle sollen nur die Erweiterungen bzw. Veränderungen der Implementierung erläutert werden (vgl. Erläuterung der Implementierung des ursprünglichen Modells in Abschnitt 2.2.3).

In der folgenden Tabelle werden die Nachrichtentypen und die dazugehörigen Behandlungsroutinen vorgestellt.

Die Deklaration der Funktion `dsm_thread_create` wurde beibehalten (vgl. Abschnitt 2.2.3). Am Anfang bereitet die Funktion ihre Parameter für die Übertragung zum Zielprozeß vor. Anschließend wird für den aktuellen Thread ein Element in die Liste, die Informationen über Thread-Erzeugungen enthält, eingefügt. Diese Informationen werden später bei der Signalisierung des erzeugenden Threads durch den Kommunikationsserver benutzt. In Abbildung 6.8 ist die Deklaration eines Listenelementes dargestellt. Mit der Variable `thread` wird dieses Listenelement mit einem Thread assoziiert. Die Bedingungsvariable `cond`, der dazugehörige Lock `mutex` und die Abbruchbedingung `thread_create_ack` werden für das Warten verwendet. Die

Nachrichtentyp / `Behandlungsroutine`	Kurzbeschreibung
THREAD_CREATE / `dsm_ThreadCreateServer`	Die Nachricht wird vom erzeugenden Thread an den Zielprozeß geschickt. Die Behandlungsroutine erzeugt den Thread und sendet eine Nachricht vom Typ THREAD_CREATED an den *Masterprozeß*.
THREAD_CREATED / `dsm_ThreadCreatedServer`	Die Nachricht wird von dem Zielprozeß (CS), in welchem der neue Thread erzeugt wurde, an den *Masterprozeß* geschickt. Die Behandlungsroutine erhöht den Zähler für Benutzer-Threads und schickt eine Nachricht vom Typ THREAD_CREATED_ACK an den erzeugenden Thread.
THREAD_CREATED_ACK / `dsm_ThreadCreatedAckServer`	Die Nachricht wird vom *Masterprozeß* (CS) an den erzeugenden Thread gesendet. Die Behandlungsroutine signalisiert den erzeugenden Thread.

Tabelle 6.1: Nachrichtentypen der veränderten Thread-Erzeugung und ihre Behandlungsroutinen

Variable `thread_create_result` speichert das Ergebnis der Thread-Erzeugung. Der Zeiger `next` dient zur einfachen Verkettung der Listenelemente.

```
typedef struct dsm_CreateAckHelp {
  pthread_t       pthread;
  pthread_cond_t  cond;
  pthread_mutex_t mutex;
  int thread_create_ack;
  int thread_create_result;
  struct dsm_CreateAckHelp *next;
} dsm_CreateAckHelp_t;
```

Abbildung 6.8: Deklaration eines Listenelementes für die Verwaltung von Informationen bei der Erzeugung entfernter Threads

Nun sendet der Thread eine THREAD_CREATE-Nachricht an den Zielprozeß. Danach wartet er an einer Bedingungsvariablen auf eine Bestätigung vom *Masterprozeß*. Erhält der Zielprozeß die THREAD_CREATE-Nachricht, wertet die Behandlungsroutine `dsm_ThreadCreateServer` die Parameter aus und erzeugt durch Aufruf von `pthread_create` einen neuen Thread. Anschließend wird dem *Masterprozeß* eine Benachrichtigung (THREAD_CREATED) über die Erzeugung eines neuen Threads geschickt.

Der *Master* inkrementiert den Zähler für Benutzer-Threads (`dsm_UserThreadsCounter`), wenn er die THREAD_CREATED-Nachricht erhalten hat. Dieser Zähler wird bei der Terminierung verwendet (siehe Abschnitt 6.4). Schließlich sendet der *Master* eine Benachrichtigung (THREAD_CREATED_ACK) an den erzeugenden Thread. Die Benachrichtung wird vom Kommunikationsserver bearbeitet. Mit Hilfe der erwähnten Liste werden der erzeugende Thread und die Bedingungsvariable, an welcher der Thread wartet, ermittelt. Die Behandlungsroutine `dsm_ThreadCreatedAckServer` speichert das Ergebnis der Thread-Erzeugung in dem Listenelement des Threads und signalisiert die Bedingungsvariable, an welcher er wartet. Der erzeugende Thread beendet die Funktion durch Rückgabe des Ergebnisses.

6.4 Terminierung

Die ursprüngliche Terminierung (vgl. Abschnitt 2.2.4) wurde gemäß des neuen Designs erweitertet (vgl. Erläuterungen zum Design in Abschnitt 5.4). Terminiert ein Benutzer-Thread, so wird automatisch die Funktion `dsm_thread_exit` ausgeführt. Diese Funktion entfernt das Element des Threads aus der Liste, die bei der Erzeugung von Threads verwendet wird. Die Datenstrukturen des Elementes (vgl. Abbildung 6.8) werden freigegeben.

Bevor die Funktion beendet wird, wird eine Nachricht vom Typ THREAD_TERMINATED an den *Master* gesendet. Falls der terminierte Benutzer-Thread im *Masterprozeß* ausgeführt wurde, ist eine andere Behandlung erforderlich. Hier wird überprüft, ob es sich bei dem beendeten Thread um den *Main-Thread* oder einen anderen Benutzer-Thread handelt. Im *Main-Thread* wird die Funktion `dsm_thread_exit` an dieser Stelle beendet. In einem Benutzer-Thread wird der Zähler für die Anzahl der Benutzer-Threads um Eins dekrementiert. Erreicht dieser Zähler nach dem Dekrementieren den Wert Null, wird die Bedingungsvariable `dsm_UTC_cond` signalisiert.

Erreicht den *Master* eine Nachricht vom Typ THREAD_TERMINATED, so führt er die Behandlungsroutine `dsm_ThreadTerminatedServer` aus. Sie dekrementiert den Zähler für die Anzahl der Benutzer-Threads und signalisiert die Bedingungsvariable `dsm_UTC_cond`, falls der Wert des Zählers nach dem Dekrementieren Null ist.

Diese beiden Erweiterungen - Ausführung der Funktion `dsm_thread_exit` nach der Terminierung eines Benutzer-Threads und asynchrone Behandlung der THREAD_TERMINATED-Nachrichten - ermöglichen eine korrekte Terminierung durch die Funktion `dsm_exit`. Die Terminierung soll nur durchgeführt werden, wenn alle Benutzer-Threads beendet wurden. Wie in der ursprünglichen Implementierung wird am Schluß der Funktion `dsm_init` (außer im *Masterprozeß*) die Funktion `dsm_exit` aufgerufen. Im *Masterprozeß* wird sie erst am Ende des *Main-Threads* aufgerufen. Die Funktion *dsm_exit* wurde an ihrem Anfang um die Überwachung der Variable `dsm_UserThreadsCounter`, die im *Master* die Anzahl aller Benutzer-Threads in allen DSM-Prozessen speichert und in den anderen Prozessen als Abbruchbedingung für ein bedingtes Warten dient, erweitert.

Wenn alle Benutzer-Threads terminiert haben und auch der *Masterprozeß* die Funktion `dsm_exit` aufgerufen hat, sendet er eine FINISH_REMOTE-Nachricht an alle Prozesse. In diesen signalisiert der Kommunikationsserver den *Main-Thread*, der an einer Bedingungsvariable auf das Eintreffen dieser Nachricht gewartet hat. Dieses Warten ist in der Funktion `dsm_exit` implementiert. Anschließend können alle Prozesse mit der ursprünglichen Terminierung fortfahren.

