TECHNISCHE UNIVERSITÄT DRESDEN
Fakultät Informatik
Institut für Systemarchitektur

Großer Beleg

Analyse der Kommunikation mit dynamisch verteilten Anwendungen auf L4

Kai Zickmann
Matrikelnummer 2763652

Betreuer: Dipl.-Inf. Ronald Aigner
Dr.-Ing. Michael Hohmuth
Verantwortlicher Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig

3. März 2005

Aufgabenstellung

Analyse der Kommunikation mit dynamisch verteilten Anwendungen auf L4

Die Aufgabe der Arbeit ist es, einige bekannte Ansätze zur Kommunikation von verteilten Anwendungen zu analysieren. Dynamisch verteilte Anwendungen sind so konstruiert, dass sie über die Verteilung Bescheid wissen und diese ausnutzen können. Für Klienten dieser Anwendungen soll es nicht ersichtlich sein, ob diese Anwendungen auf dem gleichen Rechner oder einem entfernten Rechner laufen. Um dies zu realisieren, können Proxys eingesetzt werden, welche entweder spezifisch sind und von der verteilten Anwendung bereit gestellt werden oder generisch sind. Dazu soll untersucht werden, wie groß der Aufwand der Konstruktion generischer Proxys ist und wie Timeout-Semantiken für verteilte Anwendungen beschränkt werden müssen oder mit Hilfe der Proxys realisiert werden können.

Schwerpunkte der Arbeit sind:

- Entwurf und Implementierung eines generischen Proxy
- Analyse dieses Proxys im Hinblick auf Timeout Semantiken von L4
- Entwicklung eines Beispielszenarios zur Demonstration

Inhaltsverzeichnis

1 Einleitung

Seitdem Computer existieren, empfindet sie der Nutzer als zu langsam. Ständig wird die Rechenleistung der Prozessoren erhöht. In den letzten 40 Jahren hat sich die Rechenleistung etwa jedes Jahr verdoppelt. Heutzutage können wir Prozessoren produzieren, die jede Sekunde Milliarden von Operationen ausführen können. Aber es gibt immer noch Anwendungsbereiche (z.B. Crash-Tests, Klimaforschung, ...), in denen die heute verfügbare Rechenleistung eines Prozessors bei weitem nicht ausreicht. Dazu kommt noch die Frage, ob sich die Rechenleistung auch in der Zukunft jährlich verdoppeln kann.[1] Aus diesem Grunde nutzt man für rechenintensive Probleme sogenannte Supercomputer, welche teilweise aus mehreren Tausend Prozessoren bestehen. Diese Rechner besitzen eine enorm hohe Rechenleistung, sind allerdings auch sehr teuer. Eine wesentlich preiswertere Variante ist es, eine Vielzahl von „einfachen“ Arbeitsplatzrechnern über ein Netzwerk zu verbinden und somit eine höhere Rechenleistung zu erhalten. So kann ein ganzer Rechnerverbund ein Problem lösen.

Eine weitere Möglichkeit, mehrere über ein Netzwerk verbundene Rechner einzusetzen, ist die sogenannte Prozessmigration. Angenommen, man befindet sich in einem Unternehmen mit einer Vielzahl von Mitarbeitern, die alle an einem eigenen Computer arbeiten. Dann wird die Rechenkapazität der Computer meist nicht vollständig ausgenutzt (z.B. bei Textverarbeitung, Tabellenkalkulation, im Internet surfen, ...). Wenn nun einige Mitarbeiter ab und zu sehr rechenintensive Programme ausführen müssen, wäre es sinnvoll, auch die Rechenkapazität der sich im Leerlauf befindlichen Computer zu nutzen. Dazu werden rechenintensive Prozesse auf andere Rechner verschoben (Prozessmigration). Das bedeutet, wenn ein Computer sehr viel Arbeit hat, verteilt er die Arbeit an andere Computer, die nur wenig Arbeit haben.

In der letzten Zeit gewinnen sichere Systeme immer mehr an Bedeutung. Es häufen sich Angriffe auf Computersysteme mit dem Ziel, deren Lauffähigkeit zu gefährden (z.B. Viren) oder Informationen in Erfahrung zu bringen (z.B. Spyware). Ein sicheres Betriebssystem sei hier ein Betriebssystem, welches alle Prozesse voreinander schützt. Das bedeutet, ein (bösartiger) Prozess darf keine Möglichkeit haben, die Ausführung anderer Prozesse zu beeinflussen. Die einzige Möglichkeit der Interaktion von zwei Prozessen ist die Interprozesskommunikation (IPC) bzw. der Aufbau von gemeinsamem Speicher und anschließende Informationsübertragung über diesen Speicher. Wenn man nun ein derartiges sicheres Betriebssystem besitzt, so muss ein angreifender Prozess nicht nur gestartet werden, sondern auch versuchen, mittels IPC andere Prozesse so zu manipulieren, dass sie ihm Informationen preisgeben bzw. dass das System (oder Teile von ihm) instabil werden.

Alle drei beschriebenen Probleme haben eine Sache gemeinsam: man kann sie lösen, indem man Kommunikation (z.B. mittels Proxys) überwacht. Stellt der Proxy fest, dass Prozesse miteinander kommunizieren möchten, die sich auf unterschiedlichen Rechnern befinden, so kann der Proxy die Kommunikation über das Netzwerk umleiten, ohne dass die beteiligten Prozesse dies wissen müssen. Oder es existiert eine systemweite Sicherheitsrichtlinie, die festlegt, welche Prozesse miteinander kommunizieren dürfen. Dann kann ein Proxy die Kommunikation bestimmter (oder aller) Prozesse überwachen und ggf. blocken.

[1] Die Informationsübertragung innerhalb des Prozessors kann nicht schneller als mit Lichtgeschwindigkeit erfolgen. Das bedeutet, ein moderner 3 GHz-Prozessor kann Informationen innerhalb eines Taktes maximal 10 cm weit transportieren. (Dabei sind die Schalt- und Verarbeitungszeiten schon ignoriert!) Das wiederum bedeutet: 10 cm ist der maximale Chip-Durchmesser. Wenn man nun die Rechenleistung jährlich verdoppelt, so besitzen wir in 10 Jahren einen 3 PHz (P=Peta=1024G)-Prozessor. Dieser Prozessor hätte einen maximalen Chip-Durchmesser von 0,1 mm. Wenn man bedenkt, dass der komplette Stromverbrauch eines Prozessors in Wärme umgewandelt wird und diese Wärme von einem nicht einmal 0,001mm^3 großem Chip abgeführt werden muss, so würde sich dieser Prozessor sicher sehr schnell überhitzen.

