Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur

Diplom

Netzwerktransparente IPC für L4/Fiasco

Sebastian Lehmann
Matr.-Nr. 2296852
`sepp@prak.org`

Betreuer: Dipl.-Inform. Lars Reuther
Verantwortlicher Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig

31. August 2002

diese Seite durch die Aufgabenstellung ersetzen!

Inhaltsverzeichnis

1 Einleitung

Moores Gesetz[17] verspricht eine Verdopplung der verfügbaren Rechenkapazität für etwa jedes Jahr. Dieses Gesetz gilt seit 1965 und Forscher sorgen weltweit dafür, dass es auch noch die nächsten Jahre gilt. Heutige Mikroprozessoren können mehrere Milliarden Operationen pro Sekunde ausführen. Für einige Anwendungsbereiche, wie die Wetter- oder Nuklearforschung oder für Hochleistungsserver, auf denen mehrere tausend Menschen gleichzeitig arbeiten, ist dies jedoch nicht genug. Man behilft sich dadurch, dass man mehrere Prozessoren gleichzeitig einsetzt. Es gibt verschiedene Ansätze, die Arbeit auf mehrere Rechenwerke aufzuteilen. Sie unterscheiden sich unter anderem darin, ob und wie die laufenden Programme an das zu Grunde liegende System angepasst werden müssen. Teilweise müssen die Programme speziell für den Einsatz auf einem Rechnerverbund geschrieben werden und spezielle Funktionen nutzen, wie beispielsweise MPI[7], um miteinander zu kommunizieren. Auf anderen Systemen wird versucht, den Programmen so gut wie möglich zu verbergen, dass sie auf verschiedenen Rechnern laufen, und ihnen wird vorgespielt, sie würden auf einem extrem schnellen System laufen.

Der zweite Ansatz wird von vielen Programmierern bevorzugt, da die Programme nicht oder nur minimal an die neue Umgebung angepasst werden müssen. Sie setzt jedoch die Unterstützung des Betriebssystems voraus. Ein eigenes Betriebssystem oder wenigstens ein Teil eines Clusterübergreifenden Systems muss auf jedem Rechner laufen. Ein grafisches Betriebssystem mit einer grossen Palette an unterstützter Hardware und mehreren Treiberschichten auf jedem Rechner ist hier nicht nötig. Dieser Einsatz ist für einen Mikrokern geeignet. Einzig für die Kommunikation der Prozesse, auch über die Rechnergrenze, muss gesorgt werden.

Mit dieser Arbeit soll ein erster Schritt getan werden, den Mikrokern Fiasco, welcher an der Technischen Universität Dresden entwickelt wurde, in einer Mehrcomputerumgebung einzusetzen. Die Programme sollen ohne Änderung, also völlig transparent, sowohl mit lokalen Prozessen als auch mit Prozessen auf anderen Rechnern kommunizieren können.

Fiasco bietet zwei grundlegende Möglichkeiten der Kommunikation zweier Threads. Sie können gemeinsam benutzten Speicher zum Datenaustausch verwenden oder mittels eines Systemaufrufes explizit Daten senden und empfangen. Bisher setzt diese Kommunikation voraus, dass beide Threads auf dem selben Rechner ausgeführt werden. Diese Arbeit soll den Systemaufruf so erweitern, dass auch Threads auf anderen Rechnern adressiert werden können.

1.1 Über dieses Dokument

Im zweiten Kapitel wird der Stand der Technik beschrieben. An Beispielen wird dargestellt, wie andere Systeme netzwerktransparente Kommunikation eingeführt haben. Die Semantik einer IPC-Operation unter L4/Fiasco, sowie die Änderung dieser Semantik durch vorhandene oder geplante Überwachungsmethoden wird untersucht. Ein Überblick über die für Fiasco verfügbaren Netzwerktreiber schliesst das Kapitel ab.

Im dritten Kapitel werden verschiedene Möglichkeiten entwickelt und verglichen, netzwerktransparente IPCs für Fiasco zu implementieren. Die Implementierung eines dieser Entwürfe wird im nächsten Kapitel beschrieben und im Abschnitt 5 getestet und bewertet. Ein Ausblick auf die Möglichkeiten der Weiterführung des Projektes und eine Zusammenfassung beenden die Arbeit.

1.2 Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

2 Stand der Technik

2.1 Begriffe

Diese Arbeit setzt voraus, dass sich der Leser mit den Grundbegriffen der Theorie von Betriebssystemen auskennt. Der Autor empfielt hierfür das Buch "Moderne Betriebssysteme"[20] von Andrew S. Tanenbaum. Der L4/Fiasco speziefische Teil erschliesst sich am besten aus dem L4-Referenzhandbuch[11]. Die für die Arbeit wichtigen Begriffe sind folgende:

L4/Fiasco Der L4 Mikrokern wurde von Jochen Liedtke bei der GMD entwickelt. In der Betriebssystemgruppe der TU-Dresden wurde ein zu L4 kompatibler Mikrokern, Fiasco, entwickelt und unter der freien Lizenz GPL veröffentlicht. Er ist die Grundlage für DROPS: The Dresden Real-Time Operating System Project.

Adressraum Ein Adressraum ist die Abbildung virtueller Speicherseiten auf den physisch vorhandenen Hauptspeicher mit bestimmten Zugriffsattributen. Eine physische Adresse kann in zwei verschiedenen Adressräumen unter verschiedenen virtuellen Adressen angesprochen werden. Speicherseiten können von einem Adressraum in einen anderen transferiert (Grant) oder von beiden gleichzeitig benutzt werden (Mapping).

Thread Ein Thread ist eine aktive Ausführungseinheit innerhalb eines Adressraumes. Ein Thread wird durch den verwendeten Adressraum, einem Registersatz und zusätzlichen Verwaltungsdaten im Kern charakterisiert.

IPC Eine Inter-Prozess-Communication (IPC) sendet eine Nachricht von einem Senderthread an einen Empfängerthread. Die genaue Semantik einer IPC in Fiasco wird im Abschnitt 2.3 diskutiert.

2.2 Netzwerktransparente IPC

In diesem Kapitel sollen einige existierende Systeme vorgestellt werden, in denen es möglich ist, Prozesse auf verschiedenen Rechnern miteinander kommunizieren zu lassen. In [13] werden die Ansätze in folgende drei Gruppen unterteilt:

- Systeme, die von vornherein dafür entwickelt wurden, in einer verteilten Umgebung zu laufen, wie z.B. Amoeba. Die grundlegende Kommunikationsarchitektur ist bereits auf Remote IPC ausgelegt.
- Systeme, die eine extra Schnittstelle zur Kommunikation mit Prozessen anderer Rechner bereitstellen, wie beispielsweise UNIX.
- Und Systeme, deren Kommunikationsaufrufe so erweitert wurden, dass sie auch eine netzwerktransparente IPC ermöglichen.

Aus jeder Gruppe soll im Folgenden ein Vertreter vorgestellt werden.

2.2.1 Amoeba

Amoeba [21] wurde seit 1980 unter der Leitung von Andrew S. Tanenbaum als verteiltes Betriebssystem entwickelt. Es vereinigt eine Menge von Maschinen und lässt sie als ein zusammengehöriges System arbeiten. Dem Nutzer bleiben die Anzahl und der Ort der einzelnen Prozessoren verborgen. Das System bietet den Threads zwei grundlegende Kommunikationsmöglichkeiten: den RPC und eine speziell für parallele Programme eingeführte "eins zu viele"-Kommunikation. Mit letzterer kann ein einzelner Sender eine Nachricht an mehrere Empfänger schicken. Serverprozesse registrieren beim Kern ihre Ports, auf denen sie auf Clientanfragen warten. Das System bietet eine Rechtesystem. Zum Aufruf einer Serverfunktion benötigt der Client eine "Capability". Das ist ein 128-Bit Objekt, welches den Serverport, die angesprochene Funktion, eine Rechtemaske und eine kryptographische Checksumme beinhaltet. Ist dem Kern der geforderte Serverport nicht bekannt, wird er mittels Broadcast im Netzwerk gesucht.

Bestandteil des Systems ist ausserdem eine spezielle Sprache, die Amoeba Interface Language (AIL), welche es dem Nutzer erleichtert, auf die Systemfunktionen zuzugreifen, indem sogenannte Stubs für den Zugriff auf entfernte Services generiert werden.

Sämtliche Kommunikation wurde von vornherein so geplant, dass sie entfernte Funktionsaufrufe erlaubt. Die lokale Kommunikation ist nur ein Spezialfall, bei dem Empfänger- und Senderrechner der selbe sind.

2.2.2 UNIX

Bei der Entwicklung von UNIX wurden mehrere Möglichkeiten der Kommunikation lokaler Prozesse ersonnen, z.B. shared Memory oder Pipes. Um mit Prozessen auf anderen Rechnern kommunizieren zu können, wurde eine komplett eigenständige Programmierschnittstelle eingeführt: die Sockets[20]. Damit können Verbindungen zu anderen Rechnern mit verschiedenen Parametern aufgebaut werden. So kann beispielsweise ein zuverlässiger, verbindungsorientierter Byte-Strom gewählt werden, der das entfernte Äquivalent zu einer Pipe darstellt. Alle Daten, die an einem Ende in den Socket geschrieben werden, kommen in der gleichen Reihenfolge auf der anderen Seite heraus.

Der Verbindungsaufbau erfolgt, indem der Server einen Socket erzeugt und ihn an einen Namensraum bindet. Bei der Verwendung von IPv4 (Internet Protokoll) ist dieser Namensraum ein 32-Bit-Integer zur Spezifikation des Rechners (IP Adresse) und ein weiterer 16-Bit-Integer für den speziellen Dienst auf dem Rechner (Port). Der Server führt anschliessend ein listen() aus, um dem Kern mitzuteilen, dass auf diesem Socket auf eingehende Verbindungen gewartet wird. Ein Client verbindet sich mit dem Aufruf von connect() zu dem wartenden Server, indem er die IP Adresse und Port des Servers angibt. Nachdem der Server die Verbindung mit accept() angenommen hat, ist die Verbindung aufgebaut und Daten können übertragen werden.

Ein Socket wird wie ein Dateideskriptor durch einen Integerwert repräsentiert. Die Funktionen read() und write() sind so geschrieben, dass sie sowohl mit Dateien und Pipes, als auch mit Sockets benutzt werden können. Dadurch wird ein gewisser Grad von Transparenz erreicht. Der Verbindungsaufbau ist bei Pipes und Sockets zwar verschieden, nach dem Verbindungsaufbau können beide jedoch gleich benutzt werden.

2.2.3 Mach

"Mach" ist ein von der Carnegie Mellon University School of Computer Science entwickelter Mikrokern. Er implementiert einen relativ komplexen IPC Mechanismus, dessen Hauptobjekt ein Port ist. Dieser ist die Repräsentation eines Kommunikationsendpunktes und beinhaltet eine Nachrichtenwarteschlange. Mach implementiert ein ausgeklügeltes Rechtekonzept. Jedem Port sind Sende- und Empfangsrechte zugeordnet, welche auch an andere Prozesse weitergegeben werden können. So sendet beispielsweise ein Client eine IPC an einen Server und sendet in der Anfrage ein einmaliges Senderecht für einen Port mit, auf dem der Client auf die Antwort des Servers wartet. Dieses Recht erlischt nach der einmaligen Benutzung durch den Server.

Der ursprüngliche Mach IPC-Pfad wurde nur für Einzelrechner entwickelt. Nachträglich wurde dann in mehreren Projekten versucht, die komplexe Semantik der IPC-Aufrufe auch in verteilten Umgebungen nachzubilden. Eine dieser Implementierungen ist das NORMA-IPC-Modul[4], welches den Machkern erweitert. NORMA bietet sogenannte Proxy-Ports, welche vom System wie normale Ports behandelt werden. Die Rechteverwaltung des lokalen Mach-Systems wird weiterhin benutzt. Nachrichten an diese Ports werden vom Kern zum NORMA-Modul umgeleitet, welches die Nachrichten dann über das Netzwerk weiterleitet. Das Netzwerkprotokoll implementiert ausserdem einen verteilten Referenzzahlabgleich, Empfangsrechteweiterleitung und Portdeaktivierungsmeldungen.

2.3 Semantik einer IPC in L4/Fiasco

In diesem Kapitel wird die Semantik einer Interprozesskommunikation unter L4/Fiasco dargestellt und Einschränkungen aufgezeigt, die auftreten, wenn diese Kommunikation durch bestimmte Verfahren beeinflusst wird. Der Grad der erreichbaren Transparenz zu direkter Kommunikation wird verglichen.

2.3.1 Direkte IPC

Eine lokale IPC wird duch den Systemruf l4_ipc() durchgeführt. Dabei wird eine Nachricht an einen anderen Thread geschickt. Diese Übertragung erfolgt synchron und ungepuffert. Das bedeutet, dass der Thread, der das Senden initiiert, so lange blockiert, bis der Empfänger einen entsprechenden Receive-Aufruf ausführt. Umgekehrt blockiert auch ein Receive, bis ein passender Send-Aufruf erfolgt. Bei diesem sogenannten Rendezvous werden die Daten vom Sender zum Empfänger übernommen.

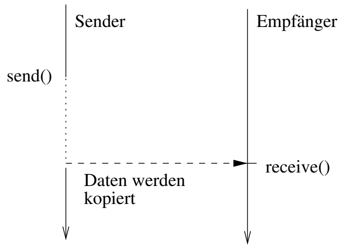

Abbildung 1: *Direkte IPCs erfolgen synchron, der Sender blockiert, bis der Empfänger eine entsprechende Receiveoperation ausführt.*

Diese Daten können 32 Bit Worte, Speicherbereiche oder Speicherseiten sein. Die Worte und Speicherbereiche, im Folgenden als Strings bezeichnet, werden vom Kern aus dem Sende- in den Empfängeradressraum kopiert. Gesendete Speicherseiten werden in den Empfängeradressraum eingeblendet. Als weiterer Parameter wird dem Systemaufruf noch ein Timeout-Parameter mitgegeben. Er gibt die maximale Dauer an, wie lange auf ein Zustandekommen der Kommunikation gewartet werden soll.

Man unterscheidet drei Arten einer IPC:

ShortIPC Es werden nur zwei Worte gesendet. Diese Daten werden in den Registern des Prozessors übertragen. Es muss also während der Übertragung kein Speicherzugriff erfolgen, wodurch diese Art der Datenübertragung die schnellste der hier vorgestellten ist.

StringIPC Dem Systemaufruf l4_ipc() kann ein Zeiger auf eine Datenstruktur übergeben werden, in der bis zu 2^{19} Wörtern und 2^5 Zeigern auf Speicherbereiche[1] gespeichert sind. Die Wörter werden vom Kern aus der Struktur des Senders in die Struktur des Empfängers und alle Strings in die vom Empfänger zur Verfügung gestellten Puffer kopiert. Die Anzahl der freien Worte und Speicherbereiche der Empfängerstruktur muss mindestens so gross sein wie die des Senders, sonst können nicht alle Daten übertragen werden.

FlexpageIPC Zusätzlich zu den Worten und Strings der StringIPC können in der übergebenen Datenstruktur Speicherseiten angegeben werden, die in den Adressraum des Empfängers eingeblendet werden.