Kapitel 7

Funktionalitätstests und Leistungsmessungen

Dieses Kapitel beschäftigt sich mit der Analyse des neuen Systems. Hierzu werden Funktionalitätstests und Leistungsmessungen durchgeführt. Mit den Funktionalitätstests soll die Frage, ob das System im wesentlichen die gleichen Eigenschaften wie vor der Kombination besitzt, geklärt werden. Die Leistungsmessungen dienen dazu, festzustellen, ob und wie sich die Leistungsfähigkeit des Systems verändert hat. Anhand der Messungen kann man Schwachstellen ausfindig machen.

7.1 Funktionalitätstests

Dieser Abschnitt beschäftigt sich damit, wie die Funktionalität des neuen Systems überprüft wurde. Für das DSM-System existieren zahlreiche Testprogramme, mit denen man verschiedene Eigenschaften des Systems testen kann. Bei der Kombination von DSM-Threads und MADELEINE waren nicht alle Komponenten von DSM-Threads betroffen. Daher wurden für die Tests zwei Programme, die die geänderten Komponenten testen, ausgewählt. Die Funktionalität wurde mit den Testprogrammen `test_create` und `test_terminate` überprüft. Mit beiden Programmen können die Kommunikation und die Implementierung des neuen Modells der Erzeugung entfernter Threads getestet werden. Mit `test_terminate` kann zusätzlich die korrekte Terminierung überprüft werden.

7.1.1 Das Programm `test_create`

Dem Programm wird entweder eine Liste mit Rechnernamen oder eine Zahl als Argument übergeben. So wird festgelegt, auf welchen Rechnern (Argument ist Liste) oder wieviele Threads gestartet werden sollen. Diese Threads werden durch die Funktion `dsm_thread_create` erzeugt. Im Originalsystem werden sie also in neuen Prozessen auf einem Rechner gestartet, wobei pro erzeugtem Thread ein neuer Prozeß erzeugt wird. Im kombinierten System existieren die Prozesse auf den Rechnern bereits. In einem Prozeß können auch mehrere Threads erzeugt werden. Die Threads führen eine Funktion aus, die lesend und schreibend auf gemeinsame Variablen zugreift. Diese Zugriffe erfordern den Austausch von Nachrichten. Auch die Terminierung basiert auf dem Austausch von Nachrichten.

Das vollständige Programm ist in Anhang A.1 abgebildet. Das Hauptprogramm initialisiert zunächst das DSM-System (Zeilen 40 und 41). Anschließend werden gemäß den Programmparametern die Threads erzeugt (Zeilen 51 - 60). Nun werden im Hauptprogramm (Zeilen 63 - 74) und in den erzeugten Threads (Zeilen 18 - 32) verschiedene Variablen ausgegeben. Es gibt lokale Variablen - `x` lokal in einer Funktion und `y` lokal in einem Prozeß. Neben diesen gibt es die gemeinsamen Variablen `z` und `f`. Die Ausgabe dieser Variablen erfordert den Austausch von Nachrichten. Werden ihre Werte korrekt ausgegeben, kann davon ausgegangen werden, daß der Nachrichtenaustausch funktioniert.

Durch die Funktion `dsm_exit` wird die Terminierung eingeleitet. An deren Ende wird eine Statistik über die Zugriffe auf die verwalteten Seiten ausgegeben. Wenn diese Statistik in der Ausgabe erscheint, kann man davon ausgehen, daß die Terminierung korrekt abgelaufen ist.

Daß der Nachrichtenaustausch funktioniert, kann man im neuen System bereits daran erkennen, daß ein neuer Thread erzeugt wird, da hierzu Nachrichten ausgetauscht werden.

Die Multi-Threading-Fähigkeiten lassen sich am besten überprüfen, wenn man hinreichend viele Threads (mehr als 10 pro Prozeß) erzeugt. So wurde beispielsweise festgestellt, daß es zu einer Blockierung aufgrund fehlender *Credits* bei Benutzung von BIP kommen kann (vgl. Erläuterungen in Abschnitt 5.2).

Tests mit `test_create`

Das Programm wurde mit unterschiedlichen Konfigurationen in der Anzahl der Rechner, der Anzahl der erzeugten Prozesse sowie der Anzahl der erzeugten Threads (beim Testen des neuen Systems) getestet. Die Multi-Threading-Fähigkeiten konnten, wie im letzten Abschnitt skizziert wurde, getestet werden. Beim Testen des neuen Systems wurde neben TCP auch die Kommunikationsstruktur BIP berücksichtigt.

7.1.2 Das Programm `test_terminate`

Mit diesem Programm wird die Terminierung überprüft. Die Prozesse, die bereits während der Startphase erzeugt werden, dürfen erst terminieren, wenn es keine Benutzer-Threads mehr gibt. Sonst könnten Daten verloren gehen. Das Hauptprogramm startet nur einen Thread. Jeder neue Thread pausiert zunächst für eine zufällige kurze Zeit. Anschließend erzeugt er einen neuen Thread und erhöht den Wert einer gemeinsamen Variable. Der einzige Parameter des Programmes gibt an, wieviele Threads auf diese Art und Weise erzeugt werden sollen. Würde der Prozeß, der die Variable zuletzt geändert hat, zu früh terminieren, hätte dies Auswirkungen auf den Wert dieser Variablen.

Der Quelltext des Programmes ist in Anhang A.2 abgebildet. Zunächst wird das DSM-System initialisiert (Zeilen 35 und 36). Anschließend ruft der *Main-Thread* im *Masterprozeß* die Routine `new_thread` mit dem Parameter `num`, der die Anzahl der zu erzeugenden Threads angibt, auf (Zeile 43). Nachdem diese Funktion abgearbeitet wurde, wird durch den Aufruf der Funktion `dsm_exit` die Terminierung im *Masterprozeß* eingeleitet (Zeile 45).

Die Funktion `new_thread` gibt zunächst den Wert des Parameters `arg` aus (Zeilen 14 - 16). Der Wert `num`-1 gibt an, wieviele Threads noch erzeugt werden müssen. Nach der Ausgabe legt die Funktion eine kurze Pause ein (Zeile 18). Anschließend wird der Wert von `num` um Eins dekrementiert. Ist der neue Wert größer Null, wird mit `dsm_thread_create` ein neuer Thread in einem DSM-Prozeß erzeugt. Die Auswahl des Prozesses wird dem DSM-System überlassen (Zeilen 20 - 22).

Nun wird die gemeinsame Variable `counter` um Eins inkrementiert (Zeile 25). Zu Beginn ist sie mit Null initialisiert. Falls dieser Thread der letzte erzeugte ist (`num` == 0), wird der Wert der gemeinsamen Variable ausgegeben. Dieser Wert müßte mit dem Parameter, der bei Programmstart angibt, wieviele Threads erzeugt werden sollen, übereinstimmen.

Tests mit `test_terminate`

Das Programm wurde mit beiden Systemen (alt und neu) erfolgreich getestet. Neben der Anzahl der eingesetzten Rechner wurde die Anzahl der erzeugten Threads variiert. Bei den Tests mit dem neuen System wurden sowohl TCP als auch BIP berücksichtigt.