Im Kapitel 2 werde ich auf Grundlagen für diese Arbeit – vor allem die Interprozesskommunikation – eingehen. Anschließend betrachte ich verwandte Arbeiten zum Thema Prozessmigration und netzwerkübergreifender Kommunikation.

Im 3. Kapitel werde ich verschiedene Ansätze, Kommunikation zu überwachen, betrachten. Anschließend werde ich mich auf einen Ansatz festlegen und diesen vertiefen. Auch die dadurch bedingten Änderungen am Nameserver werde ich erörtern.

Im Kapitel 4 dokumentiere ich schließlich die Implementation meines Ansatzes und messe in Kapitel 5 das grundlegende Laufzeitverhalten meiner implementierten Proxys aus.

Die von mir implementierten Proxys ermöglichen eine Kommunikationsüberwachung. Mit Hilfe dieser Kommunikationsüberwachung kann man sehr viele Dienste erbringen. Im Kapitel 7 sind einige Ideen dazu dargelegt.

2 Grundlagen und Stand der Technik

2.1 Begriffe

Mikrokern-Betriebssysteme
Mikrokern-Betriebsysteme sind, wie der Name schon sagt, sehr klein. Man versucht so viel Funktionalität wie möglich aus dem Betriebssystem-Kern in Nutzerapplikationen zu verlagern. Das hat den Vorteil, dass diese ausgelagerte Funktionalität nicht mehr im privilegiertem Modus (des Prozessors) abläuft und somit kein potentielles Sicherheitsrisiko darstellt. Außerdem wird der Quellcode des Betriebsystems reduziert, so dass er überschaubarer und leichter verifizierbar wird. Des Weiteren werden die Eigenschaften des Betriebsystems vorhersagbarer.

Fiasco
L4 ist eine Mikrokernfamilie der zweiten Generation und wurde von Jochen Liedke entwickelt. An der TU Dresden wurde Fiasco, ein L4-kompatibler Mikrokern, implementiert und dient als Ausgangsbasis für weitere Forschungsprojekte (Sicherheit, Echtzeitfähigkeit), und auch als Basis für meine Arbeit.

L4-Task
Eine L4-Task (=Prozess) ist eine Art „Ressourcencontainer“. Die wichtigsten Ressourcen einer L4-Task sind der Adressraum und eine begrenzte Menge an Threads. Alle Threads haben gemeinsamen Zugriff auf alle Ressourcen des Prozesses. Der Adressraum besteht aus einer Menge von virtuellen Speicherseiten, die auf physischen Speicher abgebildet werden. Dabei können auch verschiedene virtuelle Adressen (u.a. auch von verschiedenen Prozessen) auf die gleiche physische Adresse abgebildet werden. In diesem Falle spricht man von gemeinsamen Speicher.

Threads
Ein Thread ist ein mit einer Handlungsvorschrift (Programm) verknüpfter Aktivitätsträger, die Abstraktion des Prozessors. Jeder Thread gehört genau zu einer Task und hat somit Zugriff auf dessen Ressourcen. Während der Prozess nur die Ressourcen verwaltet (und die Threads als Ressource besitzt), führt der Thread die eigentliche Arbeit aus und verwendet dazu den Prozessor und die Betriebssystemfunktionalität.

Proxy
Ein Proxy ist ein Stellvertreter. Wenn man die Kommunikation zwischen zwei Bereichen (z.B. lokales Netzwerk und Internet) überwachen möchte, so verbietet man direkte Kommunikation zwischen den beiden Bereichen. Wenn Kommunikation stattfinden soll, dann geschieht dies nur über den Proxy, welcher die Kommunikation überprüfen kann und ggf. weiterleitet.

Man kann Proxys aber nicht nur zur Kommunikationsüberwachung verwenden, sondern damit auch Transparenz erreichen. Wenn aus dem Internet nicht ersichtlich werden soll, wie ein lokales Netzwerk aufgebaut ist, aber Dienste für das Internet bereitgestellt werden (z.B. Informationen auf Homepages), so kann dies ebenfalls über ein Proxy realisiert werden. In dem Fall würde der Proxy die Anfragen aus dem Internet entgegen nehmen und die gleiche Anfrage selbst an das lokale Netzwerk stellen. Wenn er dann die Antwort aus dem lokalen Netzwerk erhält, würde der Proxy diese einfach kopieren und als Antwort ins Internet weiterleiten. Dadurch erhält der Internet-Client den Eindruck, als ob der Proxy auf alle Anfragen antwortet, obwohl dies tatsächlich verschiedene Computer des lokalen Netzwerks tun.

Generischer Proxy
Proxys werden in anwendungsspezifische und generische Proxys unterteilt. Anwendungsspezifische Proxys haben Wissen von der Anwendung, mit der sie zusammenarbeiten. Dadurch können solche Proxys wesentlich effektiver arbeiten, denn die Anwendung versendet und empfängt üblicherweise nicht eine beliebige theoretisch mögliche Anfrage, sondern nur Anfragen einer gewissen Teilmenge. Allerdings erfordert eine Änderung der Anwendung (oder gar der Austausch einer Anwendung) auch eine Änderung des Proxys.

Generische Proxys haben kein Wissen von der Anwendung. Dadurch sind sie nicht ganz so effektiv, aber man benötigt nur ein generischen Proxy, welcher mit allen Anwendungen zusammenarbeitet.

2.2 IPC in L4/Fiasco

L4 bietet unter anderem die Betriebssystemfunktionalität „Interprozesskommunikation“ (IPC). Dabei wird eine Nachricht von einem Sender zu einem Empfänger übertragen. Der Sender gibt bei dieser Operation die ID des Empfängers, die Nachricht und einen Timeout an. Der Empfänger gibt an, von welchem Thread er Nachrichten empfangen will, stellt Speicherplatz für die Nachricht zur Verfügung und gibt ebenfalls einen Timeout an. Wenn sowohl Sender als auch Empfänger bereit sind, wird die IPC durchgeführt, das heißt, die Nachricht wird übertragen. Bei der Rückkehr des Betriebssystemaufrufes erhält der Sender einen Fehlercode. Der Empfänger erhält auch einen Fehlercode, zusätzlich noch die ID des Senders und ggf. Zusatzinformationen über die Nachricht.