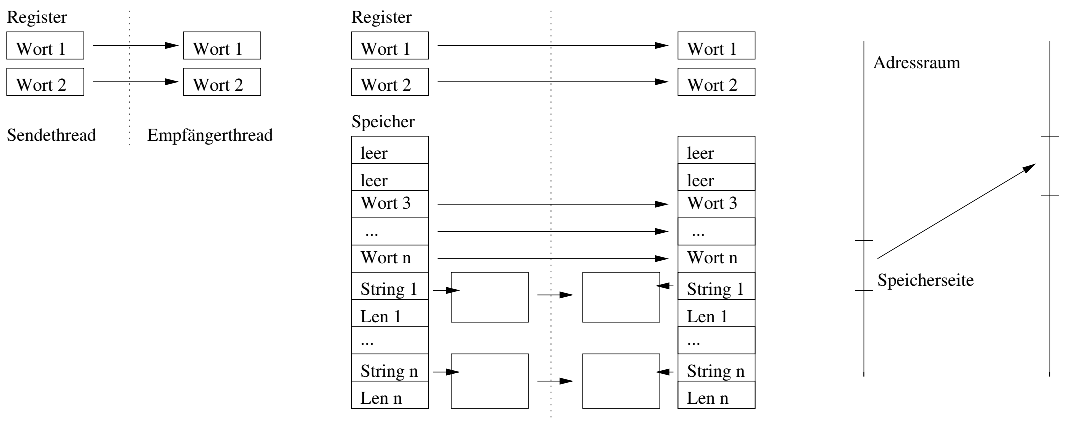

Abbildung 2: *Drei Typen von IPCs.* `Links:` *Bei der ShortIPC werden nur zwei Worte übertragen.* `Mitte:` *Mit StringIPC können mehrere Worte und Speicherbereiche kopiert werden.* `Rechts:` *Mit einer FlexpageIPC werden Speicherseiten in den Empfängeradressraum eingeblendet.*

Der Systemruf kehrt ohne Fehlercode zurück, wenn das Rendezvous stattgefunden hat und die Daten übernommen wurden. Ein Fehler kann in folgenden Situationen auftreten:

[1] In anderen Publikationen werden die Datenwörter auch "direct Strings" und die Speicherbereiche "indirect Strings" bezeichnet

TIMEOUT [2] Das Rendezvous kam nicht innerhalb der spezifizierten Zeit zustande.

NOT_EXISTENT Der angegebene Kommunikationspartner existiert nicht (mehr).

ABORTED,CANCELED Der Systemruf wurde vom System unterbrochen.

MSGCUT Die Daten konnten nicht vollständig übertragen werden, da z.B. die zu sendende Nachricht grösser war als der zur Verfügung stehende Empfangspuffer.

MAPFAILED Bei der Einblendung der gesendeten Speicherseiten in den Adressraum des Empfängers trat ein Fehler auf.

SNDPFTO,RCVPFTO Das Kopieren der Daten vom Sende- in den Empfängerpuffer konnte nicht innerhalb der spezifizierten Zeit erfolgen.

Zur Steigerung der Performance ist es möglich, eine Sende- und eine anschliessende Receiveoperation mit einem Systemcall abzuwickeln. Beispielsweise beantwortet ein Server die letzte Anfrage und wartet im Anschluss auf eine weitere Anfrage, oder ein Clientprozess stellt eine Anfrage an einen Server und wartet im Anschluss auf eine Antwort.

2.3.2 IPC über Clangrenzen

Die folgenden Angaben wurden aus [3] bezogen oder aufgrund dieser Daten geschlossen. Ein Verifizieren der Aussagen ist nicht möglich, da im verwendeten Fiasco-Kern das "Clans and Chiefs" Konzept nur teilweise implementiert ist.

Die folgende Erklärung wurde ebenfalls aus [3] übersetzt.

"Clans und Chiefs ist L4's grundlegender Mechanismus, um Sicherheitspolitiken [14] durchzusetzen. Sie erlauben das Überwachen des IPC-Informationsflusses.

Die Idee ist simpel: Der Erzeuger eines Tasks ist dieses Tasks Chief und alle (direkt) von einem Chief erzeugten Tasks bilden seinen Clan. Threads können IPCs nur direkt zu Threads im gleichen Clan oder zu ihrem Chief schicken. Wird eine Nachricht nach ausserhalb des Senders Clan geschickt, wird diese Nachricht statt dessen an den Chief des Senders geliefert (welcher die Nachricht weiterleitet oder nicht). Wird eine Nachricht an ein Mitglied eines Unterclans des Clans des Senders gesendet, wird diese Nachricht an den [Chief][3], dessen Clan den Empfänger enthält, geliefert. Dieses wird in Abbildung 3 verdeutlicht, in dem Kreise Tasks, Ovale Clans (mit deren Chiefs darüber) und Pfeile IPCs symbolisieren. Der dicke Pfeil zeigt die gewollte IPC, während die dünnen Pfeile den Weg zeigen, den die IPC tatsächlich nimmt (angenommen, alle Chiefs kooperieren)."

Hierbei sind folgende Besonderheiten gegenüber der direkten IPC festzustellen:

- Es findet kein Rendezvous zwischen Sender und Empfänger statt. Nach dem Rendezvous zwischen dem Sender und seinem Chief kann der Senderthread weiterlaufen, obwohl die Nachricht noch nicht beim Empfänger angekommen ist. Es ist nicht einmal sichergestellt, dass der Empfänger die Nachricht überhaupt erhält.

[2] Die Fehlercodes unterscheiden noch, ob der Fehler beim Senden (z.B. L4_IPC_SETIMEOUT) oder beim Empfangen (L4_IPC_RETIMEOUT) aufgetreten ist. Für die hier vorgenommenen Betrachtungen spielt das jedoch keine Rolle, so dass jedes Fehlertypenpaar (zu TIMEOUT) zusammengefasst wurde.

[3] Das Original spricht hier fehlerhaft vom Clan als Empfänger.

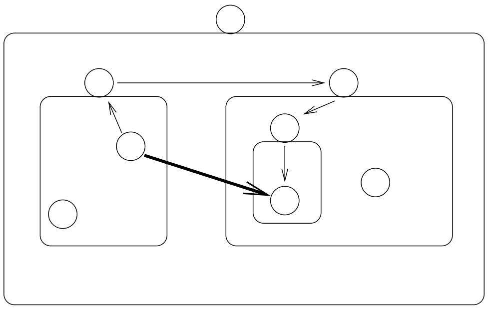

Abbildung 3: *Nachrichtenumleitung durch Clans und Chiefs nach [3].*

- Die Fehlercodes des IPC-Aufrufes müssen deshalb neu interpretiert werden:

  OK [4] Der Chief hat die Nachricht entgegengenommen. Die Nachricht ist noch nicht beim eigentlichen Empfänger angekommen. Bei der weiteren Übertragung können noch Fehler auftreten, über die der Sender nicht unterrichtet wird.

  TIMEOUT Der Chief-Prozess konnte eine Sendeoperation nicht in der spezifizierten Zeit entgegennehmen. Der eigentliche Empfänger kann durchaus in der Zeit einen Receiveaufruf durchgeführt haben, die Kommunikation fand trotzdem nicht statt.

  MSGCUT Der Chief-Prozess stellte nicht genug Puffer zur Verfügung um die Nachricht zwischenzuspeichern. Die zur Verfügung gestellten Puffer des Empfänger werden nicht beachtet.

  NOT_EXISTENT Eine IPC, welche über Clangrenzen geschickt wird, wird dem Chief zugestellt. In der L4-Hacker Mailingliste [1] wurde darüber diskutiert, ob eine IPC an einen nicht vorhandenen Thread auch eine Kommunikation über Clangrenzen ist. Sollte dies zutreffen, wird die Nachricht dem Chief zugestellt, und der Fehler NOT_EXISTENT kann überhaupt nicht auftreten. Im anderen Fall ist die Bedeutung des Fehlercodes der gleiche wie im Fall der direkten IPC: der Empfängerthread existiert nicht. Beide Ansätze sind berechtigt, und man muss bei einer konkreten Implementierung spezifizieren, welche Semantik genutzt werden soll.

Wichtig ist festzuhalten, dass auch im Fall, wenn die IPC-Operation ohne Fehlercode zurückkehrt, noch nicht die Semantik einer direkten IPC erreicht ist. Es hat nur ein Rendezvous zwischen Sender und Chief stattgefunden, nicht zwischen Sender und Empfänger. Es ist nicht einmal sicher, dass die Nachricht den Empfänger überhaupt erreicht. Zur Synchronisation ist eine solche IPC ungeeignet.

Ein typischer Anwendungsfall für eine IPC-Operation ist der sogenannte Call-Aufruf. Hierbei wird eine Nachricht an einen Server geschickt und auf die Antwort gewartet. In Fiasco kann ein solcher Aufruf mit einem einzigen Systemaufruf durchgeführt werden. Ein Timeout gibt an, nach welcher Zeit die Aktion abgebrochen wird. Man sollte diesen auf einen relativ grossen, jedoch nicht unendlich grossen, Wert setzen. Ist der Timeout auf unendlich gesetzt, wird der Prozess nicht weiterlaufen, wenn der Server Probleme mit der Anfrage hat. Beim Setzen eines Timeouts

[4] Der Rückkehrcode 0 hat die Bedeutung, dass kein Fehler auftrat. Es ist keine Konstante dafür definiert, zur besseren Lesbarkeit wird im Folgenden die Abkürzung OK verwendet.

kann bei einem Serverfehler eine clientseitige Fehlerbehandlung erfolgen, z.B. eine Fehlerausschrift oder ein erneuter Anfrageversuch. Es ist also sinnvoll, Callaufrufe mit einem endlichen Timeout zu versehen. Die Semantik eines solchen Callaufrufes mit endlichem Timeout bei einem direkten IPC und eines IPC-Aufrufes über Clangrenzen ist wieder gleich, vorausgesetzt, der Chief kooperiert[5]. Kehrt der Aufruf ohne Fehler zurück, ist in beiden Fällen die Anfrage beim Server angekommen, dieser hat sie beantwortet und das Ergebnis ist auch ordnungsgemäss zugestellt worden. Wenn ein Fehler aufgetreten ist, wird ein Fehlercode zurückgeliefert. Die Interpretation des Fehlercodes ist folgender:

ENOT_EXISTENT, SE* [6] Der Sendeaufruf zum Server ist gescheitert. Die gewünschte Aktion wurde beim Server nicht ausgeführt.

RETIMEOUT Die Daten wurden zum Server gesendet, in der spezifizierten Zeit wurde jedoch kein Ergebnis zurückgeliefert. Es ist unbekannt, ob der Server die gewünschte Aktion ausgeführt hat. Beispielsweise könnte der Server die Nachricht zwar empfangen, sie dann jedoch wegen kurzzeitiger Überlastung oder Speichermangels verworfen haben. Im Falle einer Kommunikation über Clangrenzen tritt dieser Fehlercode auch auf, wenn der Chief die Nachricht des Senders zwar angenommen hat, bei der Weiterleitung der Anfrage jedoch Probleme auftraten.

RE* Der Server hat die Aktion ausgeführt und versuchte, die Rückgabedaten zurückzusenden. Bei dieser Übertragung trat ein Fehler auf.

OK Die gewünschte Aktion wurde vom Server ausgeführt und die Rückgabewerte erfolgreich empfangen.

2.3.3 IPC mit transparenten Monitoren

Mit [10] führte IBM das Konzept der synchronen IPC über transparente Monitore ein. Das ermöglicht das Überwachen der Kommunikation von Prozessen ohne die Änderung der Semantik wie sie bei "Clans and Chiefs" auftrat. Transparente Monitore sind Prozesse, die sich in den IPC Pfad eines zu überwachenden Prozesses einklinken. Wird eine IPC durch einen transparenten Monitor gesendet, empfängt dieser die Nachricht. Im Gegensatz zu Chiefs bleibt der Sender jedoch weiterhin blockiert. Erst wenn die Nachricht zum endgültigen Empfänger gelangt, kann der Sendethread weiterlaufen. Es findet also wie bei der direkten IPC ein Rendezvous statt. Die Semantik ähnelt der direkten IPC mehr, als das beim Einsatz von "Clans und Chiefs" der Fall war.

Um diese Funktionsweise zu ermöglichen, sind Erweiterungen des Kerns nötig:

- Der Monitor nimmt mit einem speziellen Systemaufruf die Nachricht des Senders entgegen. Der Kern belässt den Sender weiterhin blockiert.
- Der Monitor kann mit einem weiteren Systemaufruf die Nachricht an den Empfänger weiterleiten. Dabei wird der ursprüngliche Sender im "true source" Feld zusätzlich mitangegeben. Die Nachricht wird nun zugestellt, als ob sie von dem ursprünglichen Empfänger stammen würde. Dessen Blockierung wird aufgehoben und eine erfolgte Zustellung zurückgemeldet.

[5] Offensichtlich ist die Semantik eines Aufrufes geändert, wenn der Chief übertragene Daten ändert.

[6] Mit SE* sind alle Fehlercodes gemeint, die beim Senden einer IPC auftreten, beispielsweise L4_IPC_SETIMEOUT.

- Der Monitor hat ebenfalls die Möglichkeit, den Absender zu ändern. Dazu wird im Feld "true source" ein anderer Thread angegeben und ausserdem der ursprüngliche Absender im Feld "blocked source". Der Kern stellt die Nachricht dann mit dem neuen Absender zu, hebt die Blockierung des ursprünglichen Absenders auf und meldet ihm die erfolgreiche Zustellung.
- Des weiteren kann der Monitor auch noch den Empfänger der Nachricht modifizieren. Dazu schickt er die Nachricht einfach an einen anderen Thread und übergibt wie in den vorigen Fällen "true source" oder "blocked source", um dem Kern mitzuteilen, dass diese IPC eine Weiterführung einer abgefangenen Nachricht ist.
- Ausserdem kann der Monitor auch eine Fehler-IPC an den Sender schicken, um ihm so einen Fehlercode mitzuteilen. Der Kern hebt dann die Blockierung des Senders auf und meldet den angegebenen Fehlercode zurück.

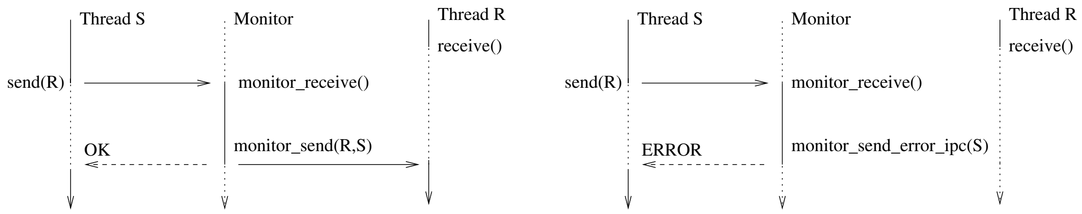

Abbildung 4: ***Links:*** *Der Sender bleibt solange blockiert, bis die Nachricht vom Monitor an den Empfänger zugestellt wurde.* ***Rechts:*** *Der Monitor kann auch eine Fehler-IPC an den Sender schicken und so einen Übertragungsfehler anzeigen.*

Die beschriebenen Funktionen erlauben es, die Semantik des direkten IPCs sehr ähnlich nachzubauen. Die verbleibenden Unterschiede sind:

OK Es hat ein Rendezvous zwischen dem Sender und dem Empfänger stattgefunden und die Daten wurden übertragen. Die Bedeutung entspricht der im Fall der direkten IPC, wenn der Monitor nicht den Absender oder Empfänger geändert hat.