7.2 Leistungsmessungen

> *Die großen Leute haben eine Vorliebe für Zahlen. [...] wenn ihr ihnen sagt: Der Beweis dafür, daß es den kleinen Prinzen wirklich gegeben hat, besteht darin, daß er entzückend war, daß er lachte und daß er ein Schaf haben wollte; denn wenn man sich ein Schaf wünscht, ist es doch ein Beweis dafür, daß man lebt, dann werden sie die Achseln zucken und euch wie Kinder behandeln. Aber wenn ihr ihnen sagt: Der Planet, von dem er kam, ist der Planet B 612, dann werden sie überzeugt sein [...]*
> *Wir freilich, die wir wissen, was das Leben eigentlich ist, wir machen uns nur lustig über die albernen Zahlen.*
> (Aus „Der Kleine Prinz“, Antoine de Saint-Exupéry [25])

Der Schwerpunkt der Arbeit lag im Design und der Implementierung des kombinierten Systems. Daher wurden nur einige Leistungsmessungen durchgeführt, um „Zahlen“ für die Analyse der Leistungsfähigkeit zu ermitteln. Tatsächlich decken die Ergebnisse ein grundlegendes Problem auf, daß auftritt, wenn man moderne Kommunikationsarchitekturen wie z.B. BIP/Myrinet in DSM-Systemen, die Multi-Threading verwenden und eine gute Responsivität bei asynchroner Kommunikation erreichen sollen, einsetzt. Die maßgebliche Ursache für dieses Problem ist, daß Kommunikationssoftware wie z.B. BIP keine Notifikation durch Signale für die Erkennung von beendeten Sende- und Empfangsoperationen bereitstellt. Stattdessen muß Polling verwendet werden.

Die Leistungsmessungen wurden mit dem Programm *LU* der SPLASH-2–Benchmarksuite [26] durchgeführt. Das Programm berechnet die LU-Zerlegung einer Matrix. Die SPLASH-2–Benchmarksuite enthält

Programme aus dem naturwissenschaftlichen Bereich, den Ingenieurwissenschaften und der Bildverarbeitung. Diese Programme eignen sich für die Evaluation von verteilten Systemen mit virtuellem gemeinsamem Speicher.

Die SPLASH-2–Programme eignen sich nicht nur wegen ihrer großen Akzeptanz sondern auch aufgrund ihrer guten Portierbarkeit. Für die Verwendung in einem bestimmten System müssen lediglich einige Makros an das verwendete System angepaßt werden.

Beschreibung des Testprogrammes

Das Programm LU faktorisiert eine dichte Matrix in ein Produkt einer unteren und oberen Dreiecksmatrix. Die dichte $n \times n$ Matrix A wird in ein $N \times N$ Feld von Blöcken der Größe $B \times B$ zerlegt (mit $n = NB$). Kleine Blockgrößen B sorgen für eine gute Lastbalancierung. Um *false sharing* zu vermeiden, werden die Elemente eines Blockes in einem zusammenhängenden Speicherbereich angelegt.

7.2.1 Messungen und Ergebnisse

Die Messungen wurden auf einem Cluster von Rechnern durchgeführt. Die Rechner sind mit FastEthernet und Myrinet (LANai Version 4.03) verbunden und besitzen 128 MByte Hauptspeicher sowie einen Prozessor des Typs AMD Athlon 550 MHz. Es wurden die Kommunikationsarchitekturen TCP (ursprüngliches System), MADELEINE/TCP und MADELEINE/BIP evaluiert. Die Größe der Matrix betrug 512 x 512. Um *false sharing* zu vermeiden, wurde die Blockgröße mit 32 x 32 so gewählt, daß auf einer Speicherseite nie zwei verschiedene Blöcke liegen.

Zunächst wurde die Ausführungszeit des ganzen Programmes für die verschiedenen Kommunikationsarchitekturen ermittelt. Anschließend wurde der Nachrichtenaustausch genauer untersucht. Bei den Messungen mit MADELEINE/BIP wurden unterschiedliche Wartezeiten zwischen zwei Polling-Tests verwendet. Für jede Messung wurden zwei Rechner eingesetzt.

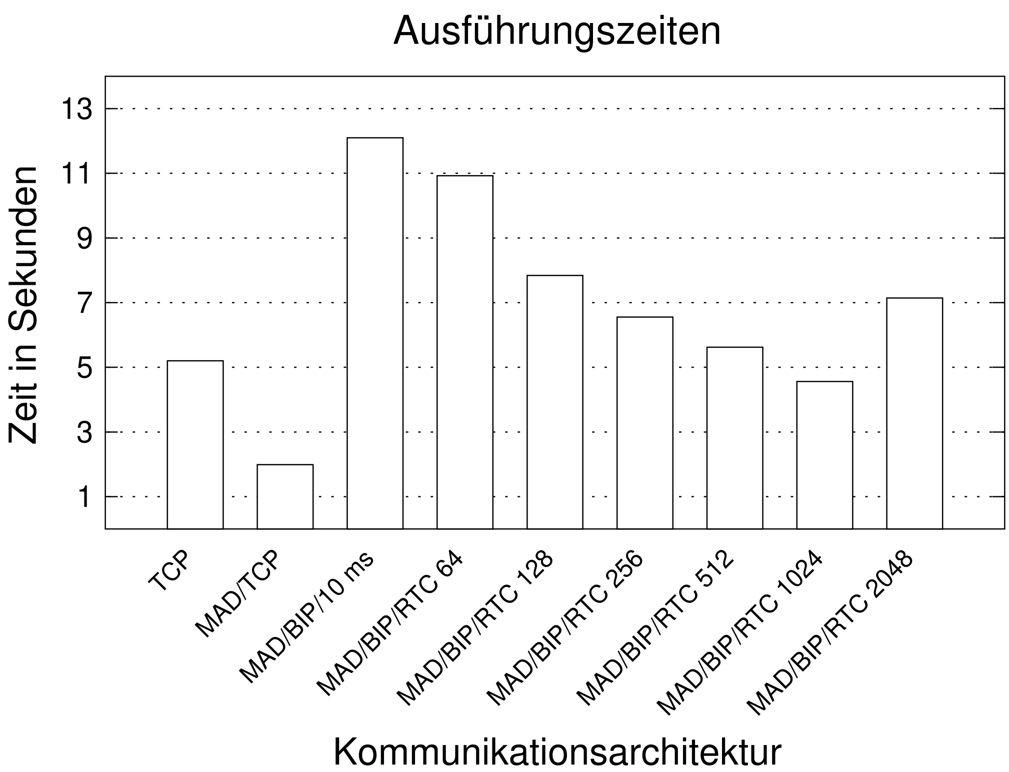

Abbildung 7.1: Ausführungszeiten des Programmes LU

Die Ausführungszeiten sind in Abbildung 7.1 graphisch dargestellt. In Tabelle 7.1 werden die exakten Zahlenwerte angegeben. Die Kombination von DSM-Threads mit MADELEINE/TCP (Bezeichnung MAD/TCP) erreicht den besten Wert. Der Unterschied zum ursprünglichen System (Bezeichnung TCP) ist enorm. Zu erwarten wäre eine etwas schlechtere Leistungsfähigkeit aufgrund des Mehraufwandes, der durch die Schicht *Programming Interface* von MADELEINE entsteht, gewesen. Die Ursache für diesen Unterschied liegt darin, daß MADELEINE/TCP zwischen den Prozessen permanente Verbindungen offenhält, während im ursprünglichen System für jede Sendeoperation eine neue Verbindung aufgebaut werden muß. Durch die Detailanalyse des Nachrichtenaustausches in Abschnitt 7.2.2 wird diese Beobachtung bestätigt.