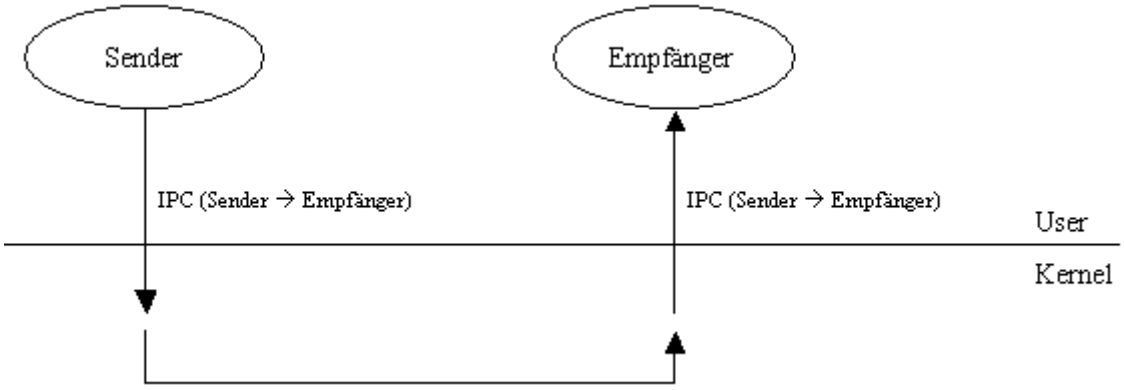


Abbildung 1, IPC in L4

In L4 existieren 3 IPC-Arten: Short-IPC, Long-IPC und Flexpage-IPC. Bei der Short-IPC besteht die Nachricht aus zwei Datenwörtern, die 32 Bit groß sind. Diese Form des IPC ist die schnellste, da die Nachricht nicht im eigentlichen Sinne kopiert wird, sondern die beiden Datenwörter werden in zwei Registern des Prozessors gespeichert. Danach gibt es einen Kontextwechsel zum Empfänger, das bedeutet, jetzt besitzt der Empfänger den Prozessor. Somit kann er die Datenwörter aus den Prozessor-Registern auslesen und hat die Nachricht damit empfangen.

Bei der Long-IPC besteht die Nachricht aus bis zu 2^{19} Datenwörtern und 2^5 Zeigern auf Speicherbereiche, die jeweils bis zu 4 MB groß sein können. Hier muss die Nachricht (Datenwörter, Zeiger, Speicherbereich) physisch kopiert werden. Dadurch dauert eine Long-IPC wesentlich länger als eine Short-IPC.

Bei der Flexpage-IPC können zusätzlich zu den Datenwörtern und Speicherbereichen auch mehrere Speicherseiten kopiert werden. Dabei werden die Speicherseiten nicht wirklich kopiert, sondern deren Inhalt (er befindet sich auf Hauptspeicher-Kacheln) wird im Adressraum des Empfängers eingeblendet. Damit kann man große Adressraum-Teile effektiv kopieren oder verschieben, gemeinsamen Speicher zwischen verschiedenen L4-Tasks etablieren und sogar Seitenfehler behandeln.

In Fiasco ist – wie in anderen Mikrokernbetriebssystemen auch – die sogenannte Client-Server-Kommunikation üblich. Da in Mikrokernbetriebssystemen soviel Funktionalität wie möglich aus dem Betriebssystem-Kern ausgelagert wird, bieten Server diese Funktionalität an. Clients nutzen diese Funktionalität nun nicht als Kernfunktionalität (wie z.B. in monolithischen Betriebssystemen), sondern als Server-Funktionalität.

Die Client-Server-Kommunikation setzt voraus, dass der Client Kenntnis von der ID des Servers hat. Dazu existiert ein Nameserver, bei dem sich jeder Server unmittelbar nach dem Start mit seinem Namen und seiner ID registriert. Wenn ein Client nun die Serverfunktionalität nutzen möchte, so muss er einmalig eine Anfrage (mit dem Server-Namen) an den Nameserver stellen, der dem Client die passende Server-ID zurückliefert.

Bei jeder Client-Server-Kommunikation sendet der Client nun seine Anfrage an die Server-ID. Der Server wartet auf Anfragen von allen Threads. Er empfängt also die Anfrage und erhält damit auch die ID des Clients. Jetzt kann der Server die Anfrage bearbeiten und die Antwort an den Client zurückschicken. Der Client erhält so die Antwort, damit ist die Client-Server-Kommunikation beendet.

Wichtig ist hier: der Server empfängt Anfragen von allen Threads und schickt die Antwort an den Thread, von dem er die Anfrage erhielt. Der Client benötigt die ID des Servers und sendet die Anfrage diesem. Anschließend empfängt der Client die Antwort ausschließlich von dem Server.

In L4 erfolgt die Übertragung der IPC synchron und ungepuffert. Das bedeutet, wenn ein Sender einem Empfänger eine Nachricht schickt, so wartet (blockiert) der Sender, bis der Empfänger empfangsbereit ist. Falls der Empfänger eher empfangsbereit ist, so blockiert dieser, bis der Sender sendebereit ist. Erst wenn sowohl Sender als auch Empfänger kommunikationsbereit sind, wird die IPC durchgeführt und die Nachricht übertragen.

Die Eigenschaften „synchron“ und „ungepuffert“ werden im weiteren Beleg auch als Semantikeigenschaften einer IPC in L4 bezeichnet.

2.3 Kernänderungen

Für Mikrokern-Betriebssysteme gilt das Minimalitätsprinzip: Im Betriebssystem-Kern sollte nur die Funktionalität implementiert sein, die nicht anders realisiert werden kann. Dazu gehören beispielsweise Kommunikationsprimitiven (IPC) und der Adressraum-Schutz. Jede andere Funktionalität, die auch im nicht-privilegierten Modus erbracht werden kann, sollte nicht im Betriebssystem-Kern implementiert werden.

Aus diesem Grunde soll in dieser Arbeit eine Kernänderung weitestgehend vermieden werden. Außerdem wird geprüft, ob eine Kommunikationsüberwachung auch ganz ohne Kernänderung möglich ist.

2.4 Verwandte Arbeiten

Mosix ist das bekannteste Betriebssystem, welches Prozessmigration unterstützt. Es ist eine Variante von Linux und wird hauptsächlich zur Lastverteilung benutzt.

Migiriert ein Prozess unter Mosix, so wird er nicht komplett auf den Zielrechner verschoben, sondern es verbleibt immer ein Prozessrumpf auf dem ursprünglichen Rechner (auf dem dieser Prozess gestartet wurde). Die Systemaufrufe werden unterteilt in ortsunabhängige und ortsabhängige Aufrufe. Die ortsunabhängigen Systemaufrufe können direkt von dem migrierten Prozess ausgeführt werden, egal auf welchem Rechner dieser sich befindet. Führt der migrierte Prozess ortsabhängige Systemaufrufe aus, so wird eine Nachricht an den Prozessrumpf gesendet. Dieser führt nun den Systemaufruf aus und übermittelt das Ergebnis an den migrierten Prozess.