TIMEOUT Es kann ein Timeout auftreten, obwohl der Empfänger empfangsbereit ist, wenn ein Monitor zu lange zur Bearbeitung der Nachricht benötigte.

MSGCUT Wenn ein Monitor einen Fehler beim Empfangen oder Senden der Nachricht verursacht, erscheint es dem Sender wie ein Fehler des Empfängers.

2.3.4 Vergleich

Der IPC-Mechanismus von L4/Fiasco erlaubt eine synchrone und sichere Übertragung von Daten zwischen einem Sende- und einem Empfangsthread. Es wurden verschiedene Möglichkeiten geschaffen, diese Übertragung zu überwachen, zu verändern oder umzuleiten.

Das Konzept der "Clans und Chiefs" ändert dabei jedoch die Semantik sehr stark, die Kommunikation ist nun nicht mehr synchron.

Diese Semantikänderung tritt nicht bei der Verwendung von transparenten Monitoren auf. Hier ist die Synchronität sichergestellt, es kann auch eine Fehlerbehandlung im Monitor erfolgen.

2.4 Netzwerktreiber für Fiasco

2.4.1 Übersicht

Zur Implementierung von netzwerktransparenten IPCs muss auf dem L4/Fiasco-System die Möglichkeit bestehen, mit anderen Rechner über ein Netzwerk zu kommunizieren. Die Mindestanforderung an einen Treiber ist das Erkennen und Initialisieren einer Netzwerkkarte und das Bereitstellen einer Schnittstelle zum Senden und Empfangen von Paketen. Wünschenswert wäre ein darauf aufsetzender TCP/IP-Stack, da dieser bereits Verbindungsmanagement mit Flusskontrolle implementiert. Echtzeitfähigkeit ist für diese Anwendung nicht notwendig.

Der Autor nutzt zum Entwickeln und Testen der Software das Produkt "VMware Workstation 3.1"[9]. Es simuliert einen kompletten Rechner, inklusive einer mit dem Netzwerk des Hostrechners verbundenen Netzwerkkarte. Die simulierte Netzwerkkarte ist vom Typ "AMD PCnet-PCI II compatible Ethernet adapter". Diese wird vom Linuxtreiber unterstützt und als "AMD Lance"[7] Karte angesprochen. Der Netzwerktreiber muss diese Karte unterstützen.

2.4.2 L4Linux

Der Treiberstack innerhalb eines Linuxkerns erfüllt alle Anforderungen. Mit dem L4Linux-Projekt [16] ist es möglich, einen Linuxkern auf L4/Fiasco zu starten. Jede von Linux unterstützte Netzwerkkarte würde auch von dieser Lösung unterstützt werden. Allerdings wäre der Aufwand sehr hoch. Die IPC müsste unter L4/Fiasco abgefangen werden, an den Routertask unter Linux weitergeleitet werden, der die Nachricht dann weiter zustellt. Nachteilig an dieser Lösung ist, dass man sie nicht ohne L4Linux betreiben kann, und es wird sehr schwer, dem L4Linux zu ermöglichen, die netzwerktransparente IPC zu nutzen, ohne in Dead-Locks zu laufen.

2.4.3 Spezialhardware

Für Spezialhardware, wie beispielsweise die Myrinet- [18] oder ATM-Netzwerkkarten [5], wurden Treiber entwickelt. Diese sind allerdings nur auf dieser Hardware funktionsfähig und erfüllen spezielle Anforderungen, wie Echtzeitfähigkeit, die bei der Weiterleitung von IPCs nicht benötigt werden. Der Myrinettreiber von Sven Reigl bietet des weiteren keinen TCP/IP-Socket, sondern nur eine einfache UDP-Unterstützung [19]. Da die von VMware simulierte Netzwerkkarte nicht unterstützt wird, ist dieser Treiber für die weitere Arbeit ungeeignet.

2.4.4 Oshkosh

Das Netzwerktreiberprojekt Oshkosh von Jork Löser portiert Netzwerktreiber aus dem 2.4. Linux-Kern auf Fiasco, bietet damit also auch einen Treiber für die "AMD Lance Netzwerkkarte". Die Client-Schnittstelle basiert auf dem "DROPS Streaming Interface" (DSI)[12]. Es gibt diverse Client-Programme, sowie einen Stub für L4Linux. Das erlaubt insbesondere, das L4Linux parallel zu anderen Applikationen auf die Netzwerkkarte(n) zugreifen kann. Ein TCP/IP-Stack wird im Rahmen des Projektes μsina[2] von Christian Helmuth, Andreas Westfeld und Immanuel Scholz entwickelt. Zur Zeit ist dieser noch nicht einsatzfähig, die Fertigstellung sollte jedoch mit der Abgabe dieser

[7] In der weiteren Arbeit wird für diese Netzwerkkarte der Name "AMD Lance" benutzt.

Arbeit zusammenfallen. Der Stack soll eine 4.4BSD-kompatible Schnittstelle erhalten. Dies sind die bereits im Fallbeispiel "Unix" beschriebenen Funktionen um einen Socket.

2.4.5 Ether-Paket

Der von Michael Hohmut aus dem OSKit[6] herausgelöste Netzwerktreiber[8] des Paketes "Ether" nutzt Netzwerkkartentreiber aus dem Linuxkern und bietet einen vollwertigen TCP/IP-Stack. Der Treiber für die "AMD Lance Netzwerkkarte" kommt in der Version 0.79 jedoch mit einem Bug, der die Hardware mehrfach erkennt. Ein Patch aus der OSKit-Mailingliste behebt das Problem. Das Paket Ether wird von Michael Hohmut nicht mehr gewartet, sollte also für zukünftige Arbeiten nicht mehr benutzt werden. Ether stellt jedoch einen vollwertigen TCP/IP-Stack auf der gewünschten Hardware zur Verfügung.

2.4.6 Bewertung

Das Paket Oshkosh ist der zukunftssicherste Netzwerktreiber für Fiasco, da er aktiv weiterentwickelt wird und Treiber für aktuelle und zukünftige Hardware bietet. Der TCP/IP-Stack ist jedoch noch in Arbeit. Andererseits ist der Ethertreiber der einzige einsetzbare Treiber mit TCP/IP-Stack, er ist allerdings veraltet. Als Kompromiss wird die Arbeit mit dem Ethertreiber fortgeführt, aber zusätzlich eine Zwischenschicht eingezogen, die zu der künftigen TCP/IP-Schnittstelle von Oshkosh kompatibel ist. Dadurch sollte die Portierung dieser Arbeit vom Ethertreiber zum Oshkoshtreiber einfach sein.

3 Implementierungsansätze

Dieses Kapitel beschreibt Möglichkeiten, netzwerktransparente IPC unter Fiasco zu implementieren und vergleicht deren Machbarkeit, Aufwand und Nutzen[8].

Hierbei wird speziell darauf eingegangen, wie die Identifikatoren eines Threads erweitert werden müssen. Einerseits sollen nun auch Threads auf einem anderen System referenziert werden, andererseits muss eine solche ID nun systemweit eindeutig sein.

Danach werden mehrere Möglichkeiten entwickelt, um Nachrichten, die an einen Thread auf einem anderen Rechner adressiert sind, abzufangen und weiterzuleiten. Einerseits erfordert es Änderungen des Kerns, um die IPCs abzufangen, und andererseits ein spezielles Programm, welches die Nachrichten über das Netzwerk verteilt.

Ausserdem wird auch darauf eingegangen, an welchen Stellen die weitere Arbeit ansetzen könnte, um das System weiter auszubauen. So kann in Zukunft die Netzwerktransparenz des IPC-Aufrufes auch auf Flexpages erweitert oder die Migration von Prozessen integriert werden.

3.1 Definition der Netzwerktransparenz

Netzwerktransparente Kommunikation bedeutet, dass ein Thread mit dem gleichen Systemaufruf, mit dem er Daten an einen anderen lokalen Thread sendet, nun auch Daten an einen Thread schicken kann, der sich auf einem anderen Rechner befindet. Dem Fiasco-Systemaufruf sys_ipc() sollen also die gleichen Parameter übergeben werden, einzig die übergebene ThreadID des Empfängers spezifiziert einen Prozess auf einem anderen Rechner.

Auf dem lokalen Rechner muss also eine Unterscheidung getroffen werden zwischen lokalen Threads, welche vom Kern direkt zugestellt werden können und entfernten Threads, die gesondert behandelt werden müssen. Zusätzlich muss es auch möglich sein, ThreadIDs an andere Threads zu senden. Wenn beispielsweise ein Thread 5 eine Nachricht an Thread 7 auf einem anderen Rechner schickt, könnte diese Nachricht die Adresse von einem Thread 3 enthalten. Beide Threads 5 und 7 müssen den gleichen Thread 3 erreichen, wenn sie eine Nachricht an die übertragene ThreadID senden.

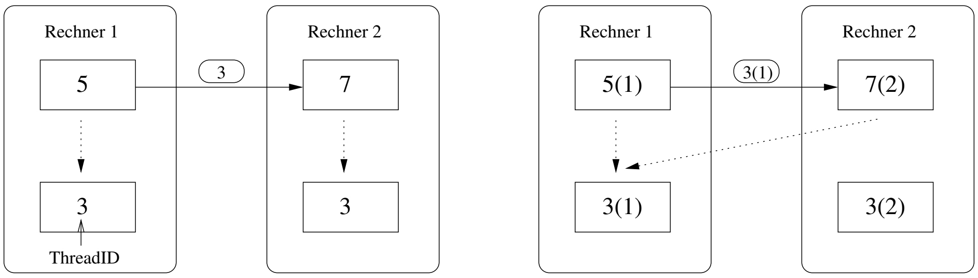

Abbildung 5: *ThreadIDs müssen im Rechnerverbund eindeutig sein.* `Links` *verweist die übertragene ThreadID 3 auf verschiedene Threads.* `Rechts` *ist die HostID Teil der ThreadID und damit systemweit eindeutig.*

Die Identifikatoren für Threads müssen also im gesamten Rechnerverbund eindeutig sein. Diese

[8]Geschwindigkeit, Einhaltung der Semantik etc.

eindeutigen IDs sind in der Spezifikation [11] bereits vorgesehen. Eine ThreadID setzt sich neben der Task- und Threadnummer auch aus einem Feld für eine SiteID zusammen. Dieses Feld ist in der bisher verwendeten Version von Fiasco nicht verwendet und ist auf 0 gesetzt. Nun wird jedem Host im Netzwerk eine eindeutige Nummer zugeordnet, die alle Prozesse auf diesem Rechner als SiteID erhalten. Damit ist sichergestellt, dass alle Prozesse im Netzwerk einen eindeutigen Identifikator haben. In einigen Fällen ist es sinnvoll, einen Prozess anzusprechen, der sich auf dem lokalen Rechner befindet. Beispielsweise ist es nicht sinnvoll, den Speichermanager eines anderen Rechners zu kontaktieren, um Speicher gemapt zu bekommen. Dafür wurde die SiteID 0 freigehalten. Eine ThreadID mit der SiteID 0 referenziert immer einen lokalen Prozess. Wenn diese ID an einen Thread auf einem anderen Rechner geschickt wird, referenziert sie nun auf einen anderen Thread. Dieses Verhalten ist sinnvoll einsetzbar, wenn beispielsweise auf allen Rechnern des Rechnerverbundes unter der gleichen Task- und Threadnummer ein Fileserver läuft. Hier kann jeder Thread seinen lokalen Fileserver verwenden.

Wenn es in Zukunft möglich sein wird, Prozesse zu migrieren, wird sich die Bedeutung der SiteID ändern. Es ist nun möglich, dass ein Prozess mit einer HostID auf einen Rechner mit einer anderen ID migriert. Diese ID gibt also nicht mehr an, auf welchem Rechner der Thread ausgeführt wird. Um wieder eine systemweit eindeutige ThreadID zu generieren, kann das Feld SiteID mit der Rechnernummer gefüllt werden, auf dem der Thread gestartet wurde.

3.2 Routerprozess

Im Folgenden werden Ansätze für das Design der zentralen Komponente, des Routerprozesses, diskutiert. Das ist die Komponente, an die der Kern die IPCs weiterleiten muss, die an nichtlokale Threads geschickt werden sollen. Der Prozess wird die Nachricht entgegennehmen, einpacken und an den Rechner im Netzwerk schicken, auf dem der Zielthread ausgeführt wird. Umgekehrt nimmt der Routerprozess auch Nachrichten aus dem Netzwerk entgegen, decodiert sie und stellt sie dann dem eigentlichen Empfänger zu.

Die Aufgabenstellung schliesst die Weiterleitung von Nachrichten, welche Speicherseiten enthalten, aus. Es sollen nur Nachrichten behandelt werden, die Daten von einem Thread zu einem anderen kopieren. Sollen auch Flexpage IPCs netzwerktransparent verarbeitet werden, ist eine verteilte Speicherverwaltung in den Routerprozess zu integrieren. Dadurch können dann auch Speicherseiten in den Adressraum eines Treads auf einem anderen Rechner eingeblendet werden.

3.2.1 Dummy-Threads

Eine einfache Herangehensweise ist die Einführung eines Dummy-Threads für jeden Thread auf einem anderen Rechner, der IPC von diesem Rechner empfangen kann. Dieser Thread wartet nur auf eingehende Nachrichten und sendet diese an einen Routerserver weiter, der sie dann über das Netzwerk weiterleitet. Auf dem anderen Rechner stellt der dortige Routerserver die IPC dem Dummy-Thread des Senders zu. Dieser sendet die IPC dann weiter an den eigentlichen Empfänger. Der Dummy-Thread auf jedem Rechner hat die gleiche ThreadID wie der Original-Thread. Es wird die HostID des Originalthreads zur Generierung der ThreadID verwendet. Dadurch ist sichergestellt, dass übertragene ThreadIDs auf allen Rechnern Gültigkeit haben. Es bedeutet jedoch auch, dass Threads, deren HostID nicht der des Rechners entspricht, ausgeführt werden müssen. Ein weiteres Problem ist die Threaderzeugung. Wenn auf einem Rechner ein Thread erzeugt wird, muss auch auf allen anderen Rechner der entsprechende Dummy-Thread generiert werden. Das bedeutet, dass es möglich sein muss, den Routertask über jede Threaderzeugung zu informieren.