Kommunikationsarchitektur	Zeit in s
TCP	5.20099
MAD/TCP	1.98746
MAD/BIP/10 ms	12.1001
MAD/BIP/RTC 64	10.9235
MAD/BIP/RTC 128	7.83841
MAD/BIP/RTC 256	6.55203
MAD/BIP/RTC 512	5.62220
MAD/BIP/RTC 1024	4.56091
MAD/BIP/RTC 2048	7.14121

Tabelle 7.1: Ausführungszeiten des Programmes LU

Die Kombination mit MADELEINE/BIP erreicht im besten Fall die Leistung des ursprünglichen Systems. Die Ursache liegt darin, daß bei der Verwendung von BIP Polling eingesetzt werden muß, um festzustellen, ob eine Sende- oder Empfangsoperation abgeschlossen ist. Zunächst wurde das Polling auf Basis der Funktion `pthread_cond_timedwait` implementiert. Mit dieser Funktion läßt sich eine Polling-Frequenz von maximal 1/10 ms erreichen (100 Hz). Die entsprechenden Messungen sind mit „MAD/BIP/10 ms" bezeichnet. Da das kombinierte System mit dieser Polling-Implementierung eine geringere Leistungsfähigkeit als das ursprüngliche System zeigte, wurde eine zweite Implementierung des Pollings auf Basis der *Real Time Clock (RTC)* (Echtzeituhr) untersucht. Mit dieser Implementierung kann eine Polling-Frequenz von maximal 8192 Hz erreicht werden. Es wurden Messungen mit Frequenzen von 64, 128, 256, 512, 1024 und 2048 Hz durchgeführt. Eine Messung wird durch „MAD/BIP/RTC X" bezeichnet, wobei X die Frequenz angibt. Es hat sich gezeigt, daß die Ausführungszeiten für Frequenzen bis 1024 Hz mit zunehmender Frequenz abnehmen. Höhere Frequenzen als 1024 Hz führen zu wieder ansteigenden Ausführungszeiten. Mit der niedrigsten getesteten Frequenz (64 Hz) wird in etwa die gleiche Ausführungszeit wie beim Polling auf Basis der Funktion `pthread_cond_timedwait` erreicht. Im besten Fall (1024 Hz) wird die Leistungsfähigkeit des ursprünglichen Systems etwas übertroffen.

Diese Beobachtung läßt sich erklären, wenn man betrachtet, wie Nachrichten mit MADELEINE/BIP ausgetauscht werden. BIP unterscheidet zwischen kurzen und langen Nachrichten[1]. Während kurze Nachrichten asynchron übertragen werden können, müssen sich Sender und Empfänger synchronisieren, um lange Nachrichten auszutauschen. Der Sender muß warten, bis der Empfänger bereit ist, eine Nachricht entgegenzunehmen. An dieser Stelle hat die Polling-Frequenz einen Einfluß darauf, wie schnell diese Synchronisierung zustande kommt. Je höher die Frequenz ist, um so schneller können sich Sender und Empfänger synchronisieren. Daß die Ausführungszeit ab einer Frequenz von 2048 Hz wieder ansteigt, liegt daran, daß für jeden Polling-Test Mehraufwand für Interruptbehandlung, Signalnotifikation und Kontextwechsel ensteht. Sei $v = o - s$ die Veränderung der Ausführungszeit, wobei o den Mehraufwand und s die Einsparungen durch beschleunigte Synchronisierung bezeichnen. Dann beschreibt die Fallunterscheidung (7.1) das Verhalten eines Programmes in Abhängigkeit der Relation von o zu s.

$$\begin{array}{lcl} s > o & : & \text{Die Ausführungszeit nimmt ab.} \\ s = o & : & \text{Die Ausführungszeit bleibt gleicht.} \\ s < o & : & \text{Die Ausführungszeit nimmt zu.} \end{array} \tag{7.1}$$

Im folgenden Abschnitt wird der Nachrichtenaustausch genauer untersucht.

7.2.2 Analyse des Nachrichtenaustausches

Zum besseren Verständnis der ermittelten Ausführungszeiten wurde eine Analyse des Nachrichtenaustausches durchgeführt. Dazu wurde das Senden, implementiert durch die Funktion `dsm_commSend`, genauer untersucht. Zunächst werden für jede Kommunikationsarchitektur geeignete Meßpunkte anhand der Implementierung definiert. Im Anschluß werden die ermittelten Meßwerte dargestellt und diskutiert.

Das Senden einer Nachricht wird durch die Funktion `dsm_commSend` implementiert. In Tabelle 7.2 ist die Implementierung dieser Funktion für jede Kommunikationsarchitektur skizziert. Die drei Schritte beim

[1] Die Grenze zwischen kurzen und langen Nachrichten wird durch die Konstante `BIPSMALLSIZE` festgelegt und liegt bei 1016 Bytes.

Senden unter MADELEINE/BIP sind nur bei langen BIP-Nachrichten notwendig. Passen die zu sendenden Daten in eine kurze BIP-Nachricht, ist nur der erste Schritt erforderlich. In allen drei Schritten muß das Ende der Operation mittels Polling festgestellt werden.

DSM-Threads–Funktion	TCP	MADELEINE/TCP	MADELEINE/BIP
`dsm_client_open`	`socket`, `bind`, `connect`	`mad_sendbuf_init`	`mad_sendbuf_init`
`dsm_write`	`write`	`mad_pack_byte`	`mad_pack_byte`
`dsm_client_close`	`close`	`mad_sendbuf_send` `writev`	`mad_sendbuf_send` 1. Sende REQUEST 2. Warte auf ACK 3. Sende DATA

Tabelle 7.2: Implementierung der Funktion `dsm_commSend` in Abhängigkeit der Kommunikationsarchitektur

Gemessen wurden die Ausführungszeiten für die ganze Funktion, für `dsm_client_open`, für die Aufrufe von `dsm_write` (Summe) und `dsm_client_close`. Bei den ermittelten Ausführungszeiten muß berücksichtigt werden, daß die Übertragung der Daten[2] im ursprünglichen und im kombinierten System durch verschiedene Funktionen realisiert wird. Im ursprünglichen System (TCP) wird die Übertragung durch die Funktion `dsm_write` erledigt. Im kombinierten System leisten das die Funktionen `dsm_write` und `dsm_client_close`.

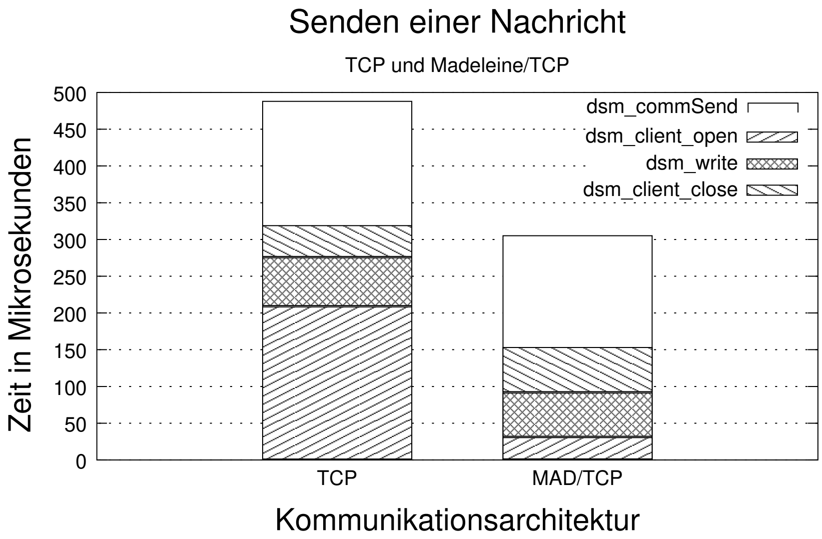

Abbildung 7.2: Aufschlüsselung der Ausführungszeiten der Funktion `dsm_commSend` für TCP und MADELEINE/TCP

In Abbildung 7.2 sind die durchschnittlichen Zeiten für das Senden einer Nachricht mit den Kommunikationsarchitekturen TCP und MADELEINE/TCP angegeben. MADELEINE/TCP benötigt deutlich weniger Zeit. Die Übertragung der Daten dauert zwar etwas länger, dafür wird aber kaum Zeit für den Verbindungsaufbau (`dsm_client_close`) benötigt.