Jan Glauber hat in seiner Diplomarbeit „Checkpointing als Basis für transparente Service-Migration unter Linux“ [7] Linux-Prozesse migriert, indem er das sogenannte Checkpointing verwendet hat. Im Gegensatz zu Mosix migriert er in seiner Arbeit komplette Prozesse, es bleiben also keine Prozessrümpfe auf dem ursprünglichen Rechner zurück.

Unter Checkpointing versteht man, eine Sicherheitskopie (=Checkpoint) eines laufenden Prozesses zu erstellen. Wenn man einen sehr langlebigen Prozess besitzt, so kann es sinnvoll sein, regelmäßig diese Checkpoints zu erzeugen. Falls ein Fehler auftritt (z.B. Systemabsturz), braucht man nicht den Prozess neu zu starten, sondern nur den letzten Checkpoint laden.

Jan Glauber hat in seiner Diplomarbeit nun Prozesse migriert, indem er mit der Checkpointing-Implementierung „Crak“ einen Checkpoint des zu migrierenden Prozesses erzeugt hat. Nun hat er den Prozess beendet, den Checkpoint auf den Zielrechner übertragen und ihn dort geladen.

Sebastian Lehmann hat in seiner Diplomarbeit „Netzwerktransparente IPC für L4/Fiasco“ [4] bereits transparente IPC über Rechnergrenzen hinweg implementiert. Transparente IPC bedeutet dabei, dass Threads wie im lokalen Fall miteinander kommunizieren können. Falls sich zwei an einer IPC beteiligten Threads auf unterschiedlichen Rechnern befinden, so wird die IPC über das Netzwerk umgeleitet und dem Ziel-Thread auf dem anderen Rechner zugestellt. Über diese Umleitung der IPC wissen die beteiligten Threads nicht Bescheid – die Umleitung ist weitestgehend transparent.

In meiner Arbeit werde ich Kernänderungen vermeiden und mich darauf konzentrieren, die IPC von bestimmten Threads zu überwachen, d.h. ich werde Proxys implementieren, welche die IPC bestimmter Threads abfangen und dann an den ursprünglichen Empfänger weiterleiten. Um eine netzwerktransparente Kommunikation zu erreichen, müssen die Proxys lediglich so modifiziert werden, dass sie die erhaltene IPC ggf. über das Netzwerk umleiten. Diese Funktionalität kann direkt aus Sebastian Lehmanns Arbeit kopiert werden.

Durch das Beschränken auf die Überwachung der IPC kann meine Arbeit nicht nur für verteilte Anwendungen benutzt werden, sondern z.B. auch für Sicherheitsanwendungen. In dem Fall würde man die Proxys so modifizieren, dass sie die erhaltene IPC überprüfen und ggf. nicht weiterleiten.

3 Entwurf

3.1 Szenario Client-Server-Kommunikation

Es gibt verschiedene Kommunikationsmuster. Obwohl diese Arbeit weitestgehend transparent zu den verwendeten Kommunikationsmustern ist, werde ich mich auf die Client-Server-Kommunikation beziehen, da diese (zumindest in Mikrokernbetriebssystemen) am weitesten verbreitet ist.

Im Folgenden gehe ich davon aus, dass sich bei einer Kommunikationsbeziehung der Server beim Nameserver anmeldet und die Clients danach über den Server-Namen die Server-ID erfragen können. Der Client sendet nun seine Anfrage an die erhaltene Server-ID und wartet auf die Antwort, indem er eine Nachricht von genau dieser Server-ID empfängt. Für diese Sende- und Empfangsoperation gibt es in L4 eine Operation: ein Call.

Der Server wartet nach der Anmeldung beim Nameserver auf eintreffende Anfragen von Clients. Da seine Funktionalität jedem Thread zur Verfügung steht, d.h. jeder Thread ein potentieller Client sein könnte, empfängt der Server Nachrichten von jedem beliebigen Thread. Diese Empfangsoperation wird auch als Open-Wait bezeichnet. Nach dem Erhalt der Nachricht überprüft der Server, ob es sich um eine gültige Anfrage handelt und bestimmt ggf. das geforderte Ergebnis. Dieses Ergebnis sendet der Server dann einfach an den Thread zurück, von dem er die Anfrage erhalten hat. Um welchen Thread es sich dabei handelt, ist für den Server uninteressant. Anschließend wartet der Server auf die nächste Anfrage.

Wenn es sich nicht um eine Client-Server-Beziehung handelt, sondern zwei (oder mehr) Kommunikationspartner beliebig miteinander kommunizieren, so funktionieren die Proxys trotzdem, solange mindestens ein Kommunikationspartner beim Nameserver angemeldet ist und der andere dessen ID abfragt.

3.2 Techniken / Alternativen

Ganz allgemein ist das Ziel dieser Arbeit die Überwachung der Interprozesskommunikation (IPC) bestimmter Threads. Das bedeutet, dass die zu überwachenden Threads nicht direkt mit anderen Threads kommunizieren sollen, sondern dass jede Kommunikation von den Proxys überwacht und ggf. modifiziert oder blockiert werden soll.

Wie in [4] dargelegt, gibt es grundsätzlich mehrere Möglichkeiten, um sich in die Kommunikation zwischenzuschalten.

Redirect and Deceit

Die erste Möglichkeit wäre, die IPC im Betriebssystemkern umzuleiten. Das bedeutet, der Sender versendet eine IPC an einen Empfänger. Allerdings stellt das Betriebssystem die IPC nicht dem angegebenen Empfänger zu, sondern an einen anderen Thread, z.B. dem Proxy (Redirect). Sobald der Proxy die IPC empfangen hat, ist die Kommunikation beendet, d.h. der Sender geht davon aus, dass die IPC an den Empfänger zugestellt wurde (1). Von der Umleitung an den Proxy bekommt der Sender nichts mit.

Der Proxy kann die IPC nun bearbeiten, also den Inhalt auswerten und ggf. modifizieren oder die IPC über ein Netzwerk an ein Proxy auf einem entfernten Rechner übertragen. Schließlich kann der Proxy auch entscheiden, ob die IPC blockiert wird oder ob sie dem ur-

sprünglichen Empfänger zugestellt wird. Falls die IPC zugestellt werden soll, so wird die IPC an den Empfänger gesendet (2). Das Betriebssystem „fälscht“ bei dieser Operation den Absender, so dass der Empfänger den Eindruck erhält, er hätte die IPC direkt vom ursprünglichen Sender erhalten (Deceit).