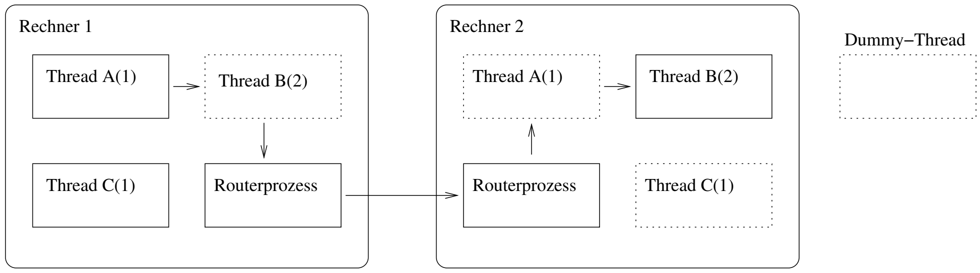

Abbildung 6: *Für jeden nichtlokalen Thread gibt es einen Dummythread. Dieser nimmt die Nachricht entgegen und leitet sie an den Routerprozess weiter.*

3.2.2 Clans & Chiefs

Bereits in [14] beschrieb Jochen Liedtke, wie das Konzept von "Clans und Chiefs" auf das Netzwerk ausgeweitet werden kann. Dazu werden alle Prozesse, die über das Netzwerk kommunizieren sollen, in einem Clan zusammengefasst. Innerhalb dieses Clans können die Threads mit direkten IPCs kommunizieren. Werden Nachrichten an Threads ausserhalb des Clans gesendet, werden diese an den Chief gesendet und dieser kümmert sich um die Weiterleitung zum Empfänger. Wenn man definiert, dass Threads, die auf diesem Rechner nicht existieren, auch Threads ausserhalb des Clans sind und deshalb die Kommunikation an den Chief umgeleitet wird, kann dieser die Nachrichten über das Netzwerk weiterleiten. So lassen sich auf einfachste Weise netzwerktransparente IPCs mit der Semantik einer Kommunikation über Clangrenzen implementieren.

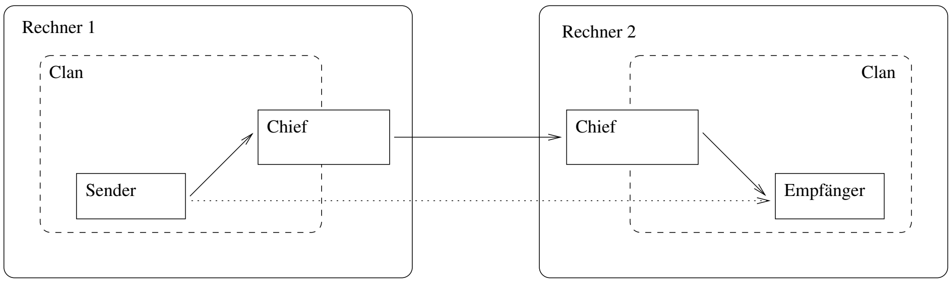

Abbildung 7: *Netzwerktransparente IPC lässt sich durch je einem Clan pro Rechner erreichen. Nachrichten nach ausserhalb des Rechners werden an den Chief gesendet.*

Leider ist das Konzept der "Clans and Chiefs" im aktuellen Fiascokern nicht implementiert. Da die Implementierung von "Clans und Chiefs" nicht Aufgabe dieser Arbeit ist, wird im nächsten Kapitel analysiert, welche Teile dieses Konzeptes notwendig sind, um netzwerktransparente IPCs zu implementieren.

3.2.3 Redirect und Deceit

Der IPC-Pfad einer Kommunikation über Clangrenzen beruht darauf, dass die Nachrichten, die die Grenze passieren sollen, an einen ausgezeichneten Thread zugestellt werden: den Chief. Adaptiert man dieses auf eine Kommunikation über Rechnergrenzen, müssen also alle Nachrichten, welche an einen Thread auf einem anderen Rechner geschickt werden, an einen speziellen Prozess "umgeleitet" werden.

Die Aufgabe des Chiefs übernimmt ein Routertask, der die Nachricht entgegennimmt, einpackt und über das Netzwerk an den Zielrechner versendet. Dort entpackt ein anderer Routertask die Nachricht wieder und stellt sie dem ursprünglichen Empfänger zu. Die nötigen Unterstützungen des Kerns zur Durchführung dieses Ablaufes sind die folgenden zwei:

Redirecting Die Nachricht, welche an einen Prozess auf einem anderen Rechner adressiert ist, wird vom Kern an den Routertask umgeleitet. Dieser bekommt in seiner Receive-Operation nicht nur alle übertragenen Daten und die ID des Senders übergeben, sondern zusätzlich auch die ID des eigentlichen Empfängers. Er kann sich nun darum kümmern, dass die Nachricht den Empfänger erreicht. Aus Sicht des Senders und des Kerns ist die Kommunikation abgeschlossen und die Nachricht erfolgreich zugestellt, auch wenn der Routertask diese im Folgenden wegwirft oder andere Fehler verhindern, dass sie den Empfänger erreicht.

Deceiving Der Routertask auf Empfängerseite muss die Nachricht so zustellen, dass es aussieht, als würde sie vom eigentlichen Sender kommen und nicht vom Routertask. Dafür wird der Sendeoperation ein weiterer Parameter mitgegeben: die ID des ursprünglichen Senders. Diese setzt der Kern als Absender der Nachricht, wenn die Nachricht den Empfänger erreicht.

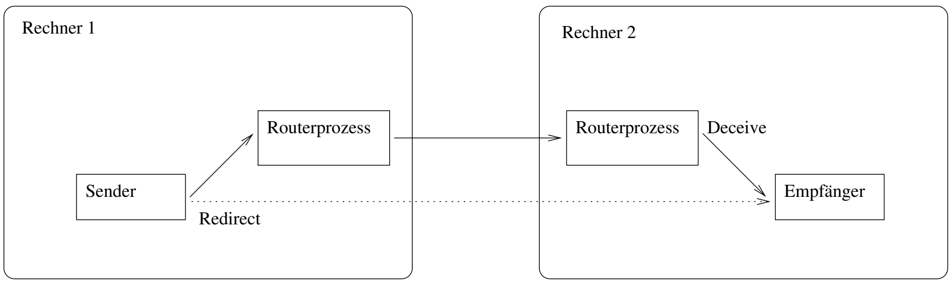

Abbildung 8: *Zur Implementierung der netzwerktransparenten IPC sind nur Teile des "Clans und Chiefs"-Konzept nötig: Redirecting und Deceiving.*

Mittels dieser zwei Änderungen des Kerns ist die Implementierung einer netzwerktransparenten IPC möglich. Eine Implementierung ist im folgenden Kapitel beschrieben.

Mittels Redirect und Deceiving wird die IPC wie beim Konzept der "Clans und Chiefs" von einem anderen Thread abgefangen und asynchron weitergeleitet. Das bedeutet, dass auch die Semantik der IPC der Semantik einer Kommunikation über Clangrenzen entspricht. Einzige Besonderheit ist der Fehlercode NOT_EXISTENT. Da sich der Empfängerthread nicht auf dem gleichen Rechner wie der Sender befindet, kann der Kern nicht feststellen, ob der Empfänger existiert. Die Nachricht wird also immer an einen Routertask zugestellt. Der Fehlercode NOT_EXISTENT kann bei einer entfernten Kommunikation nicht mehr vorkommen.

Dies wirkt sich auch auf die Semantik des Callaufrufes aus. Trat im lokalen Fall der Fehlercode NOT_EXISTENT auf, bedeutete das, der Server existiert nicht, was der Kern bei einem entfernten Funktionsaufruf jedoch nicht feststellen kann. Die Nachricht wird an den Routerprozess umgeleitet, welcher sie dann aber nicht zustellen kann. Der Sender bekommt darüber keine Rückmeldung, sondern bekommt nach Ablauf des angegebenen Timeouts die Fehlermeldung L4_IPC_RETIMEOUT.

Im Abschnitt 2.3.2 wurde bereits gezeigt, dass die Semantik eines Call-Aufrufes mittels direkten IPC und IPC über Clangrenzen gleich ist. Die Benutzung von netwerktransparenten IPCs mittels Redirect und Deceit ändert gegenüber den IPCs über Clangrenzen nur die Interpretation des Fehlercodes NOT_EXISTENT. Daraus folgt, dass durch eine Umdefinierung der Fehlercodes die Semantik eines Call-Aufrufes unter Verwendung netzwerktransparenter IPC in die Semantik überführt werden kann, die mittels direkter IPC erreicht wird. Die Abbildung der Fehlercodes im Fall der direkten IPC in die Fehlercodes im Fall der netzwerktransparenten IPC ist folgende:

$$f : Errorcodes \to Errorcodes \setminus \{\text{L4\_IPC\_NOT\_EXISTENT}\}$$
$$f : x \mapsto \begin{cases} \text{L4\_IPC\_RETIMEOUT} & falls \quad \text{x=L4\_IPC\_NOT\_EXISTENT} \\ x & sonst \end{cases}$$

3.2.4 Transparente Monitore

Ein grosser Nachteil an der Implementierung der netzwerktransparenten IPC durch Redirect und Deceiving ist die Änderung der Semantik. Sobald die Nachricht vom Routertask entgegengenommen wurde, läuft der Sender weiter, als wenn die Nachricht ordnungsgemäss vom Empfänger erhalten wurde. Es ist jedoch nicht sichergestellt, ob der Routertask die IPC wirklich zustellen kann. Zur Erhaltung der Semantik muss der Sender so lange blockiert bleiben, bis die Nachricht dem Empfängerthread zugestellt wurde. Eventuell auftretende Fehler oder die erfolgte Zustellung müssen also vom entfernten Rechner zurückgemeldet und dem Sender mitgeteilt werden. Das Konzept der transparenten Monitore ist dazu in der Lage. Solange der Monitor die Nachricht bearbeitet, bleibt der Sender blockiert. Der Sender verbleibt in diesem Zustand solange der Monitor die Nachricht über das Netzwerk versendet und auf die Bestätigung wartet. Ist ein Fehler bei der Zustellung aufgetreten, kann er eine FehlerIPC an den Sender schicken, dieser bekommt dann den Fehlercode vom Kern mitgeteilt. Ist die IPC ordnungsgemäss beim Empfänger angekommen, muss die Blockierung des Senders aufgehoben werden. Beim normalen Einsatz der transparenten Monitore erfolgt diese Entblockung, sobald der Monitor die Nachricht an den Empfänger weitersendet. Es ist dem Monitor auch möglich, den Empfänger zu ändern und so die Nachricht an einen anderen Prozess zu senden. Das kann man ausnutzen und bei erfolgreicher Zustellung der Nachricht auf dem Zielrechner die wartende IPC an eine Datensenke umleiten. Das ist ein Thread, der nur Nachrichten empfängt. Sobald die IPC dort eintrifft, läuft der Sender weiter und bekommt die erfolgreiche Zustellung gemeldet.

3.2.5 Vergleich

Das Konzept der transparenten Monitore wurde bisher noch nicht umgesetzt. Die hier beschriebene Vorgehensweise kann deshalb noch nicht implementiert werden. Sobald eine Version der Monitore

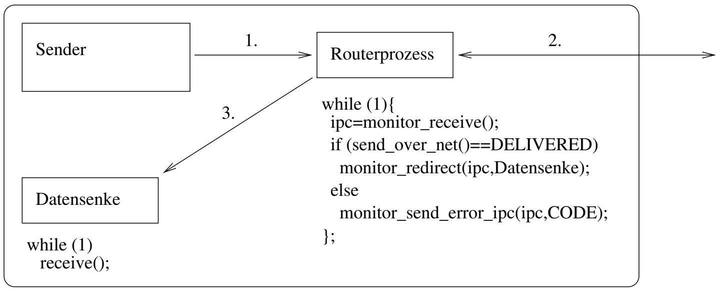

Abbildung 9: *Netzwerktransparente IPC mittels transparenter Monitore. Der Sender blockiert, bis die Nachricht an die Datensenke weitergeleitet wurde oder eine FehlerIPC gesendet wurde.*

für Fiasco zur Verfügung steht, kann die Implementierung der Netzwerktransparenten IPCs mittels Redirect und Deceiving mit kleineren Anpassungen übertragen werden.

3.3 Kommunikationsinfrastruktur

Die einzelnen Routerprozesse auf jedem Rechner des Netzwerkes müssen miteinander kommunizieren. Hierbei sind zwei prinzipielle Herangehensweisen möglich. Einerseits können die Prozesse direkt miteinander kommunizieren, andererseits kann eine zentrale Instanz die Weiterleitung der Nachrichten übernehmen. Beide Ansätze werden im Folgenden verglichen.

3.3.1 Punkt-zu-Punkt Kommunikation

Bei einer "Punkt-zu-Punkt" Kommunikation sendet jeder Routerprozess die Daten direkt an den Routerprozess des Rechners, auf dem der Empfängerthread läuft. Offensichtlich ist das die schnellste Möglichkeit, die Daten zu übertragen. Jeder Rechner muss dabei jedoch auch die Verbindungen zu den anderen Rechnern unterhalten. Jeder Routerprozess muss dazu die Adressen der anderen Rechner kennen. Dafür muss ein Mapping vorgenommen werden, welches für jede HostID die zugehörige Netzwerkadresse ausweist. Dies könnte eine Liste sein, die in jedem Routerprozess gespeichert ist oder ein zentraler Verzeichnisdienst. Dafür könnten bestehende Systeme wie DNS oder LDAP genutzt werden. Wenn der Routerprozess eine Nachricht für einen Rechner erhält, mit dem er derzeit noch keine Verbindung unterhält, stellt er beispielsweise eine DNS-Anfrage nach host4.l4cluster.tu-dresden.de und bekommt die IP-Adresse des Rechners mitgeteilt. Der dortige Routerprozess ist dann über einen vorher vereinbarten Port zu erreichen. Die Adresse des DNS-Servers wurde dem Rechner während des Bootvorganges vom DHCP-Server mitgeteilt. So können sich die Routerprozesse gegenseitig finden.

Bei der Implementierung netzwerktransparenter IPC mit "Clans und Chiefs" tritt eine Semantikänderung auf. Die Nachrichten werden nicht mehr synchron, sondern asynchron übertragen. Der Sendeaufruf kehrt also zurück, sobald die Nachricht dem Chief übergeben wurde. Anschliessend werden die Daten mit relativ langen Latenzzeiten über das Netzwerk gesendet. Hierbei kann es nun vorkommen, dass sich Nachrichten überholen, wie in Abbildung 10 angedeutet. Durch die Einführung von Zeitstempeln könnte die Auslieferung der Nachrichten in der falschen Reihenfolge verhindert werden.

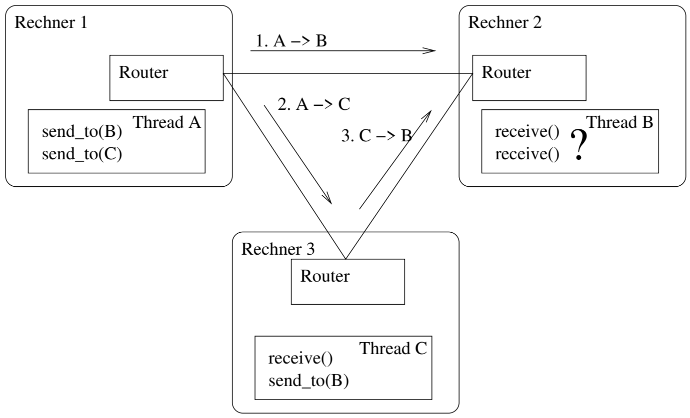

Abbildung 10: *Bei "Punkt-zu-Punkt"-Verbindungen kann es zu Überholungen von Nachrichten kommen. Bei direkter IPC muss Nachricht 1 vor Nachricht 3 eintreffen. Mit der Semantik von "Clans und Chiefs" kann Nachricht 3 zuerst eintreffen.*

Mit der Einführung migrierender Prozesse entsteht ein weiteres Problem: aus der ThreadID lässt sich nicht mehr der aktuelle Ausführungsort des Threads bestimmen. Es muss also ein zusätzliches Protokoll implementiert werden, um den Zielrechner der IPC zu finden. Beispielsweise kann der Rechner kontaktiert werden, auf dem der Thread ursprünglich gestartet wurde.