In Abbildung 7.3 sind die durchschnittlichen Zeiten für das Senden einer Nachricht mit MADELEINE/BIP dargestellt. Da die Meßwerte für diese Kommunikationsarchitektur wesentlich höher sind, werden sie in einer eigenen Abbildung gezeigt. Der Anteil der Funktionen `dsm_client_open` und `dsm_write` an der Ausführungszeit der gesamten Funktion `dsm_commSend` ist so gering, daß auf ihre Darstellung verzichtet wurde. Die Ausführungszeit der Funktion `dsm_client_close` ist nahezu identisch mit der von `dsm_commSend`. Die Ausführungszeiten nehmen mit zunehmender Polling-Frequenz ab. Im besten Fall dauert das Senden einer Nachricht mit MADELEINE/BIP achtmal so lang wie im ursprünglichen System (TCP) und sogar etwa 13-mal so lang wie mit MADELEINE/TCP.

Die Analyse der Kommunikationsarchitektur MADELEINE/BIP wurde noch verfeinert. Es wurden zusätzliche Meßpunkte eingeführt (Schritte 1. - 3. der Funktion `mad_sendbuf_send` in Tabelle 7.2), um zwischen

[2] Mit der Übertragung ist an dieser Stelle die Übergabe der Daten an das Netzwerk bzw. die Kommunikationssoftware gemeint.

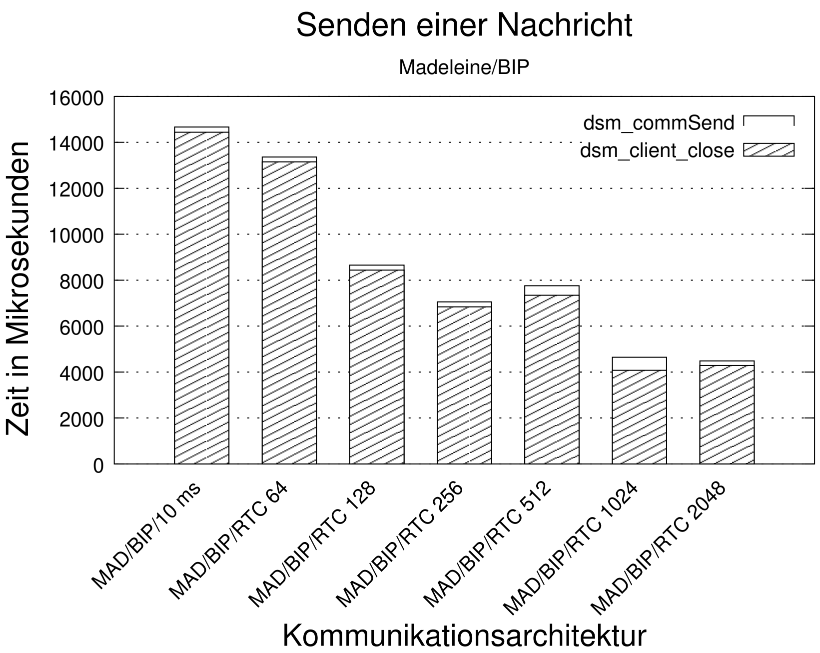

Abbildung 7.3: Aufschlüsselung der Ausführungszeiten der Funktion `dsm_commSend` für MADELEINE/BIP

kurzen und langen BIP-Nachrichten zu unterscheiden. In Abbildung 7.4 werden die durchschnittlichen Zeiten der Schritte für die unterschiedlichen Polling-Frequenzen angegeben.

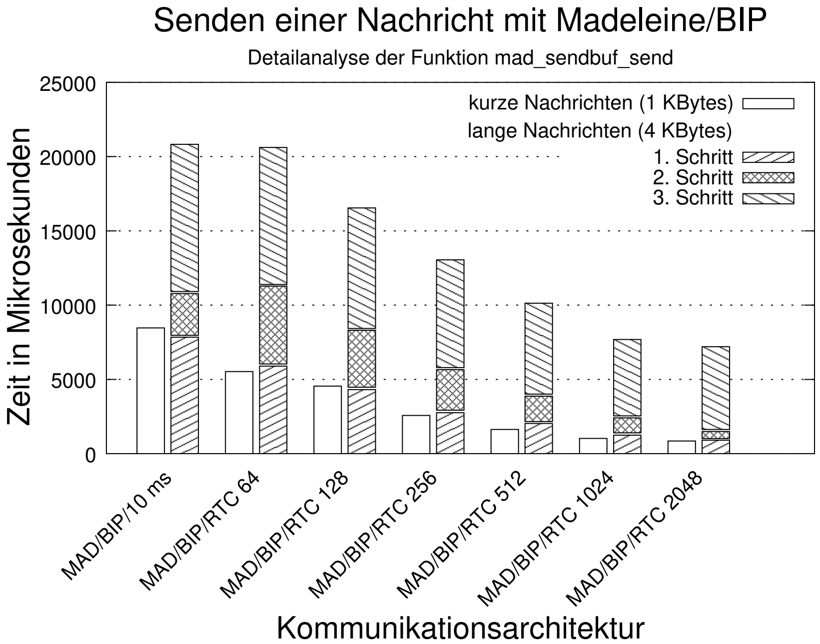

Abbildung 7.4: Ausführungszeiten der Funktion `mad_sendbuf_send` für kurze und lange BIP-Nachrichten

Beim Senden von kurzen Nachrichten ist eine deutliche Beschleunigung bei Erhöhung der Polling-Frequenz festzustellen. Dennoch dauert es im besten Fall (Frequenz 2048 Hz) etwa doppelt solange wie mit TCP bzw. MADELEINE/TCP. Auch das Senden von langen Nachrichten profitiert von der Erhöhung der Frequenz. Allerdings ist die Beschleunigung bei weitem nicht so hoch wie bei kurzen Nachrichten. Selbst im besten Fall dauert das Senden mehr als zehnmal so lang wie mit TCP bzw. MADELEINE/TCP.

Kapitel 8

Verwandte Arbeiten

In diesem Kapitel werden andere Arbeiten, die sich mit Systemen die virtuellen gemeinsamen Speicher simulieren im Allgemeinen und der Kommunikation in verteilten Systemen unter Berücksichtigung von Multi-Threading im Speziellen beschäftigen, vorgestellt und in Beziehung zur vorliegenden Arbeit gesetzt.

8.1 TreadMarks

TreadMarks [27] unterstützt paralleles Rechnen in einem Netzwerk von Computern durch die Simulation von gemeinsamem Speicher. Es wird durch eine *User-Level*-Bibliothek für das Betriebssystem Unix implementiert. Durch die Verwendung von Standard-Compilern für C, C++ oder Fortran ist das System portabel einsetzbar. Es unterstützt asynchrone Kommunikation, die auf UDP / Ethernet oder AAL3/4 / ATM basiert.

Eine TreadMarks-Applikation ist *single-threaded*, d.h. pro Prozeß existiert lediglich ein Thread. TreadMarks unterscheidet zwei Nachrichtenarten, Anfragen und Antworten. Eine Anfrage kann durch einen expliziten Aufruf einer Bibliotheksroutine oder durch eine Signalbehandlungsroutine für Seitenfehler gesendet werden.

Der Prozeß, der eine Anfrage gesendet hat, wird blockiert, bis eine Antwort eingetroffen ist. Nachdem die Behandlungsroutine abgearbeitet wurde, kann der unterbrochene Prozeß fortfahren.