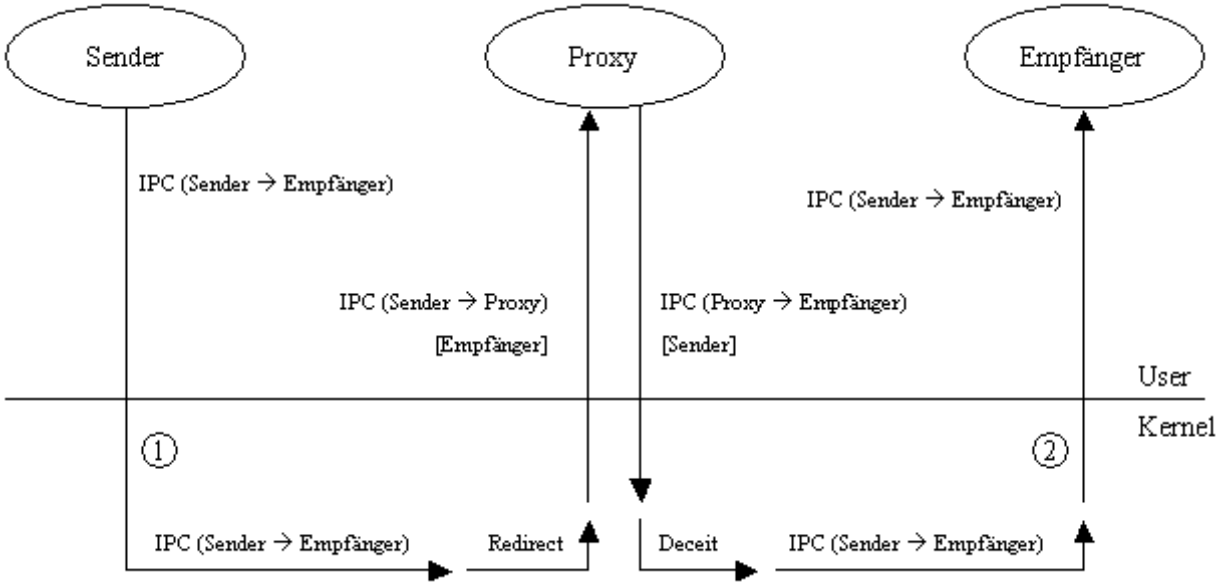


Abbildung 2, Redirect and Deceit

Der Nachteil von Redirect and Deceit ist, dass die Kommunikation nun nicht mehr synchron ist und damit die Semantik der IPC verändert wurde. Die Hauptprobleme bestehen darin, dass sich so überwachte Threads nicht mehr so einfach mittels IPC synchronisieren können und dass nicht alle auftretenden IPC-Fehler beiden Threads zugestellt werden. Wenn z.B. der Empfänger einen zu kleinen Nachrichtenpuffer bereitstellt, ergibt sich folgendes Szenario: der Sender verschickt eine IPC, welche der Proxy korrekt empfängt. Damit ist diese IPC-Beziehung beendet, und der Sender würde als Rückgabewert (vom Proxy) „OK“ erhalten. Die Sender-Logik geht also davon aus, dass der Empfänger die IPC erhalten hat. Nun leitet der Proxy die IPC an den Empfänger weiter. Dabei tritt aber ein IPC-Fehler auf, weil der Nachrichten-Puffer des Empfängers zu klein ist. Zu diesem Zeitpunkt hat der Proxy keine Möglichkeit mehr, diese Fehlermeldung an den Sender weiterzuleiten.

Transparente Monitore

Diese Veränderung der Semantik kann man mit dem Konzept der transparenten Monitore vermeiden [6]. Transparente Monitore sind Threads, die sich in den IPC Pfad eines zu überwachenden Threads einklinken. Im Gegensatz zu „Redirect and Deceit“ existiert nur eine IPC-Beziehung (1) zwischen Sender und Empfänger. Die IPC vom Sender an den Empfänger wird nicht einfach an den Monitor umgeleitet, sondern der IPC-Pfad wird erweitert. Das bedeutet, dass die IPC vom Sender zunächst dem Monitor zugestellt wird. Wenn der Monitor die IPC vom Sender erhält, ist die IPC-Beziehung zwischen Sender und Empfänger allerdings noch nicht beendet und der Betriebssystemaufruf „Send“ des Senders kehrt noch nicht zurück. Der Monitor kann nun die IPC beliebig manipulieren und danach dem Empfänger zustellen. Erst wenn der Empfänger die IPC erhält, ist die IPC-Beziehung zwischen Sender und Empfänger beendet.