3.3.2 Zentraler Gateway

Eine andere Möglichkeit, die Nachrichten dem richtigen Host zuzustellen, ist, einer zentralen Komponente die Aufgabe der Verteilung der Nachrichten zu überlassen. Jeder Routerprozess schickt die Nachrichten, die er von lokalen Threads für Prozesse auf anderen Rechnern erhält, an einen ausgezeichneten Rechner, im Folgenden als Gateway bezeichnet. Die Adresse des Gateways muss allen Routerprozessen bekannt sein und kann ihnen beispielsweise beim Programmstart als Parameter mitgegeben werden. Der Gateway kennt alle angeschlossenen Rechner und kann auch mit dynamischen Netzwerkadressen umgehen, da sich die Rechner bei ihm anmelden. Der Vorteil eines solchen Routings ist die Einfachheit. Die einzelnen Routerprozesse brauchen keine Fehlerbehandlung durchzuführen, diese erfolgt zentral auf dem Gateway. Ein weiterer Vorteil ist, dass durch den zentralen Austauschpunkt auch eine totale Ordnung der Nachrichten erreicht wird. Diese werden nach dem Zeitpunkt des Eintreffens beim Gateway sortiert und in der gleichen Reihenfolge weitergesendet. Eine Überholung der Nachrichten, wie im vorigen Abschnitt beschrieben, kann nicht mehr auftreten.

Bei Einführung von migrierenden Prozessen hat der einzelne Gateway ausserdem den Vorteil, dass er an einer zentralen Stelle die Liste führen kann, welcher Thread gerade auf welchem Rechner abgearbeitet wird. Ein Protokoll zum Finden des aktuellen Ausführungsortes in jedem Routerprozess entfällt also.

Nachteilig wirkt sich jedoch die doppelte Übertragung der Nachrichten aus: einmal zum Gateway und einmal zum Zielrechner. Dies erhöht die Laufzeiten und belastet die Übertragungskapazität des darunterliegenden Netzwerkes stärker. Ausserdem ist der Gateway auch ein "Single Point of Failure": das gesamte System bricht also beim Ausfall dieser einen Komponente zusammen.

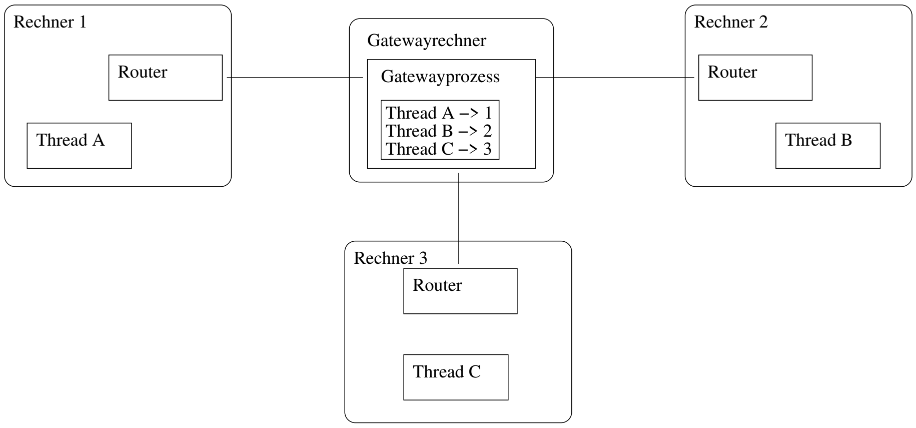

Abbildung 11: *Alle nicht lokalen Nachrichten werden an den Gatewayprozess geschickt, der sie weiterroutet. Wenn Prozesse migrieren, wird an einer zentralen Stelle die Information über den Ausführungsort gehalten.*

4 Implementierung

Im vorigen Kapitel wurden mehrere Ansätzen zur Implementierung netzwerktransparenter IPCs für Fiasco entwickelt und verglichen. Im Folgenden wird die Implementierung von "Redirect und Deceit" beschrieben. Hiermit kann die Semantik einer Kommunikation über Clangrenzen erreicht werden. Die Kopplung der Rechner wird durch einen zentralen Gatewayprozess erfolgen.

Im ersten Abschnitt werden die Änderungen am Kern beschrieben. Durch diese wird das Redirecting und Deceiving von IPCs ermöglicht. Der zweite Abschnitt behandelt den Routerprozess. Dieser empfängt die redirecteten IPCs, versendet sie über das Netzwerk und stellt sie auf dem Zielrechner mittels Deceiving an den endgültigen Empfänger zu. Der dritte Abschnitt befasst sich mit der Implementierung eines Testprogramms, mit dessen Hilfe die Korrektheit, Robustheit und Geschwindigkeit gemessen werden kann. Die Auswertung der Tests erfolgt dann im nächsten Kapitel.

4.1 Änderungen am Fiasco-Kern

4.1.1 Redirecting

Wie im Kapitel 3.2.3 beschrieben, soll der Kern Nachrichten, die an einen Thread auf einem anderen Rechner gerichtet sind, an einen ausgezeichneten Prozess zustellen. Dieser Prozess ist der "Preempter" des betrachteten Threads. Diese Festlegung wurde getroffen, da der Preempter in der derzeitigen Implementierung nicht benutzt wird und dieses Feld noch frei war. Eine andere Möglichkeit wäre, eine systemweite Variable zu benutzen, welche den Routingprozess angibt. Wie bereits diskutiert, werden nun alle IPCs an nicht lokale Threads zum Preempter umgeleitet. Betrachtet man ein System ohne Migration, müssen nur IPCs an Threads mit einer anderen als der lokalen HostID umgeleitet werden. Mit Migration müssen Nachrichten zu allen nicht auf dem Rechner befindlichen Threads dem Preempter zugestellt werden. In der Implementierung wurde sich für letzteres entschieden, da dies die zukunftssichere Variante ist.

Die Umleitung der IPC an den Preempter erfolgt durch folgenden Code:

```
if (!partner || partner->state() == Thread_invalid)
{
 if (regs->thread_esi_edi()->id.site!=0) //do no redirect for "local" calls
  partner=_preempter;
}
```

Hierbei wird im Falle, dass der Empfänger nicht existiert, der Empfänger durch den Preempter ersetzt. Wenn die Nachricht an einen lokalen Thread mit der HostID 0 gesendet wird, erfolgt keine Umleitung an den Preempter. Zusätzlich wird der originale Empfänger gespeichert. Dieser muss beim Receive()-Systemcall mit übergeben werden. Die Spezifikation [11] des Systemaufrufes ist jedoch an dieser Stelle unvollständig, es ist keine Möglichkeit vorgesehen, diesen zusätzlichen Parameter zurückzugeben. Deshalb wurde die Spezifikation wie folgt erweitert: der originale Empfänger der IPC wird in den Registern ESI und EDI zurückgegeben. Das wird durch diese Zeilen erreicht:

```
dest_regs->ecx = sender_regs->esi;
dest_regs->ebp = sender_regs->edi;
dest_regs->eax |= L4_IPC_REDIRECT_MASK;
```

Um den zusätzlichen Parameter des Systemaufrufes auch ansprechen zu können, muss ein zusätzliches C-Binding eingeführt werden:

```
L4_INLINE int
l4_i386_ipc_chief_wait(l4_threadid_t *src,l4_threadid_t *to,
                       void *rcv_msg, dword_t *rcv_dword0, dword_t *rcv_dword1,
                       l4_timeout_t timeout, l4_msgdope_t *result)
```

Zusätzlich muss der Assembler-Shortcut für den IPC-Pfad ausgeschaltet werden, da dieser im Receiveteil die Register ESI und EDI überschreibt und dadurch falsche Werte an den Empfänger geliefert werden.

4.1.2 Deceiving

Um das Deceiving anzusprechen, muss ein weiteres C-Binding eingeführt werden. Ein entsprechendes Binding existierte schon für den 2.7.3-gcc-Compilern und wurde nur in die Bindings für den gcc der Version 2.9.5 kopiert:

```
L4_INLINE int
l4_i386_ipc_send_deceiting(l4_ipc_deceit_ids_t ids,
                           const void *snd_msg,
                           dword_t snd_dword0, dword_t snd_dword1,
                           l4_timeout_t timeout, l4_msgdope_t *result)
```

Nun muss nur noch der Absender durch den gefälschten Absender (faked ID), welcher auf dem Userstack übergeben wird, ersetzt und das Deceiving-Bit gesetzt werden.

Dies wird durch folgende Codezeilen erreicht:

```
if (! config::deceit_bit_disables_switch
    && (sender_regs->eax & 1)) {
 dest_regs->esi=((unsigned *)((thread_regs_t *)sender_regs)->esp)[0];
 dest_regs->edi=((unsigned *)((thread_regs_t *)sender_regs)->esp)[1];
 dest_regs->eax |= L4_IPC_DECEIT_MASK; // indicate deceiving
}else{
 * dest_regs->thread_esi_edi() = id();
};
```

4.1.3 Setzen der HostID

Jedem Rechner im Netzwerk muss eine eindeutige Nummer zugeordnet werden.

Das könnte beispielsweise nach dem Booten des Kerns geschehen, wenn die Netzwerkkarte initialisiert wurde und so die eindeutige Netzwerkkartenadresse benutzt wird, um die HostID zu generieren. Dies ist problematisch, da zu diesem Zeitpunkt bereits Prozesse aktiv sind, welche ThreadIDs in Variablen gespeichert haben. Wenn sichergestellt ist, dass bis zur Initialisierung der HostID nur "lokale" ThreadIDs, also mit der HostID 0, verwendet wurden, wäre es möglich, diese Vergabe der HostIDs zu benutzten.

In dieser Arbeit wird ein anderer Weg beschritten. Die HostID wird dem Kern beim Booten als Parameter mitgegeben:

```
module= (fd0)/main -hostid=3
```

Die übergebene Nummer wird als Teil der ID jedes Threads benutzt, um diese systemweit eindeutig zu machen. Bei der Erzeugung jeden Threads muss also sichergestellt werden, dass die HostID eingetragen wird:

```
PROTECTED inline
sender_t::sender_t(const l4_threadid_t& id)
  : _id (id), sender_next (0)
{
 if ((_id.lh.low==0)&&(_id.lh.high==0)) return;
 _id.id.site=config::hostid;
}
```

4.2 Routertask

Der Routertask empfängt die IPCs, die vom Kern an ihn umgeleitet wurden, verpackt sie in Netzwerknachrichten und schickt sie zu dem Rechner, für den die Nachricht bestimmt war. Umgekehrt empfängt er aus dem Netzwerk eingehende Nachrichten, dekodiert sie und stellt sie dem Empfänger zu.

4.2.1 Designentscheidungen

Es bestehen mehrere Möglichkeiten der Implementierung des Routertasks. Einerseits ist zu entscheiden, welche der vorgestellten Kommunikationsinfrastrukturen eingesetzt, andererseits, welches Semantiklevel angestrebt wird.

Die beste Nachbildung der Semantik wird durch Verwendung der transparenten Monitore erreicht. Diese sind jedoch nicht im Fiasco-Kern implementiert und würden umfangreiche Änderungen von Kerndatenstrukturen nachsichziehen. Die Implementierung mittels Redirect und Deceit, die die Semantik einer IPC-Operation über Clangrenzen nachbilden kann, ist mit weniger Aufwand fertigzustellen. Die Implementierung wird in den folgenden Abschnitten beschrieben.

Die Wahl der Kommunikationsinfrastruktur ist ebenfalls implementierungstechnisch bedingt. Dem verwendeten IP-Stack ist es nicht möglich, bei einem accept()-Aufruf einen Timeout-Parameter anzugeben oder vorher nachzuprüfen, ob der Aufruf blockieren wird. Dies bedeutet, dass das Programm an dieser Stelle stehenbleiben wird, bis sich ein anderer Rechner versucht zu verbinden. Das stellt ein Problem bei der "Punkt-zu-Punkt"-Kommunikation dar, da hier der Routerprozess immer auf potentiell eingehende Verbindungen warten muss. Praktisch könnte man das Problem lösen, indem man einen eigenen Thread für diese Aufgabe starten würde, der solange blockiert, bis es eine eingehende Verbindung gibt. Dies scheitert jedoch daran, dass der verwendete IP-Stack nur von dem Thread benutzt werden kann, der ihn initialisiert hat. Aus diesem Grund fiel die Entscheidung zugunsten des zentralen Gatewayprozesses. Hierbei müssen sich die Routerprozesse nur zu diesem verbinden und nicht auf eingehende Verbindungen warten.

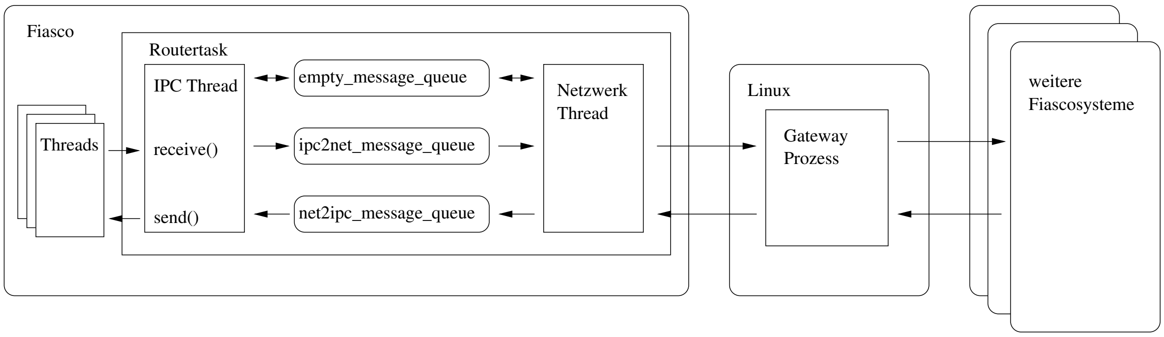

Abbildung 12: *Generelles Konzept des Routertasks.*

4.2.2 Pufferverwaltung

Der Hauptteil des Routertasks ist die Pufferverwaltung. Alle Nachrichten, die das System verarbeitet, werden in Messagepuffer gespeichert, welche ihrerseits in Queues bis zur Abarbeitung gehalten werden. Die Funktionen der Pufferverwaltung sind "threadsave", das heisst, sie können aus mehreren Threads gleichzeitig aufgerufen und zur Synchronisation und Datenaustausch benutzt werden. Beim Programmstart wird vom Heap ein Speicherbereich ausgegriffen, auf welchem eine eigene Speicherverwaltung initialisiert wird. Alle Puffer werden zu Beginn in die "empty_message_queue" eingehängt. Diese verwaltet alle z.Z. nicht benutzten Puffer. Zwei weitere Queues sind definiert: die "ipc2net_message_queue" und "net2ipc_message_queue". Die Erste beinhaltet Nachrichten, welche vom IPC Thread empfangen wurden und noch ins Netzwerk gesendet werden müssen, die Zweite diejenigen, die von anderen Rechnern für lokale Prozesse empfangen wurden.