Die Konzepte von TreadMarks und DSM-Threads sind ähnlich. DSM-Threads unterstützt zusätzlich Multi-Threading. Dies ermöglicht eine Überlappung von Kommunikation und Berechnungen, wenn man die Ausführung eines einzelnen Prozesses betrachtet. Nachdem ein Thread eine Anfrage gesendet hat, wird nicht der ganze Prozeß blockiert. Andere Threads können mit ihren Berechnungen fortfahren. In TreadMarks gibt es keine Überlappung von Kommunikation und Berechnungen, da ein Prozeß nach dem Senden einer Anfrage bis zum Eintreffen einer Antwort blockiert.

Ein Vorteil von TreadMarks ist die einfachere Implementierung auf Basis von Kommunikationsbibliotheken, die Notifikation durch Signale nicht unterstützen.

8.2 DSM-PM2

DSM-PM2 [28] ist eine Bibliothek, die virtuellen gemeinsamen Speicher in einer verteilten Umgebung simuliert. DSM-PM2 basiert auf PM2, einem Laufzeitsystem, welches Multi-Threading, effiziente Kommunikation im RPC-Stil und präemptive Thread-Migration unterstützt. Die Multi-Threading-Eigenschaften erbt PM2 von dem Thread-Paket Marcel. Die Kommunikation basiert auf Madeleine, einer Bibliothek für den Austausch von Nachrichten zwischen Prozessen. Madeleine unterstützt zahlreiche Kommunikationsarchitekturen, wie z.B. TCP/Ethernet, BIP/Myrinet oder MPI.

Wie in DSM-Threads stehen dem Benutzer mehrere Konsistenzprotokolle zur Verfügung. Darüberhinaus stellt DSM-PM2 eine Schnittstelle zur Verfügung, mit der der Benutzer eigene Konsistenzprotokolle implementieren kann. Konsistenzprotokolle können sowohl durch die Replizierung und Migration von Seiten als auch durch die Migration von Threads während der Laufzeit implementiert werden. DSM-Threads verwendet nur die Replizierung und Migration von Seiten.

In DSM-PM2 werden alle Nachrichten von einem besonderen Thread, dem communication thread, empfangen. Dieses Konzept wird in DSM-Threads durch den Kommunikationsserver realisiert.

8.3 Millipede

Millipede [29] ist ein DSM-System, das sich durch besondere Eigenschaften auszeichnet. Es unterstützt lediglich Sequentielle Konsistenz für gemeinsame Daten. Das Problem des *false sharing* wird durch die neue Technik MultiView behandelt. Variablen, die in einer physikalischen Speicherseite liegen, werden mit Hilfe des virtuellen Speichermanagements in unterschiedliche *views* eingeblendet. Für diese *views* kann man die üblichen Techniken zur Erkennung eines Zugriffsfehlers verwenden. Das System vermeidet so *false sharing*, da jeder Variablen ein eigener *view* zugeordnet werden kann.

Durch die Anwendung der MultiView-Technik ist das System sehr effizient. Es unterstützt fein-granulare Anwendungen. Da auf komplexe Implementierungen schwacher Konsistenzprotokolle (wie z.B. *Lazy Release Consistency*) verzichtet werden konnte, bleibt die Implementierung übersichtlich und klein im Vergleich zu anderen DSM-Systemen. Die Technik der MultiViews ermöglicht einen effizienten Einsatz von modernen Kommunikationsarchitekturen (wie z.B. FastMessages (FM) über Myrinet). Im besonderen können Möglichkeiten des *zero-copy overhead* beim Nachrichtenaustausch, d.h. die Daten werden direkt ohne Extrakopien an die Netzwerkkarte übergeben (Senden) bzw. von der Netzwerkkarte an ihre Zieladresse geschrieben (Empfangen), ausgeschöpft werden.

Millipede verwendet FastMessages für die Kommunikation. FastMessages ist eine Kommunikationssoftware, die speziell für moderne Netzwerke wie Myrinet entwickelt wurde. Typischerweise unterstützt solche Kommunikationssoftware nur Polling, um die Beendigung einer Sende- oder Empfangsoperation festzustellen. Mit Polling lassen sich gute Resultate erzielen, wenn bekannt ist, wann Sender und Empfänger Nachrichten austauschen. In DSM-Systemen ist dies nicht der Fall, d.h. Nachrichten werden zu nicht vorhersagbaren Zeitpunkten gesendet. In dieser Situation bremst häufiges Polling die Berechnungen, während seltenes Polling zu Verzögerungen beim Empfangen und Bearbeiten der Nachrichten führt.

Dieses Problem wird detailliert in [30] unter besonderer Berücksichtigung von Multi-Threading beschrieben. Sie schließen, daß Polling in Verbindung mit asynchroner Kommunikation und Multi-Threading inadäquat ist. Eine Lösung des Problems ist die Notifikation durch Signale für die Beendigung von Nachrichtenoperationen.

DSM-Threads behandelt das Problem des *false sharing* mit den üblichen Methoden, d.h. es stellt schwache Konsistenzprotokolle zur Verfügung. Beim Einsatz von modernen Kommunikationsarchitekturen (wie z.B. BIP/Myrinet) lassen sich ähnliche Probleme mit Polling beobachten. Die einzige konsequente Lösung ist, die Notifikation von Signalen in diese Kommunikationsarchitekturen zu integrieren. Dieser Ansatz wurde nicht realisiert, da er den Rahmen dieser Arbeit bei weitem gesprengt hätte.

8.4 Responsivität ohne Interrupts

Moderne Kommunikationsarchitekturen ermöglichen Prozessen die nahezu volle Leistungsfähigkeit neuer Netzwerke auszuschöpfen. Diese Aussage gilt nicht für alle Anwendungen. Anwendungen, die Standards wie PVM oder MPI einsetzen, können die Leistungsfähigkeit dieser modernen Kommunikationsarchitekturen ausschöpfen, da Prozesse in der Regel zu denselben Zeiten miteinander kommunizieren. In DSM-Systemen kann Kommunikation zu jedem beliebigen Zeitpunkt stattfinden, insbesondere ist dem Empfänger unbekannt, wann eine Nachricht eintrifft.

Moderne Kommunikationsarchitekturen bieten nicht die Möglichkeit der asynchronen Notifikation eines Prozesses, wenn eine Nachricht eintrifft. Stattdessen muß der Empfänger Polling einsetzen, um festzustellen ob eine Nachricht eingetroffen ist. Um ein zügiges Bearbeiten der Nachrichten zu gewährleisten, müßte das Polling mit einer hohen Frequenz durchgeführt werden. In Standards wie PVM und MPI wird Polling aber nur nach dem Senden einer Nachricht durchgeführt. Einerseits befreit das den Programmierer davon, selbst Pollingtests in das Programm einzufügen. Andererseits genügt dieser Ansatz nicht der Forderung nach einer hohen Frequenz.

In [19] werden zwei Ansätze, mit diesem Problem umzugehen, vorgestellt. Erstens wird vorgeschlagen, Latenzzeiten durch Einsatz von Multi-Threading zu kompensieren. Dieser Ansatz wird auch mit DSM-Threads verfolgt. Er funktioniert gut, wenn oft kommuniziert wird, so daß auf Ebene der Kommunikationssoftware Polling durchgeführt werden kann. Findet Kommunikation nicht häufig bzw. nicht mit hinreichender Frequenz statt, reicht dieser Ansatz nicht aus. Zweitens wird vorgeschlagen, Pollingtests nachträglich in den ausführbaren Code einzufügen. Es wird untersucht, wie eine optimale Frequenz an Tests erreicht werden kann. Dieses Problem ist nicht trivial. DSM-Threads verfolgt diesen Ansatz nicht, da die Manipulierung des Binärcodes die Portabilität des Systems verschlechtert.