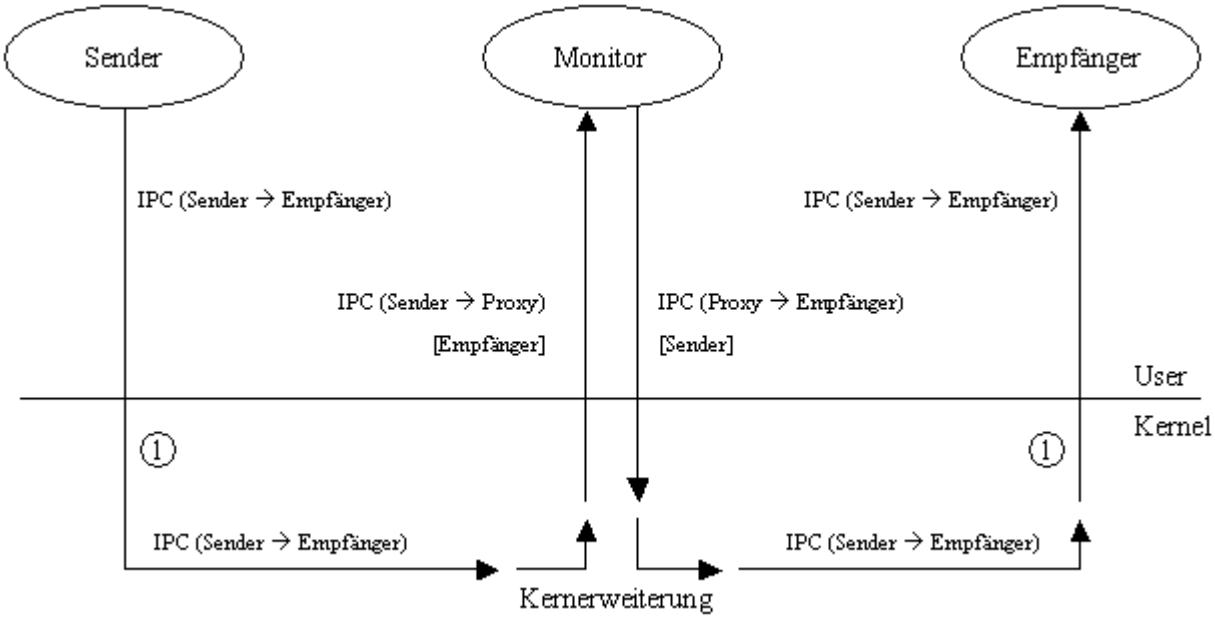


Abbildung 3, Transparente Monitore

Falls es bei der Zustellung der IPC vom Monitor an den Empfänger zu einem Fehler kommt (oder der Monitor die IPC z.B. aus Sicherheitsgründen blockiert), kann dem Sender immer noch ein Fehlercode zurückgeliefert werden, da die IPC-Beziehung zwischen Sender und Empfänger zu diesem Zeitpunkt ja noch besteht. Des Weiteren ist auch eine einfache Synchronisation zwischen Sender und Empfänger möglich: Wenn der Sender eine IPC an den Empfänger verschickt und dieser Funktionsaufruf (Betriebssystemaufruf) ohne Fehler zurückkehrt, so hat der Empfänger die IPC garantiert erhalten.

Stellvertreter-Threads

Eine weitere Möglichkeit der Überwachung stellt die Kommunikation über Stellvertreter-Threads dar. Das bedeutet, dass die zu überwachenden Threads und das restliche System nicht direkt miteinander kommunizieren, sondern ausschließlich über Stellvertreter-Threads.

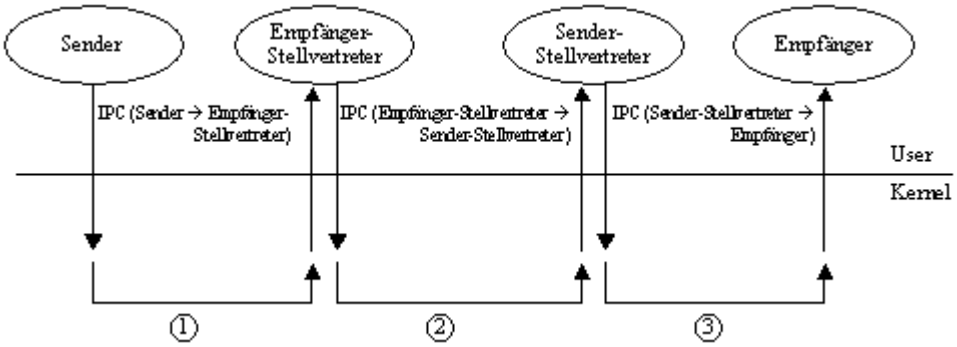


Abbildung 4, Kommunikation über Stellvertreter-Threads

Wenn bei dem Sender/Empfänger-Beispiel einer der beiden Threads überwacht werden soll, so schickt der Sender die IPC nicht direkt an den Empfänger, sondern an den Empfänger-Stellvertreter (1). Dieser Empfänger-Stellvertreter leitet die IPC an den Sender-Stellvertreter weiter (2) und der Sender-Stellvertreter schickt die IPC schließlich an den Empfänger (3).

Dabei soll die Überwachung transparent sein, d.h. der Sender soll den Empfänger-Stellvertreter für den wahren Empfänger halten und keinen Unterschied feststellen. Genau das Gleiche gilt für den Empfänger: auch er soll nicht bemerken, dass die IPC nicht vom Sender, sondern von dessen Stellvertreter kam.

Der Nachteil der Stellvertreter-Threads ist, genau wie bei Redirect and Deceit, dass die Kommunikation nun nicht mehr synchron ist und damit die Semantik der IPC verändert wurde. Ein weiteres Problem könnte die Anzahl der Stellvertreter-Threads werden. Für jeden zu überwachenden Thread und für jeden Thread, der mit einem zu überwachenden Thread kommuniziert, muss ein Stellvertreter-Thread angelegt werden.

Allerdings hat die Kommunikation über Stellvertreter-Threads den großen Vorteil, dass keine Kernänderung notwendig ist.

Wertung

Man kann mittels Kernänderung eine synchrone, netzwerktransparente IPC implementieren oder man kann Kernänderungen vermeiden. Beim Vermeiden von Kernänderungen kann man aber lediglich eine asynchrone, netzwerktransparente IPC erhalten.

Die Kommunikationsüberwachung mittels der Stellvertreter-Threads ist die einzige Möglichkeit, Kommunikation zwischen zwei Threads ohne Kernänderung zu überwachen. Da die IPC durch den Betriebssystemkern per Definition vom Sender zum Empfänger übertragen wird, gibt es nur die beiden Möglichkeiten: Entweder ändert man den Betriebssystemkern und informiert den Proxy über die IPC, oder der Sender spezifiziert von sich aus den Proxy (hier den Stellvertreter-Thread) als Empfänger. Dann wird keine Kernänderung benötigt, da der Proxy die IPC durch die bisher vorhandene Betriebssystemfunktionalität erhält und an den wirklichen Empfänger weiterleiten kann.

Daraus folgt: da der einzige Ansatz ohne Kernänderung, die Stellvertreter-Threads, keine synchrone Kommunikation bereitstellt, ist eine synchrone Kommunikation trotz Überwachung nur mit Kernänderung möglich.

Da in dieser Arbeit die Kommunikationsüberwachung ohne Kernänderung implementiert werden sollte, fällt die Wahl mangels Alternativen auf den Ansatz der Stellvertreter-Threads. Dieser Ansatz wird im folgenden (ab Kapitel 3.4) vertieft und wurde implementiert.

Wenn man eine Kernänderung in Betracht zieht, so muss man zwischen der Erhaltung der Synchronität, was nur mit Kernänderung möglich ist, und dem Minimalitätsprinzip der Mikrokerne, wie Fiasco einer ist, abwägen. Falls man sich dann doch für die Kernänderung entscheidet, sollte man den Ansatz „Transparente Monitore“ implementieren, da bei diesem zumindest noch die Synchronität gewahrt wird.