Zwei Funktionen dienen dem Entnehmen und Einstellen der Nachrichten in die Queues:

```
void get_a_message2(struct spool_message_queue_t *q,struct spool_message_t **m);
void enqueue_message2(struct spool_message_t **m,struct spool_message_queue_t *q);
```

Ausserdem sind Funktionen zum Ein- und Auspacken der IPC-Daten und Unterstützung zum sequenziellen Ein- und Auslesen zur Übertragung ins Netzwerk vorgesehen:

```
int setdata(struct spool_message_t *m,dword_t *dwords,int dwordscount,
            l4_strdope_t* strings,int stringcount);
int getdata(struct spool_message_t *m,dword_t *dwords,int *dwordscount,
            l4_strdope_t* strings,int *stringcount);

void init_reader(struct spool_message_t *m);
void get_readpointer(struct spool_message_t *m,char**p,int*l);
void seek_readpointer(struct spool_message_t *m,int n);

void init_writer(struct spool_message_t *m);
void get_writepointer(struct spool_message_t *m,char**p,int*l);
void seek_writepointer(struct spool_message_t *m,int n);
void end_writer(struct spool_message_t *m);
```

Die Benutzung der Funktionen ist sehr einfach. Zuerst wird init_reader() aufgerufen, der den Readpointer an den Anfang setzt. Nun liefert get_readpointer() einen Zeiger auf einen Datenpuffer mit l Bytes zurück. Wurden diese gelesen, wird der Readpointer mit seek_readpointer() weitergesetzt und wieder get_readpointer() aufgerufen, bis die Funktion die Länge 0 zurückgibt. Die gesamte Nachricht ist nun ausgelesen. Analog erfolgt die Benutzung des Writers, allerdings muss zum Abschluss noch end_writer() aufgerufen werden um das Schreiben des Puffers zu beenden.

4.2.3 IPC-Thread

Dieser Thread nimmt die vom Kern umgeleiteten IPCs entgegen. Er erlaubt das Empfangen von Short- und StringIPCs. Nachrichten mit Flexpages werden an dieser Stelle abgewiesen, da es z.Z. noch keine Möglichkeit gibt, diese über das Netzwerk in den Adressraum des Empfängers zu mappen. Der Sender bekommt eine korrekte Fehlernachricht.

Die Daten der empfangenen Nachricht werden dann in einem Puffer gespeichert und dieser in die ausgehende Warteschlange gestellt. Danach wird in der eingehenden Warteschlange nachgesehen, ob Nachrichten von anderen Rechnern eingetroffen sind. Diese werden dann an den Empfänger zugestellt und dabei die ursprüngliche AbsenderID mittels Deceiving fälscht. Diese beiden Funktionen müssen in einem Thread ablaufen. Ein Thread, welcher netzwerktransparente IPC benutzen soll, muss den IPC-Thread als Preempter gesetzt haben. Dann werden alle nichtzustellbaren Nachrichten hierher umgeleitet. Führt ein Thread ein "closed receive" aus, erwartet also nur Nachrichten von einem angegebenen Kommunikationspartner. Existiert dieser Partner nicht, muss der Kern den Aufruf so ändern, dass auf Nachrichten vom Preempter gewartet wird. Dieser kann dann IPCs aus dem Netzwerk mittels Deceiving zustellen.

Da die beiden Funktionen des IPC-Treads, das Senden und Empfangen, sich gegenseitig behindern, müssen die Timeoutwerte aufeinander abgestimmt werden. So darf es nicht vorkommen, dass der Routerprozess lange blockiert, weil er eine aus dem Netz empfangene Nachricht für einen lokalen Thread zuzustellen versucht und dieser gerade nicht bereit ist. Während der Blockade kann er keine Nachricht von anderen lokalen Threads entgegennehmen, so dass diese mit einem Timeoutfehler zurückkehren. Für das Senden von Nachrichten aus dem Netzwerk an einen lokalen Thread wurde ein Timeout von Null festgesetzt; das heisst, der Systemaufruf kehrt sofort zurück, wenn der Empfänger nicht bereit ist, die Nachricht zu empfangen. Wenn die Nachricht nicht zugestellt werden konnte, wird die Nachricht wieder in die Warteschlage zurückgestellt, und es wird beim nächsten Schleifendurchlauf erneut versucht, sie zuzustellen. Durch dieses Verhalten können keine Nachrichten an Prozesse geschickt werden, die ihre receive()-Operation auch mit einem Timeout von Null aufrufen, in der Annahme, dass der Sender mit einem langen Timeout die Nachricht für ihn "zur Abholung" bereithält. Dies ist jedoch als schlechter Programmierstil anzusehen und kann auch im Fall einer direkten IPC-Kommunikation zu Problemen führen. Beispielsweise könnte der senderseitige send()-Aufruf von einem l4_thread_ex_regs()[9] gerade dann unterbrochen werden, wenn der Empfänger die Nachricht entgegennehmen will. Wenn der Empfängerprozess ein Timeout hat, welches sehr viel grösser ist als der Abstand zweier Sendeversuche des Routerprozesses, sollte es nicht zu den beschriebenen Timeout-Fehlern kommen. Ein weiterer Punkt, welcher dafür spricht, dass der Kommunikationspartner mit hohen Timeoutwerten arbeiten muss, ist, dass die Nachricht für die Übertragung über das Netzwerk Zeit benötigt.

[9]Mit l4_thread_ex_regs() kann man Parameter eines Threads verändern. Bei diesem Systemcall werden auch gerade laufende IPCs unterbrochen.

Für den Empfangsteil, der redirectete Nachrichten von lokalen Prozessen empfängt, wurde ein Timeout von einigen Millisekunden zugelassen.[10] Ohne dieses Timeout wäre die Sende- und Empfangsschleife ohne anstehende Aufträge eine Leerschleife. Das Timeout sorgt dafür, dass der Thread schläft und die anderen Prozesse mehr Rechenzeit bekommen.

Mit der in [15] vorgeschlagenen, aber noch nicht implementierten, Weiterleitung von Anfragen könnte man die Sende- und Empfangsoperation trennen und sogar mehrere Threads für das Senden und Empfangen einsetzen.

Der Programmcode der Hauptschleife des IPC-Threads ist folgender:

```
while (1) {
 if (rmsg==0) {
  rmsg=get_a_message(empty_message_queue);
 };
 if (rmsg!=0){
// init message
  res = l4_i386_ipc_chief_wait(&src,&to,&msg,&msg.dwords[0],&msg.dwords[1],
                               L4_IPC_TIMEOUT(255,12,255,12,0,0), &result );
  if (res==0){
   i=setdata(rmsg,msg.dwords,result.md.dwords,msg.strings,result.md.strings);
   rmsg->from=src;
   rmsg->to=to;
   enqueue_message(rmsg,ipc2net_message_queue);
   rmsg=0;
  }; //res==0
 }; //rmsg!=0

 smsg=get_a_message(net2ipc_message_queue);
 if (smsg!=0){
// init message
  k=getdata(smsg,msg.dwords,&i,msg.strings,&j);
  deceiteid.true_src=smsg->from;
  deceiteid.dest=smsg->to;
  if ((i<=2)&&(j==0)){
   res = l4_i386_ipc_send_deceiting(deceiteid,0,msg.dwords[0],msg.dwords[1],
                                    L4_IPC_TIMEOUT(255,0,255,12,0,0), &result);
  } else {
   res = l4_i386_ipc_send_deceiting(deceiteid,&msg,msg.dwords[0],msg.dwords[1],
                                    L4_IPC_TIMEOUT(255,0,255,12,0,0), &result);
  };
  if(res){
   enqueue_message(smsg,net2ipc_message_queue); // couldnt send, so put back in list
  }else{
```

[10] Das Programm wurde unter VMware entwickelt, welches einen Rechner nur simuliert. Die Wartezeit eines IPC-Systemaufrufes schwankten je nach Systemauslastung sehr deutlich und haben in der Praxis wenig mit den in [11] spezifizierten Zeiten gemeinsam.

```
   enqueue_message(smsg,empty_message_queue); //send is done.
  };
 }; //smsg!=0
}; //while(1)
```

4.2.4 Netzwerkkommunikations-Thread

Der Netzwerkthread ist dafür verantwortlich, die Nachrichten über das Netzwerk zuzustellen und zu empfangen. Dafür wird der Netzwerktreiber aus dem Paket Ether verwendet. Wie im Kapitel 2.4.6 festgelegt wurde, soll eine Zwischenschicht programmiert werden, die die gleichen Funktionsaufrufe bietet, wie der noch in Entwicklung befindliche TCP/IP-Stack für das Oshkosh-Projekt.

Bevor die eigentliche Entwicklung am Netzwerktreiber beginnen kann, muss zuerst die Unterstützung der Lance-Netzwerkkarte, welche VMware simuliert, gesichert werden. In der Version 0.97 des OSKit ist ein Bug enthalten, der die Netzwerkkarte mehrfach erkennt. Die Routine, die die Karte initialisiert, generiert so viele Ethernetdevices, bis der Speicher voll ist und der Rechner abstürzt. Die Änderung ist simpel und funktioniert nur für eine Netzwerkkarte. In der Erkennungsschleife wird ein Zähler eingebaut, der dafür sorgt, dass die Erkennung nach einer gefundenen Netzwerkkarte abbricht:

```
#ifdef OSKIT
 static int first = 1;
 if (first) {
  first = 0;
 } else {
  return -ENODEV;
}
#endif
```

Durch diese Änderung wurde der Ether-Treiber einsetzbar, er zeigte jedoch im Laufe der Entwicklung seltsame, nicht immer nachvollziehbare Eigenschaften. Beispielsweise versagte l4_thread_ex_regs() bei etwa fünf Prozent der Aufrufe den Dienst und erzeugte keinen neuen Thread. Ebenso lieferte die Funktion l4_myself() nicht die eigene ThreadID zurück, sondern die des Pagerprozesses. Eine intensive Fehlersuche endete schliesslich darin, dass Teile des Ethertreibers neu geschrieben wurden. Der eigentliche Treiber ist in einem eigenen Task, netipcn, ausgelagert.

Er initialisiert die Hardware mit den OSKit Funktionen:

```
 oskit_clientos_init();
 osenv = start_osenv();
 oskit_dev_init(osenv);
 oskit_linux_init_osenv(osenv);
 DRIVER_INIT(ETHER_DRIVER)();
 oskit_dev_probe();
 err = oskit_freebsd_net_init(osenv, &socket_create);
 ndev = osenv_device_lookup(&oskit_etherdev_iid, (void***)&etherdev);
```

Diese Routine initialisiert ebenfalls den TCP/IP-Stack. Die Funktionen des OSKit zur Erzeugung und Nutzung eines TCP-Sockets können jetzt benutzt werden. Die Funktionsaufrufe sind jedoch nicht 4.4BSD-konform. Da eine 4.4BSD-konforme Schnittstelle eine der Designentscheidungen war, muss noch eine Protokollumsetzung erfolgen. Dazu läuft der Netzwerktreiber in einer Endlosschleife, welche auf Anfragen der Clientbibliothek wartet und diese dann auf die OSKit-Funktionen abbildet. Das soll hier am Beispiel der Funktion close() beschrieben werden. Während das OSKit zur Identifizierung eines Sockets einen Zeiger auf eine oskit_socket_t-Struktur verwendet, nutzt 4.4BSD einen Integerwert. Die Speicherung der Strukturen des OSKits erfolgt in dem Array "allsockets". Die Endlosschleife wartet mit l4_i386_ipc_wait auf Anfragen, dekodiert die Parameter und ruft danach die OSKit-Funktion oskit_socket_release auf. Danach wird der Rückgabewert gesetzt und das Ergebnis mit l4_i386_ipc_send zurückgesendet.

```
oskit_socket_t * allsockets[MAXSOCKETS];
//...
 while(1){
  res = l4_i386_ipc_wait(&src,&m,&p1,&p2,L4_IPC_NEVER,&dope);
  input8=(p1>>8)&0xff;
  input16=(p1>>16)&0xffff;
  switch (p1&0xf) {
/...
   case 2:
    if (ipdebug) printf("close(%u)\n",input8);
    if (allsockets[input8]==0) {p1=-1;goto done;};
    oskit_socket_release(allsockets[input8]);
    allsockets[input8]=0;
    p1=0;
    goto done;
/...
   };
done:
  res = l4_i386_ipc_send(src,L4_IPC_SHORT_MSG,p1,p2,L4_IPC_NEVER,&dope);
 } //while(1)
```

Analog verlaufen auch die Aufrufe für die anderen Funktionen. Zu beachten ist, dass alle angelegten Sockets automatisch mit einem Sende- und Empfangstimeout versehen werden. Dies muss erfolgen, damit die read()- und write()-Operationen nicht blockieren. Das OSKit hat keine Funktion, die vergleichbar mit dem in BSD4.4 definierten select()-Aufruf ist, mit dem vor dem read()- oder write()-Aufruf festgestellt werden kann, ob dieser blockieren wird. Bei der Portierung auf den Netzwerktreiber Oshkosh mit dem BSD4.4-konformen TCP/IP-Stack muss also entweder vor jeder Lese- oder Schreiboperation ein select() aufgerufen werden oder alle Sockets auch in einen nicht blockierenden Modus geschaltet werden. Das kann mit folgendem Code erfolgen:

```
 fcntl(sock,F_SETFL,O_NONBLOCK|fcntl(s,F_GETFL));
```

Dieser Netzwerktreiber muss die TaskID 6 besitzen. Der Prozess registriert sich zwar beim Nameserver unter dem Namen "netipcn", dabei wird jedoch aus unbekannten Gründen die falsche

TaskID "4" übergeben. Im Routerprozess musste deshalb die ID des Netzwerktreibers fest einprogrammiert werden.

```
if (names_waitfor_name("netipcn", &net_thread_id,30000)){
  if (net_thread_id.id.task!=6){
    printf("according to names_waitfor_name() the netipcn is %u(%u/%u)\n",
     net_thread_id.id.task,net_thread_id.id.lthread,net_thread_id.id.site);
    printf("thats WRONG!, setting it to 6\n");
    net_thread_id.id.task=6;
  };
}else{
 \\ ...
};
```

Der Kommunikationsthread benutzt den TCP/IP-Stack, um eine Verbindung zum Gateway herzustellen. Dessen IP-Adresse und optional dessen Port werden dem Routerprozess als Kommandozeilenoption mitgegeben. Nach dem Verbindungsaufbau werden die lokale HostID und die HostID der gewünschten Gegenstelle zum Gateway übertragen. Die ID der gewünschten Gegenstelle wird nicht ausgewertet und kann für zukünftige "Punkt-zu-Punkt"-Verbindungen benutzt werden, um sicherzustellen, dass der richtige Host erreicht wird. Der Gateway bestätigt den Verbindungsaufbau. Ab diesem Zeitpunkt wird ein symmetrisches Protokoll benutzt, welches die Daten ohne zusätzliche Sicherung überträgt. Es wird der Flusskontrolle von TCP/IP überlassen, die Nachrichten sicher und in der richtigen Reihenfolge zu übermitteln. Durch die Symmetrie der Übertragung kann diese auch bei "Punkt-zu-Punkt"-Verbindungen eingesetzt werden, da nach dem Verbindungsaufbau egal ist, welcher Routerprozess die Verbindung initiierte.