Kapitel 9

Zusammenfassung

Diese Arbeit beschäftigte sich mit der Kommunikation in verteilten Systemen mit gemeinsamem Speicher, die auch DSM-Systeme[1] genannt werden. Es wurden Software-Realisierungen solcher Systeme, die sich durch Unterstützung von priorisiertem Multi-Threading und asynchrone Kommunikation auszeichnen, betrachtet. Die Ziele der Arbeit waren es, herauszufinden,

- wie diese Eigenschaften kombiniert werden können *und*
- wie von verschiedenen Kommunikationsarchitekturen abstrahiert werden kann.

In der Arbeit wurden allgemeine Anforderungen, die DSM-Systeme mit diesen Eigenschaften an eine Kommunikationsarchitektur stellen, um eine gute Responsivität und Multi-Threading-sichere Kommunikation zu realisieren, entwickelt. Diese Anforderungen lauten:

- Unterstützung indirekter Kommunikation,
- Unterstützung von Interrupts bzw. Notifikation durch Signale für die Erkennung abgeschlossener Nachrichtenoperationen *und*
- Trennung von verteilter Ausführung und Kommunikation.

Diese Anforderungen wurden mit Hilfe einer Fallstudie, die die Kombination von DSM-Threads und Madeleine als Ziel hatte, herausgearbeitet. DSM-Threads ist ein DSM-System, das die obigen Eigenschaften besitzt und verteilte Ausführung und die Realisierung des gemeinsamen Speichers in einem Netzwerk von Rechnern bereitstellt. Madeleine ist eine Kommunikationssoftware, die dem Anwender eine einheitliche Schnittstelle für die Benutzung verschiedener Kommunikationsarchitekturen bietet.

In Kapitel 1 wurden eine Einführung in die Thematik gegeben und allgemeine Ergebnisse dargelegt. An dieser Stelle wurden die Anforderungen, die ein DSM-System an eine Kommunikationsarchitektur stellt, detailliert beschrieben. Das zweite Kapitel stellte das zugrundegelegte DSM-System DSM-Threads vor. Neben den grundlegenden Konzepten und Eigenschaften wurden das Design und die Implementierung der Komponenten, die bei der Fallstudie berücksichtigt werden mußten, erläutert. Die Komponenten sind

- die Initialisierung,
- die Kommunikation,
- die Erzeugung entfernter Threads *und*
- die Terminierung.

Im dritten Kapitel wurde die Kommunikationssoftware Madeleine vorgestellt. Neben dem Design – schichtenartiger Aufbau mit Programmierschnittstelle und Portabilitätsschnittstelle – wurde die Implementierung erläutert. Die Beschreibung der Implementierung der Portabilitätsschnittstelle beschränkte sich auf die Module für die Unterstützung von TCP und BIP. Außerdem wurde die Anwendung der Kommunikationsbibliothek BIP erläutert. Am Schluß dieses Kapitels wurden zwei Verbesserungsmöglichkeiten für Madeleine vorgeschlagen. Diese sind die Trennung von verteilter Ausführung und Kommunikation und eine konsequente Unterstützung von Multi-Threading.

Die Methodik der Kombination wurde im vierten Kapitel erläutert. Die zentralen Komponenten bei einer Kombination im Kontext dieser Arbeit waren die Initialisierung und die Kommunikation. Werden bei diesem

[1] DSM steht für *Distributed Shared Memory* zu deutsch Verteilter gemeinsamer Speicher.

Schritt Teile des DSM-Systems verändert, so müssen weitere Komponenten berücksichtigt werden. Bei der durchgeführten Fallstudie mußten die Komponenten der Erzeugung entfernter Threads und der Terminierung zusätzlich berücksichtigt werden. In diesem Kapitel wurde ebenfalls das Design und die Implementierung eines Zwischenschrittes beschrieben. Dieser Zwischenschritt wurde durchgeführt, um die Komplexität der Kombination zu reduzieren.

Das kombinierte System wurde in den Kapiteln 5 (Design) und 6 (Implementierung) beschrieben. Die Komponenten Initialisierung, Kommunikation, Erzeugung entfernter Threads und Terminierung wurden detailliert betrachtet. Alternative Ansätze, wie z.B. verschiedene Modelle für die Erzeugung entfernter Threads und verschiedene Implementierungen des Pollings, wurden diskutiert.

In Kapitel 7 wurden Funktionalitätstests und Leistungsmessungen beschrieben. Letztere haben gezeigt, daß die Kombination von Polling, asynchroner Kommunikation und priorisiertem Multi-Threading keine zufriedenstellende Leistung ermöglicht. Die Beobachtungen bekräftigen die Forderung nach der Unterstützung von Interrupts bzw. der Notifikation durch Signale in Kommunikationsbibliotheken wie z.B. BIP.

Das achte Kapitel setzte die vorliegende Arbeit in Relation zu verwandten Arbeiten. Es wurden die Systeme TreadMarks, DSM-PM2 und Millipede vorgestellt und begründet, warum die beschriebenen Ansätze in dieser Arbeit nicht verfolgt wurden.

Anhang A

Testprogramme für DSM-Threads

A.1 Das Programm test_create

```
#include <pthread.h>
#include <dsm.h>

DSM_DATA_BEGIN
int z = 1;
float f = 1.5;

dsm_thread_t thread = {};
dsm_threadattr_t attr = {};
DSM_DATA_END

int y = 2;

void new_thread(void *arg) {
  int x = 3;
  extern char *dsm_selfhost();

  fprintf(stderr, "%s:%d:%x new thread (pid %d)\n", dsm_selfhost(), dsm_selfport(),
          pthread_self(), getpid());

  fprintf(stderr, "%s:%d:%x x %s\n", dsm_selfhost(), dsm_selfport(), pthread_self(),
          x == 3 ? "OK" : "ERROR");
  fflush(stderr);
  fprintf(stderr, "%s:%d:%x y %s\n", dsm_selfhost(), dsm_selfport(), pthread_self(),
          y == 2 ? "OK" : "ERROR");
  fflush(stderr);
  fprintf(stderr, "%s:%d:%x z %s\n", dsm_selfhost(), dsm_selfport(), pthread_self(),
          z == 4 ? "OK" : "ERROR");
  fflush(stderr);
  fprintf(stderr, "%s:%d:%x f %s\n", dsm_selfhost(), dsm_selfport(), pthread_self(),
          f == 1.5 ? "OK" : "ERROR");
  fflush(stderr);

  z = 4;
}

int main(int argc, char **argv) {
  int i, x = 3;

  pthread_init();
  dsm_init(argc, argv);

  if (argc < 2) {
    fprintf(stderr, "use: %s hostname1 [hostname2 ...]\n", argv[0]);
    fprintf(stderr, " or: %s #threads\n", argv[0]);
    exit(0);
  }
```

```
  z = 4;

  dsm_threadattr_init(&attr);

  if (isdigit(argv[1][0]))
    for (i = 0; i < atoi(argv[1]); i++)
      dsm_thread_create(&thread, &attr, (void *(*)(void*)) new_thread, NULL);
  else
    for (i = 1; i < argc; i++) {
      dsm_threadattr_setnode(&attr, argv[i]);
      dsm_thread_create(&thread, &attr, (void *(*)(void*)) new_thread, NULL);
    }

  fprintf(stderr, "%s:%d:%x main thread\n", dsm_selfhost(), dsm_selfport(), pthread_self());
  fprintf(stderr, "%s:%d:%x x %s\n", dsm_selfhost(), dsm_selfport(), pthread_self(),
          x == 3 ? "OK" : "ERROR");
  fflush(stderr);
  fprintf(stderr, "%s:%d:%x y %s\n", dsm_selfhost(), dsm_selfport(), pthread_self(),
          y == 2 ? "OK" : "ERROR");
  fflush(stderr);
  fprintf(stderr, "%s:%d:%x z %s\n", dsm_selfhost(), dsm_selfport(), pthread_self(),
          z == 4 ? "OK" : "ERROR");
  fflush(stderr);

  fprintf(stderr, "%s:%d:%x main quits %x\n", dsm_selfhost(), dsm_selfport(), pthread_self(),
          &main);

  dsm_exit(NULL);
}
```