Wie bereits im Kapitel 2.2 dargelegt wurde, ist Interprozesskommunikation in L4 synchron und ungepuffert. Dass die Synchronität bei einigen Ansätzen nicht erhalten bleibt, wurde bereits betrachtet. Aber die Semantik der IPC wird durch alle 3 angegebenen Ansätze im Hinblick auf die Pufferung verletzt. Die Nachricht wird stets während der Übertragung vom Sender zum Empfänger im Proxy zwischengespeichert (gepuffert). Allerdings führt diese Semantik-Verletzung zu keinen Problemen. Sie ist auch ganz vermeidbar: Dazu muss der Proxy die Nachricht bereits im Adressraum des Senders auswerten und ggf. modifizieren. Anschließend wird die IPC direkt an den Empfänger geschickt. Dazu wäre aber auch eine Kernänderung nötig, da der Proxy sonst keinen Zugriff auf den Adressraum des Senders hätte.

3.3 Dynamisch verteilte Anwendungen mit Kernänderung

Dynamisch verteilte Anwendungen können trotz Kommunikationsüberwachung synchrone IPC nutzen, allerdings setzt die Beibehaltung der Semantik Kernänderungen voraus. Durch die Implementierung des Konzeptes der transparenten Monitore würde jede zu überwachende IPC an den Monitor umgeleitet werden, welcher sie entsprechend bearbeiten kann. Anschließend wird die IPC dem wahren Empfänger zugestellt. Erst jetzt ist die Kommunikationsbeziehung für den Sender beendet. Dadurch würde IPC weiterhin synchron übertragen werden. Anhand des Rückgabecodes weiß der Sender, ob der Empfänger die IPC sicher erhalten hat oder warum die Kommunikation unterbunden wurde.

Eventuell müssten IPC-Fehlercodes hinzugefügt werden. Wenn der Monitor eine Kommunikation abbricht, kann der Monitor somit dem Sender mitteilen, was der Grund für den Abbruch war. Allerdings sollte man beim Einsatz in Sicherheitsanwendungen darüber nachdenken, ob dem Sender überhaupt mitgeteilt werden soll, dass seine Nachricht geblockt wurde.

3.4 Dynamisch verteilte Anwendungen ohne Kernänderung

Wie bereits in Kapitel 3.2 erwähnt, bleibt bei einer Implementation ohne Kernänderung nur noch der Ansatz der Stellvertreter-Threads übrig. Bei diesem Ansatz wird für jeden Thread, der an einer zu überwachenden Kommunikation beteiligt ist, ein Stellvertreter-Thread angelegt. Das bedeutet, wenn man auch nur einen Thread (z.B. Server) überwacht, so kann dieser mit sehr vielen anderen Threads (z.B. Clients) kommunizieren, so dass man sehr viele Stellvertreter-Threads anlegen muss. Darum sollte man zuerst einmal prüfen, ob man die Anzahl der Stellvertreter-Threads reduzieren kann (unter der Nebenbedingung, dass der Einsatz der Stellvertreter-Threads dann immer noch transparent ist).

Reduzierung der Stellvertreter-Threads

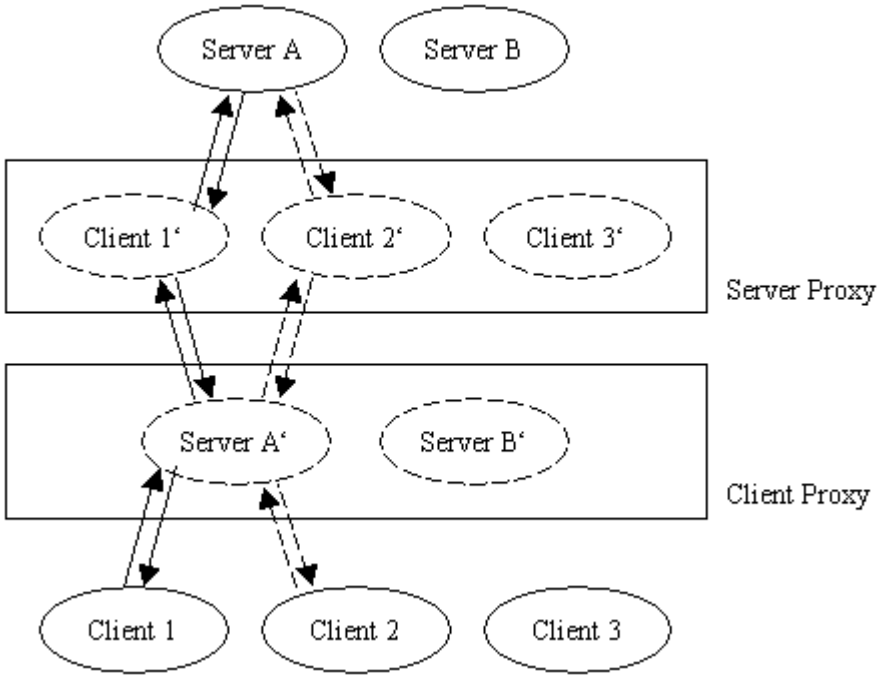


Abbildung 5

Im Folgenden gehe ich von einer Client-Server-Architektur aus. Es sollen die beiden Server A und B überwacht werden. Demzufolge müssen für beide Server Stellvertreter angelegt werden (A' und B'). Des weiteren sollen die Clients 1, 2 und 3 mit den Servern kommunizieren, dazu muss auch für jeden Client ein Stellvertreter angelegt werden.

Wie man auf der Abbildung 5 erkennen kann, sind die Stellvertreter-Threads zweischichtig angelegt. Um die Anzahl der Stellvertreter-Threads zu reduzieren, kann man nun entweder die Anzahl der Threads pro Schicht oder die Anzahl der Schichten reduzieren.

Wenn man eine der beiden Schichten weglässt (in unserem Beispiel die Schicht mit den Client-Stellvertretern) so ist die Überwachung nunmehr alleine durch den Client-Proxy nicht mehr generisch. Wenn sowohl Client 1 als auch Client 2 eine Anfrage an den Server A versendet, so senden beide Clients ihre Anfrage an den Server-Stellvertreter A', welcher die Anfrage an den Server A weiterleitet. Der Server A beantwortet nun beide Anfragen. Dabei ist die Reihenfolge der Antworten egal. Da ein Server die Antworten immer an den Thread verschickt, von welchem er die Anfrage erhielt, versendet der Server A also die beiden Antworten in beliebiger Reihenfolge an den Server-Stellvertreter A'. Dieser kann nun nicht entscheiden, welche Antwort an welchen Thread ausgeliefert wird. Der Stellvertreter-Thread kann zwar die Zuordnung der Anfragen zu den Clients 1 und 2 festhalten, aber er kann die Antworten nicht der jeweiligen Anfrage zuordnen. Damit diese Zuordnung funktioniert, müsste der Server-Stellvertreter über die Logik des Servers Bescheid wissen.