4.2.5 Gatewayprozess

Der Gateway, der für das Weiterleiten der Nachrichten verantwortlich ist, wird für Linux entwickelt. Er kann eine bestimmte Anzahl von Routerprozessen bedienen. Die höchste HostID, welche sich verbinden darf, muss in der Kommandozeile übergeben werden, was für die Speicherreservierung notwendig ist. Als zweiter Parameter kann noch der Port angegeben werden, an dem er auf Verbindungsaufbauten aus dem Netzwerk wartet. Der Aufruf erfolgt folgendermassen:

```
gateway MaxHostID [Port]
```

Nachdem sich die Routerprozesse verbunden haben, nimmt der Gateway Nachrichten von diesen entgegen und sendet sie an den richtigen Rechner weiter. Die Pufferverwaltung und grosse Teile des Kommunikationsthreads des Routerprozesses wurden wiederverwendet. Die Nachrichten werden nicht interpretiert, einzig Absender, Empfänger und für Debugzwecke die ersten zwei übertragenen Worte werden dekodiert. Jede empfangene Nachricht wird auf der Standardausgabe ausgeschrieben. Ausserdem wird die vergangene Zeit seit dem Empfang der ersten Nachricht angegeben. Dadurch ist eine Protokollierung der gesamten Kommunikation des Clusters möglich.

4.2.6 Testprogramm

Zur Messung der Geschwindigkeit der Datenübertragung werden zwei Programme geschrieben, die sich IPCs zusenden. Das erste Programm, im Folgenden als Receiver bezeichnet, wartet auf eingehende Nachrichten. Er berechnet die Summe aus den empfangenen Worten und die Summe aller in den Strings enthaltenen Bytes. Die beiden Ergebnisse werden an den den Sender zurückgesendet. Durch diese Berechnung kann der Sender überprüfen, ob die Daten korrekt übertragen wurden.

Das zweite Programm ist der Sender. Er generiert zufällige Nachrichten und schickt diese an den Receiver. Dann wartet er auf die Antwort und überprüft die zurückgegebenen Daten. Über Kommandozeilenparameter kann das Verhalten gesteuert werden:

Zielthread Mit drei Zahlen wird Task-, Thread- und Sitenummer des Receivers angegeben.

Daten Die folgenden Zahlen beschreiben die Anzahl der zu übertragenden Worte, gefolgt von der Anzahl der Strings. Die nächste Zahl gibt die Länge der Strings an.

Parallele Sendethreads Der nächste Parameter spezifiziert die Anzahl der gleichzeitig ablaufenden Sendeoperationen. Es werden mehrere Threads gestartet, die unabhängig voneinander Nachrichten an den Receiver schicken. Hierdurch kann Last simuliert werden, die bei mehreren unabhängigen Programmen entsteht.

Anzahl der Durchläufe Jeder Sendethread sendet nacheinander diese vorgegebene Anzahl von Anfragen an den Empfänger.

Keiner der Parameter ist optional. Der Aufruf erfolgt folgendermassen:

```
sender Task Thread Site Words Strings Stringlen NrThreads SendCount
```

5 Messung und Bewertung

5.1 Messanordnung

Die im vorigen Abschnitt beschriebenen Testprogramme werden eingesetzt, um die Leistungsfähigkeit der Implementierung zu testen. Dazu werden einerseits Latenzzeiten und andererseits die Bandbreite der Übertragung gemessen. Bei jeder Übertragung wird dazu auch die Korrektheit der übertragenen Daten geprüft. Dazu berechnet der Empfänger die Summe aller übertragenen Worte und Strings und schickt diese zum Sender zurück. Der Sender überprüft die Berechnung und kann dadurch Übertragungsfehler feststellen.

Für die Messung der Latenz wird von einem Sendethread eine ShortIPC vom Sender zum Empfänger gesendet und auf die Antwort in Form einer ShortIPC gewartet. Dieses "Pingpong" wird so lange wiederholt, bis eine ausreichende Messgenauigkeit erreicht wird.

Zur Messung der über das Netzwerk übertragbaren Bandbreite werden von mehreren Sendethreads lange StringIPCs an den Empfänger geschickt, welcher jedes Paket wieder mit einer ShortIPC bestätigt. Sowohl die Anzahl der Sendethreads als auch die Grösse der übertragenen Daten werden variiert. Gemessen wird mittels einer Stoppuhr, welche bei der Startausschrift des Senders gestartet wird und angehalten wird, sobald das letzte Paket empfangen wurde. Jede Messung wird drei mal wiederholt, die Ergebnisse gemittelt und jeweils auf volle Sekunden gerundet. Eine ausreichende Messgenauigkeit wird erreicht, wenn die Messung länger als 100 Sekunden dauert, die durch Messung und Rundung bedingten Fehler liegen dann unter einem Prozent.

Zur Messung werden zwei virtuelle Rechner durch je eine VMware-Instanz simuliert. Einer dieser Rechner führt den Sendeprozess aus, der andere den Empfängerprozess. Die beiden Prozesse übertragen die Daten durch einem Gatewayprozess, welcher auf einem dritten Computer unter Linux läuft. Die Rechner sind mit einem 100-MBit/s-Netzwerk verbunden. Obwohl die zwei kommunizierenden Fiasco-Systeme nur auf virtuellen Rechner simuliert werden, müssen die Daten trotzdem zweimal über das Netzwerk zum Gateway und zurück transportiert werden. Dies sollte die Ergebnisse nicht verfälschen.[11] Die Ausstattung der Rechner sind in Tabelle 1 aufgeführt. Tabelle 2 zeigt die von VMware simulierte Hardware.

	Host für Sendeprozess	Host für Empfänger	Gatewayrechner
Prozessor	Mobile Pentium II 366 MHz	Athlon 1000 MHz	Dual Pentium II 350 MHz
Speicher	196 MB	256 MB	384 MB
Netzwerkkarte	Intel EEPro100	Realtech	3Com 3c905B
IP-Adresse	DHCP	DHCP	192.168.1.1
Software	Linux 2.4.18, VMware WS 3.1	Linux 2.4.19, VMware WS 3.1	Linux 2.4.18

Tabelle 1: *Hardwareausstattung der Testumgebung.*

[11] Ursprünglich sollte das System mit mehreren echten Rechnern der Betriebssystemgruppe getestet werden, dies ist jedoch nach dem Hochwasser vom August 2002 und dessen katastrophalen Auswirkungen auf die Fakultät Informatik nicht möglich.

	Senderechner	Empfängerrechner
Prozessor	Celeron 366 MHz	Athlon 1000 MHz
Speicher	32 MB	32 MB
Netzwerkkarte	AMD Lance	AMD Lance
IP-Adresse	DHCP	DHCP
Software	Fiasco	Fiasco

Tabelle 2: *Virtuelle Hardwareausstattung der simulierten VMware Instanzen.*

In einer zweiten Messanordnung wird der Empfängerprozess auf dem gleichen virtuellen Rechner mitgestartet, der den Sendeprozess ausführt. Sie kommunizieren mittels direkter IPC miteinander. Dies ist die Vergleichsmessung, an der die vorigen Messungen gewichtet werden.

Als Drittes wird eine Messung vorgenommen, bei der der Routerprozess modifiziert wird. Statt Nachrichten aus der "ipc2net_message_queue" über das Netzwerk zu senden, werden sie modifiziert und in die "net2ipc_message_queue" gestellt. Es erfolgt keine Netzwerkübertragung, der Sendeprozess erhält jedoch falsche Daten zurück. Für die Zeitmessung spielt dies allerdings keine Rolle.

Mit dieser Messung kann der Overhead durch das Umleiten der Nachrichten und durch die Pufferverwaltung gemessen werden.

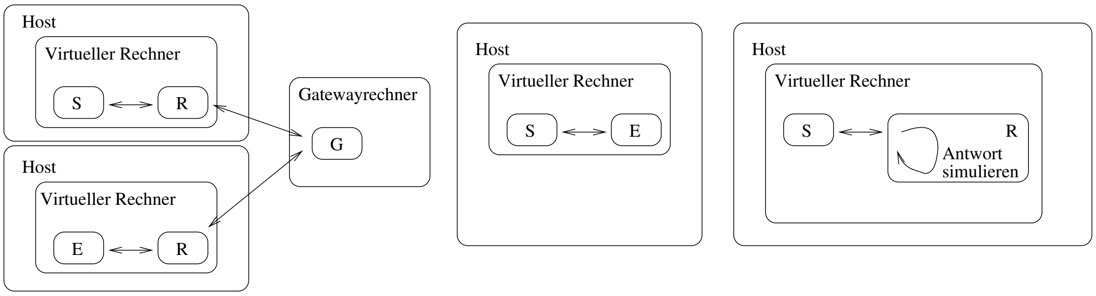

Abbildung 13: *Testanordnung:* **`Links`** *werden die IPCs über ein Netzwerk verschickt. In der* **`Mitte`** *werden Sender und Empfänger auf dem selben Rechner ausgeführt.* **`Rechts`** *wurde der Routerprozess modifiziert: Die Antworten werden sofort generiert.*

5.2 Messungen

Für die Messung der Latenz werden die beiden virtuellen Machinen der ersten Messanordnung mit folgenden Bootoptionen gestartet:

```
Sender

kernel= (fd0)/rmgr -sigma0
module= (fd0)/main -hostid=2
module= (fd0)/sigma0
module= (fd0)/names
```

```
Empfaenger

kernel= (fd0)/rmgr -sigma0
module= (fd0)/main -hostid=3
module= (fd0)/sigma0
module= (fd0)/names
```

```
module= (fd0)/netipcn                    module= (fd0)/netipcn
module= (fd1)/netipc 192.168.1.1         module= (fd1)/netipc 192.168.1.1
module= (fd1)/sender 8 0 3 2 0 5 X Y     module= (fd1)/receiver
```

Hierbei wird X, die Anzahl der Sendethreads, variiert. Der Wert Y gibt die Anzahl der Sendeoperationen an, die jeder Sendethread ausführt. Die Zahl wird so gewählt, dass die gemessenen Zeiten über 100 Sekunden liegen und so die Messungenauigkeit unter einem Prozent liegt. Die Ergebnisse sind in Tabelle 3 aufgeführt.

Sendethreads (X)	1	3	5	10	15	20	30
Durchläufe (Y)	100	100	400	400	400	400	400
Messwert	211 s	107 s	197 s	169 s	181 s	202 s	239 s
Latenz	2110 ms	1070 ms	493 ms	423 ms	453 ms	505 ms	598 ms

Tabelle 3: *Latenzzeiten für einen Call-Aufruf über Rechnergrenzen.*

Die zweite Messung soll die Geschwindigkeit bestimmen, mit der Daten über das Netzwerk übertragen werden. Die Bootoptionen der virtuellen Rechner in der ersten Messanordnung sind folgende:

```
Sender                                   Empfaenger

kernel= (fd0)/rmgr -sigma0               kernel= (fd0)/rmgr -sigma0
module= (fd0)/main -hostid=2             module= (fd0)/main -hostid=3
module= (fd0)/sigma0                     module= (fd0)/sigma0
module= (fd0)/names                      module= (fd0)/names
module= (fd0)/netipcn                    module= (fd0)/netipcn
module= (fd1)/netipc 192.168.1.1         module= (fd1)/netipc 192.168.1.1
module= (fd1)/sender 8 0 3 A B C X Y     module= (fd1)/receiver
```

Die Bedeutung der Parameter X und Y stimmt mit der aus der vorhergehenden Messung überein. Die Parameter A (Anzahl der Datenworte), B (Anzahl der Strings) und C (Länge der Strings) bestimmen die übertragene Datenmenge. Die gemessenen Werte sind in Tabelle 4 festgehalten.

Die nächste Messung wird nur auf einer virtuellen Maschine der zweiten Messanordnung ausgeführt. Sender und Empfänger benutzen direkte IPCs, um zu kommunizieren. Die Bootparameter sind die folgenden:

```
kernel= (fd0)/rmgr -sigma0
module= (fd0)/main -hostid=2
module= (fd0)/sigma0
module= (fd0)/names
module= (fd0)/netipcn
```

Anzahl der Worte (A)	5				30			
Strings (B,C)	5x2047 Byte				5x9210 Byte			
Datenmenge	10 kByte				45 kByte			
Sendethreads (X)	1	5	10	15	1	5	10	15
Durchläufe (Y)	100	100	100	50	80	20	10	10
Messwert	292 s	263 s	543 s	399 s	748 s	239 s	226 s	340 s
Übertragungsrate aller Threads in KByte/s	3,4	19,0	18,4	18,8	4,8	18,8	19,9	19,9

Tabelle 4: *Übertragungsraten für einen Call-Aufruf über Rechnergrenzen.*

```
module= (fd1)/netipc 192.168.1.1
module= (fd1)/sender 9 0 2 A B C X Y
module= (fd1)/receiver
```

Die Parameter A, B, C, X und Y bedeuten dasselbe wie bei der vorigen Messung. Die Ergebnisse sind in Tabelle 5 vermerkt.

Anzahl der Worte (A)	2		5		30	
Strings (B,C)	0		5x2047 Byte		5x9210 Byte	
Datenmenge	8 Byte		10 kByte		45 kByte	
Sendethreads (X)	1	10	1	10	1	10
Durchläufe (Y)	500.000	50.000	100.000	10.000	100.000	10.000
Messwert	348 s	344 s	150 s	186 s	350 s	389 s
Latenz	0,7 ms	6,9 ms	1,5 ms	18,6 ms	3,5 ms	38,9 ms
Übertragungsrate aller Threads in KByte/s	11,2	11,4	6667	5376	12857	11568

Tabelle 5: *Latenzzeiten und Übertragungsraten für einen direkten Call-Aufruf.*

Für die letzte Messung wird die dritte Anordnung benutzt. Hier wird ein modifizierter Routerprozess verwendet, der Nachrichten des Senders entgegennimmt, diese in Nachrichtenpuffer einpackt, dann jedoch Absender und Empfänger vertauscht und anschliessend in die Nachrichtenwarteschlange eingestellt, in der Nachrichten aus dem Netzwerk verwaltet werden. Es wird also simuliert, dass der Empfänger bereits geantwortet hat, ohne dass Daten über das Netzwerk übertragen werden. Die Daten werden vom IPC-Thread und von der Pufferverwaltung bearbeitet. Die Auswirkung dieser zwei Komponenten auf die Latenz der Nachricht werden also gemessen. Die virtuelle Maschine wird mit folgenden Parametern gestartet:

```
kernel= (fd0)/rmgr -sigma0
module= (fd0)/main -hostid=2
module= (fd0)/sigma0
```

```
module= (fd0)/names
module= (fd0)/netipcn
module= (fd1)/netipc2 192.168.1.1
module= (fd1)/sender 8 0 3 2 0 5 X Y
```

Die Parameter X und Y haben wieder die gleiche Bedeutung wie in der ersten Messung. Die Messergebnisse sind in Tabelle 6 festgehalten.