A.2 Das Programm test_terminate

```
#include <pthread.h>
#include <dsm.h>

DSM_DATA_BEGIN
int counter = 0;
dsm_thread_t thread = {};
dsm_threadattr_t attr = {};
DSM_DATA_END

void new_thread(void *arg) {
  int num = (int)arg;
  extern char *dsm_selfhost();

  fprintf(stdout, "%s:%d:%x new thread (pid %d): num %d\n",
          dsm_selfhost(), dsm_selfport(), pthread_self(), getpid(), num);
  fflush(stdout);

  sleep(random()%3 + 1);
  if (--num) {
    dsm_threadattr_init(&attr);
    dsm_thread_create(&thread, &attr, (void *(*)(void*)) new_thread,
                      (void*)num);
  }

  counter++;
  if (!num) {
    fprintf(stdout,"counter %d\n", counter);
    fflush(stdout);
  }
}
```

```
int main(int argc, char **argv) {
  int num = (argc == 2) ? atoi(argv[1]) : 3;

  pthread_init();
  dsm_init(argc, argv);

  if (argc < 2) {
    fprintf(stderr, "use: %s #threads\n", argv[0]);
    exit(0);
  }

  new_thread((void *)num);

  dsm_exit(NULL);
}
```

Literaturverzeichnis

[1] Mark Baker and Rajkumar Buyya. Cluster computing: the commodity supercomputer. *Software Practice and Experience*, 29(6):551–576, May 1999.

[2] T. Agerwala, J. Martin, J. Mirza, D. Sadler, D. Dias, and M. Snir. SP2 system architecture. *IBM Systems Journal*, 34(2):152–184, 1995.

[3] Cray Inc. *Cray T3E*. http://www.cray.com/products/systems/crayt3e/.

[4] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[5] Compaq Computer Corporation. *ServerNet II*. http://www.servernet.com/.

[6] Gigabit Ethernet Alliance. *Gigabit Ethernet Technology Overview*. http://www.gigabit-ethernet.org/technology/index.html.

[7] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In David Abramson and Jean-Luc Gaudiot, editors, *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–267, Gold Coast, Australia, May 1992. ACM Press.

[8] T. E. Anderson, D. E. Culler, and D. A. Patterson. A case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[9] L. Prylli and B. Tourancheau. BIP: A new protocol designed for high performance networking on myrinet. *Lecture Notes in Computer Science*, 1388:472–??, 1998.

[10] MPI Forum. *Message Passing Interface (MPI) Forum Home Page*. http://www.mpi-forum.org/.

[11] G. A. Geist and V. S. Sunderam. The PVM system: Supercomputer level concurrent computation on a heterogeneous network of workstations. In Quentin F. Stout and Michael Joseph Wolfe, editors, *The Sixth Distributed Memory Computing Conference proceedings April 28–May 1, 1991, Portland, Oregon*, pages 258–261, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1991. IEEE Computer Society Press.

[12] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, Chicago, Illinois, April 18–21, 1994. IEEE Computer Society TCCA and ACM SIGARCH.

[13] IEEE. *IEEE 1003.1c-1995: Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language)*. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995.

[14] Luc Bougé, Jean-François Méhaut, and Raymond Namyst. Madeleine: An efficient and portable communication interface for RPC-based multithreaded environments. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, pages 240–247, Paris, France, October 12–18, 1998. IEEE Computer Society Press.

[15] T. Roeblitz and F. Mueller. Combining multi-threading with asynchronous communication: A case study with dsm-threads using myrinet via bip and madeleine. In *Proceedings of the 1st Myrinet User Group Conference*, pages 131–138, Lyon, France, September 2000. INRIA.

[16] Scott Pakin, Mario Lauria, and Andrew Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In Sidney Karin, editor, *Proceedings of the 1995 ACM/IEEE Supercomputing Conference, December 3–8, 1995, San Diego Convention Center, San Diego, CA, USA*, pages ??–??, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995. ACM Press and IEEE Computer Society Press.

[17] R.A.F. Bhoedjang, K. Verstoep, T. Ruhl, and H.E. Bal. Reducing data and control transfer overhead through network-interface support. In *Proceedings of the 1st Myrinet User Group Conference*, pages x1–x2, Lyon, France, September 2000. INRIA.

[18] Myricom Inc. *The GM API - Table of Contents.* http://www.myri.com/scs/GM/doc/gm_toc.html.

[19] Dejan Perkovic and Peter J. Keleher. Responsiveness without interrupts. In *Proceedings of the 1999 Conference on Supercomputing*, ACM SIGARCH, pages 101–108, N.Y., June 20–25 1999. ACM Press.

[20] F. Mueller. Distributed shared memory threads: DSM-threads. In *Proc. of the Workshop on Run-Time Systems for Parallel Programming*, pages 31–40, April 1997.

[21] F. Mueller. On the design and implementation of DSM-threads. In *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, pages 315–324, June 1997.

[22] Frank Mueller. A library implementation of POSIX threads under unix. In *Proceedings of the Winter 1993 USENIX Technical Conference and Exhibition*, pages 29–41, San Diego, CA, USA, January 1993.

[23] Michael Eberl, Hermann Hellwagner, Wolfgang Karl, Markus Leberecht, and Josef Weidendorfer. Fast Communication Libraries on an SCI Cluster. In *SCI Europe '98*, Bordeaux, France, September 1998.

[24] Hermann Hellwagner and Alexander Reinefeld. *SCI-Scalable coherent interface: architecture and software for high-performance compute clusters*, volume 1734 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1999.

[25] Antoine de Saint-Exupéry. *Der Kleine Prinz.* Karl Rauch Verlag, Düsseldorf, Frühjahr 2000.

[26] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *the Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.

[27] C. Amza, A. L. Cox, S. Dwarkadas, P. Keheler, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared Memory on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[28] Gabriel Antoniu, Luc Bougé, and Raymond Namyst. Dsm-pm2: a multi-protocol DSM layer for the PM2 multithreaded runtime system. In *Second Workshop on Parallel Computing for Irregular Applications WPCIA2*, Toulouse, France, January 2000.

[29] Ayal Itzkovitz and Assaf Schuster. Multiview and Millipage - Fine-Grain Sharing in Page-Based DSMs. In *the Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 215–228, New Orleans, February 1999.

[30] A. Itzkovitz, A. Schuster, and Y. Talmor. Harnessing the power of fast low-latency networks for software DSMs. In *Proc. of the 1st Workshop on Software Distributed Shared Memory (WSDSM'99)*, June 1999.

Eigenständigkeitserklärung

Hiermit erkläre ich, daß die vorliegende Arbeit

„Kommunikation in verteilten Systemen mit gemeinsamem Speicher“

von mir selbst und ohne jede unerlaubte Hilfe nur unter Verwendung der angegebenen Quellen angefertigt wurde.

Berlin, den 6. November 2000 Thomas Röblitz

Einverständniserklärung

Hiermit erkläre ich mein Einverständnis, daß diese Arbeit in der Bibliothek des Institutes für Informatik der Humboldt-Universität zu Berlin öffentlich ausgelegt wird.

Berlin, den 6. November 2000 Thomas Röblitz