Sendethreads (X)	1	10
Durchläufe (Y)	100.000	10.000
Messwert	471 s	442 s
Latenz	4,7 ms	44,2 ms

Tabelle 6: *Latenzzeiten für einen Call-Aufruf mit modifiziertem Routerprozess.*

5.3 Bewertung

Die Messungen der Übertragungszeiten über das Netzwerk sind in Abbildung 14 dargestellt und zeigen, dass sich die Übertragung sehr verlangsamt, wenn wenige Daten übertragen werden.

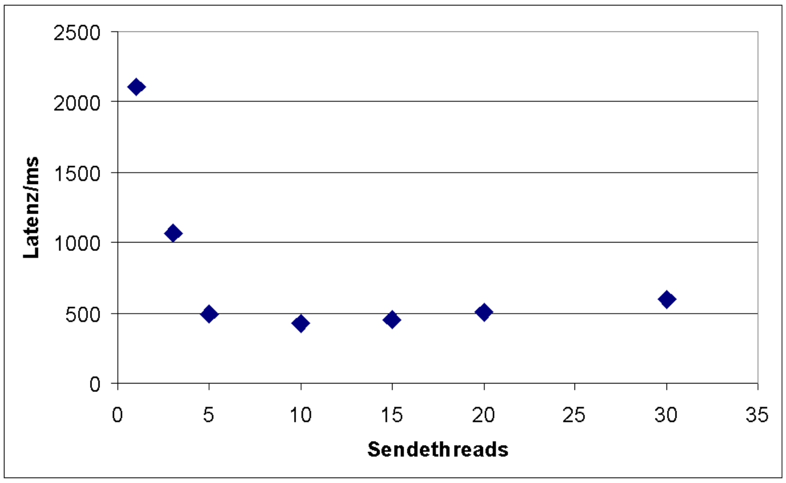

Abbildung 14: *Latenzzeiten für einen Call-Aufruf über Rechnergrenzen.*

Besonders bei der Latenzzeitmessung mit einem Sendethread konnten Stockungen der Übertragung von bis zu 20 Sekunden beobachtet werden. Bei einer Messung blieb der virtuelle Rechner auch komplett stehen und musste neu gebootet werden. Der Autor vermutet, dass die Ursache dieser Beobachtung verlorengegangene Interrupts sind. Dadurch werden empfangene Pakete aus dem Netzwerk nicht an den Netzwerktreiber gemeldet. Das verwendete TCP/IP-Protokoll sorgt nach einiger Zeit dafür, dass nichtbestätigte Netzwerkpakete erneut gesendet werden. Werden mehr Daten übertragen, entweder durch mehrere Sendethreads oder durch grössere Datenmengen, die in mehrere Netzwerkpakete aufgespalten werden, treten die Stockungen seltener auf. Hierbei lösen nachfolgende Pakete weitere Interrupts aus, und der Treiber liest nun sowohl das erste Netzwerkpaket, dessen Benachrichtigung verloren ging, als auch die folgenden.

Abgesehen davon erkennt man aus der Messung der Latenzzeiten, dass diese bei einer Netzwerkkommunikation nur langsam mit der Zahl der Sendethreads steigt. Die Übertragung der Daten über das Netz beansprucht so viel mehr Zeit als das Verarbeiten der Daten beim Empfänger. Die Kapazität des Netzwerkes ist mit den wenigen kleinen Paketen jedoch überhaupt nicht ausgelastet, wodurch sich die Übertragungszeit kaum verändert. Solange Prozesse also mit kleinen Nachrichten kommunizieren, spielt die Anzahl der Prozesse eine untergeordnete Rolle auf die Dauer einer Nachrichtenübertragung.

Anders sieht es bei der Übertragung grosser Datenmengen aus. Hier wurde die Übertragungskapazität des Netzwerkes auf die Threads aufgeteilt. Je mehr Threads Daten sendeten, desto langsamer erfolgte dieses. In Abbildung 15 wurden die Punkte für die Messung mit einem Sendethread nicht aufgenommen. Der Empfänger antwortet mit einer einzelnen kurzen Nachricht. Diese wird wie in der vorigen Messung vom Netzwerktreiber häufig verzögert und verfälscht dadurch die Messung.

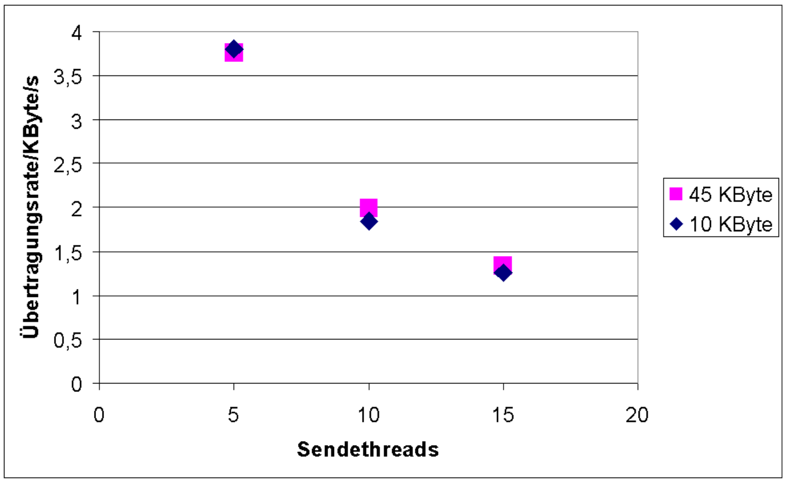

Abbildung 15: *Übertragungsraten für einen Call-Aufruf über Rechnergrenzen.*

Die anderen Messpunkte zeigen eine umgekehrt reziproke Abhängigkeit zwischen der Anzahl der Sendethreads und der Übertragungsrate pro Thread. Die maximal über das Netzwerk erreichte Datenmenge ist etwa 19 KByte/s.

Der Vergleich zwischen den gemessenen Latenzzeiten und Übertragungsraten im lokalen und im entfernten Funktionsaufruf unterscheiden sich um mehrere Grössenordnungen. Dies wurde auch erwartet. Im lokalen Fall kann der Kern die Daten vom Adressraum des Senders in den des Empfängers kopieren. Im entfernten Fall müssen die Daten schon auf dem Senderechner mehrfach kopiert und bearbeitet werden. Erst kopiert der Kern die Daten vom Adressraum des Senders in den des Routertasks. Der Router kopiert sie in die Messagepuffer und anschliessend mit einer weiteren IPC-Operation in den Adressraum des Netzwerktreibers. Dieser sendet die Daten über ein Netzwerk, dessen Übertragungsrate sehr viel langsamer ist als die des Hauptspeichers. Auf Empfängerseite wird die Nachricht wieder mehrfach kopiert, genau wie auf dem Senderechner.

Einige dieser Kopiervorgänge könnte man durch Designänderungen verhindern, indem man Puffer zwischen Adressräumen teilt. Die Übertragung wird jedoch immer vom langsamen Netzwerk gebremst.

Die letzte Messung eliminiert den Einfluss des Netzwerkes. Nachrichten werden direkt vom Routerprozess beantwortet. Die gemessenen Werte von 4,7 Millisekunden für einen Sendethread sind etwa das Siebenfache der Zeit der Kommunikation zwischen Sender und Empfänger. Der Mehraufwand durch das Abfangen der IPC durch den Kern, das Empfangen der Nachricht im IPC-Thread und die Pufferverwaltung kosten diese extra Zeit.

Trotz der gemessenen extrem langen Kommunikationszeiten kann die Implementierung praktisch eingesetzt werden, und zwar dort, wo relativ selten kleine Datenmengen ausgetauscht werden müssen, die Anwendung aber von einer Erhöhung der Prozessorzahl profitiert.

6 Ausblick

Die in dieser Arbeit beschriebene Implementierung von netzwerktransparenten Short- und String-IPCs ist der erste Schritt, den Mikrokern Fiasco als Betriebssystem auf Multicomputersystemen einzusetzen. Prozesse können mittels dem in dieser Arbeit entwickelten Routerprozesses IPCs auch an Prozesse auf anderen Rechnern schicken. Die Geschwindigkeit der Implementierung ist noch nicht für einen produktiven Einsatz geeignet.

Die weitere Arbeit an diesem Projekt sollte folgende Schwerpunkte abdecken:

Einsatz von Oshkosh Der in der Arbeit verwendete und aus dem OSKit herausgelöste Netzwerktreiber genügt zur Demonstration und Überprüfung der Durchführbarkeit der netzwerktransparenten IPCs für Fisaco. Für den richtigen Einsatz sollte jedoch der Treiber mit TCP/IP-Stack des Paketes Oshkosh verwendet werden, der bessere Performance und eine breitere Hardwareunterstützung bietet. Die Portierung dieser Arbeit auf Oshkosh wurde bereits vorbereitet, indem bereits die zukünftige Programmierschnittstelle des TCP/IP-Stacks von Oshkosh verwendet wurde.

FlexpageIPC Zur Zeit können nur Short- und StringIPCs vom Routerprozess verarbeitet werden. FlexpageIPCs werden zurückgewiesen. Sollen auch diese verarbeitet werden, muss eine verteilte Speicherverwaltung genutzt werden, um Speicherseiten auch über Rechnergrenzen mappen zu können. Damit kann dann der volle Funktionsumfang des IPC-Systemaufrufes genutzt werden.

Transparente Monitore Die derzeitige Implementierung netzwerktransparenter IPC ändert die Semantik der Nachrichtenübertragung. Durch den Einsatz des Konzeptes der transparenten Monitore kann die Semantik weitestgehend wiederhergestellt werden, da die Kommunikation wieder synchron abläuft.

Migration In dieser Arbeit wurde mehrfach darauf eingegangen, dass das Endziel ein System aus mehreren Rechnern sein könnte, welches den darauf laufenden Prozessen jedoch wie ein einziges grossen System erscheint. Prozesse werden dann lastabhängig von einem Rechner zu einem anderen migriert. Dazu muss ein Checkpointingsystem für Fiasco-Prozesse geschaffen werden und die Behandlung der eingeführten HostIDs erweitert werden, so dass auch Prozesse anderer Rechner als dem lokalen ausgeführt werden können.

7 Zusammenfassung

Mit dieser Arbeit wurden Konzepte zur Implementierung von netzwerktransparenten Interprozesskommunikationen (IPC) auf dem Mikrokern Fiasco der Betriebssystemgruppe der Fakultät Informatik entwickelt. Die Konzepte unterscheiden sich in der erreichbaren Semantikkonformität mit der direkten IPC. Ein Konzept, welches nur eine minimale Erweiterung des Fiasko-Kerns bedurfte, wurde implementiert und getestet.

Es konnte gezeigt werden, dass bei dieser Implementierung die Semantik eines in der Praxis häufig vorkommenden Call-Aufrufes mit endlichem Timeout, im direkten und entfernten Aufruf ähnlich ist, wenn man eine Neuinterpretation der Fehlercodes vornimmt. Die Semantik einer normalen IPC-Operation entspricht der Semantik einer Kommunikation über Clangrenzen, mit der Ausnahme, dass die Behandlung einer unzustellbaren Nachricht verändert ist.

Programme können beim Einsatz der entwickelten Lösung den in L4/Fiasco vorhandenen Systemaufruf zum Senden einer Nachricht an ein anderes Programm nun auch nutzen, um Daten an ein Programm auf einem anderen Rechner zu schicken. Sämtliche Parameter des Aufrufes unterscheiden sich nicht von einem Aufruf im Falle einer lokalen Kommunikation, einzig die Adresse des Empfängers verweist auf einen Prozess auf einem anderen Rechner. Die Implementierung weist jedoch Nachrichtensendungen zurück, die Speicherseiten in den Adressraum des Empfängers einblenden sollen.

Der Test offenbarte Mängel der Performance und Stabilität des verwendeten Netzwerktreibers. Es wurden jedoch bereits Vorbereitungen getroffen, den verbesserten Netzwerktreiber Oshkosh einzubinden, sobald dieser einen TCP/IP-Stack bietet.

Desweiteren wurde beim Design und bei der Implementierung an mehreren Stellen darauf eingegangen, wo die weitere Arbeit an diesem Projekt ansetzten kann, um netzwerktransparente IPCs weiterzuentwickeln. Beispielsweise könnte eine verteilte Speicherverwaltung und die Unterstützung von Migration das System zu einem einsatzfähigen Clusterbetriebssystem ausbauen.

Literatur

[1] L4-Hacker Mailingliste. http://os.inf.tu-dresden.de/pipermail/l4-hackers.mbox/l4-hackers.mbox.

[2] Mikrokernbasierte sichere Inter-Netzwerk-Architektur (μsina). http://os.inf.tu-dresden.de/mikrosina/.

[3] Alan Au and Gernot Heiser. L4 user manual.

[4] Joseph S. Barrera III. A fast mach network IPC implementation. In *Proceedings of the Second Mach USENIX Symposium*, Monterey, CA (USA), 1991.

[5] Uwe Dannowsky. An ATM driver for DROPS. Grosser Beleg, Juli 1998.

[6] Bryan Ford, Kevin Van Maren, Jay Lepreau, Stephen Clawson, Bart Robinson, and Jeff Turner. The flux OS toolkit: Reusable components for OS implementation. In *Workshop on Hot Topics in Operating Systems*, pages 14–19, 1997.

[7] The MPI Forum. Mpi: A message passing interfase.

[8] Michael Hohmuth. Using the oskit as a base for L4 applications.

[9] VMware Inc. Vmware - virtual mashine software. http://vmware.com.

[10] Trent Jaeger, Jonathon E. Tidswell, Alain Gefflaut, Yoonho Park, Jochen Liedtke, and Kevin Elphinstone. Synchronous ipc over transparent monitors.

[11] IBM Watson Technical Report Jochen Liedtke. Lava nucleus (ln) reference manual [x86] version 2.2, 1998.

[12] Lars Reuther Jork Löser, Hermann Härtig. A streaming interface for real-time interprocess communication, 2001.

[13] Linus Peter Kamb. Extending fluke ipc for transparent remote communication.

[14] Jochen Liedtke. Clans & chiefs. In *12. GI/ITG-Fachtagung Architektur von Rechensystemen*, pages 294–305, Kiel, 1992.

[15] Jochen Liedtke. L4 version X in a nutshell, 1999.

[16] J. Wolter H. Härtig M. Borriss, M. Hohmuth. Portierung von Linux auf den μ-Kern L4, 1997.

[17] Gordon E. Moore. Cramming more components onto integrated circuits, 1965.

[18] Sven Reigl. Erstellung eines Myrinettreibers für DROPS. Grosser Beleg, Oktober 2000.

[19] Sven Reigl. Ein echtzeitfähiger IP-Stack für die Myrinet-Architektur. Diplom, August 2001.

[20] Andrew S. Tanenbaum. Moderne Betriebssysteme, 2002.

[21] Andrew S. Tanenbaum and Et. Al. The amoeba distributed operating system, 1990.