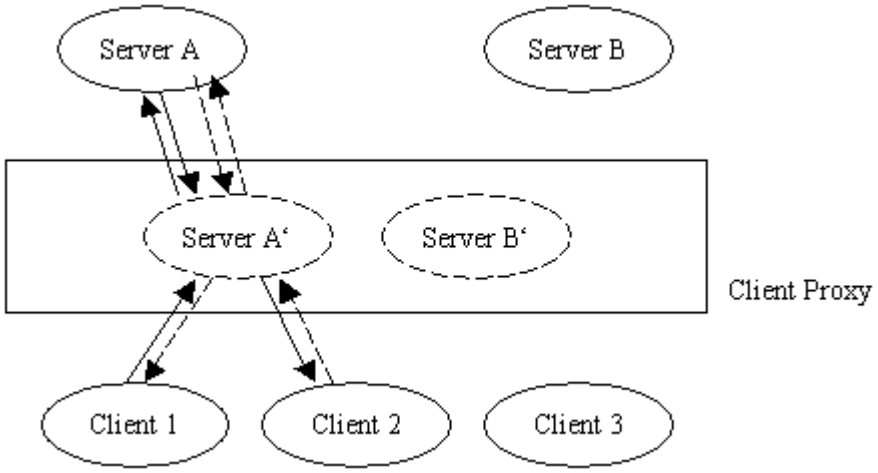


Abbildung 6

Wenn man statt den Client-Stellvertretern die Server-Stellvertreter weglässt, hat man das gleiche Problem, wenn ein Client sowohl mit Server A als auch mit Server B kommunizieren möchte.

Wenn man nun beide Schichten anlegt, aber versucht, die Stellvertreter-Threads pro Schicht zu reduzieren, hat man wiederum genau das gleiche Problem. Wie man in Abbildung 7 erkennt, erhält der Server-Stellvertreter A' beide Antworten von dem Client-Stellvertreter. Auch hier ist A' nicht in der Lage, die Nachrichten richtig zuzuordnen.

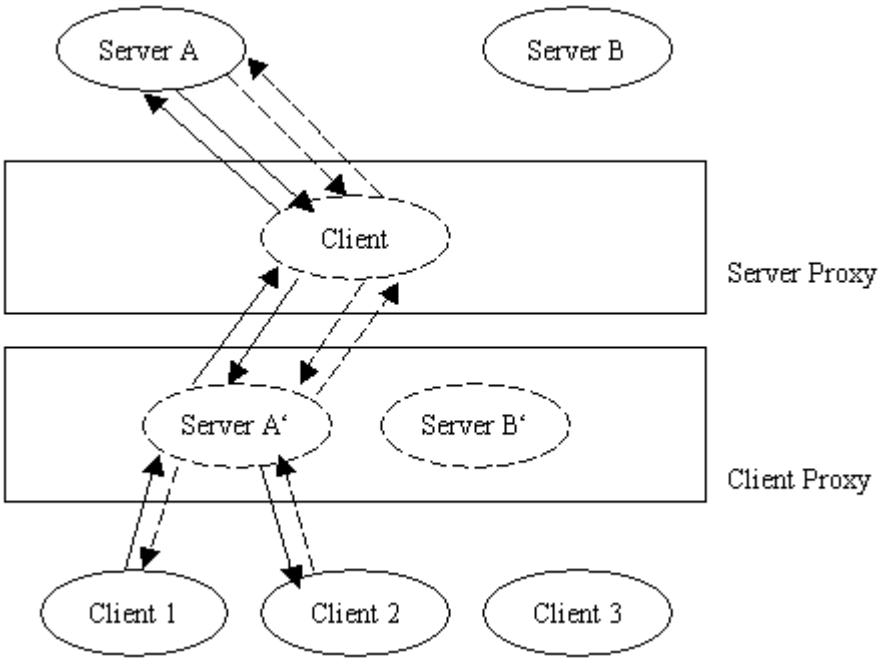


Abbildung 7

Die intuitive Konstellation der Stellvertreter-Threads aus Kapitel 3.2 war, für jeden an einer zu überwachenden Kommunikation beteiligten Thread einen Stellvertreter-Thread anzulegen. Dabei gibt es unter der Vorraussetzung der Client-Server-Architektur zwei Schichten: die Client-Stellvertreter (zusammengefasst im Server-Proxy) und die Server-Stellvertreter (zusammengefasst im Client-Proxy). In diesem Kapitel wurde nun gezeigt, dass sich diese Anzahl der Stellvertreter-Threads (zumindest mit generischen Proxys) nicht reduzieren lässt. Nur so kann man erreichen, dass ein Client verschiedene Server unterscheiden kann und ein Server verschiedene Clients (d.h. das Kommunikationsmuster bleibt aufrecht erhalten).

Funktionsweise

Die Transparenz der Proxys ist für den Einsatz dieser Arbeit besonders wichtig: Wenn man nicht anwendungsspezifische, sondern generische Proxys verwendet, möchte man Proxys mit möglichst wenig Aufwand einsetzen. Vor allem möchte man nicht alle Anwendungen ändern müssen, also müssen generische Proxys auch transparent sein. Dies ist besonders wichtig, wenn man die Client-Server-Architektur zugrundelegt: Der Client versendet seine Anfrage an den Server, dieser bearbeitet die Anfrage und versendet die Antwort an den Client. Die Frage ist aber: Woher kennt der Client den Server und umgekehrt?

Dies veranschaulicht Abbildung 8. In einer Client-Server-Architektur existiert ein Nameserver, bei dem sich alle Server registrieren, bevor sie auf Anfragen warten (1+2). Der Nameserver hat somit für alle registrierten Server die Zuordnung Server-Name⇔Server-ID. Wenn nun ein Client die Funktionalität eines Servers nutzen möchte, so fragt der Client beim Nameserver die ID des Servers ab (3+4). An genau diese ID wird die Anfrage dann gesendet und unmittelbar danach wird die Antwort von genau dieser ID erwartet (5+6, 7+8, usw.). Der Server hingegen erwartet Anfragen von beliebigen Threads. Wenn solch eine Anfrage eintrifft, so erhält der Server außer der eigentlichen Anfrage auch die ID des anfragenden Threads. Diese ID registriert sich der Server, es ist die ID des Clients. Der Server bearbeitet nun also die Anfrage und versendet die Antwort an die ID des Clients. Dieser wartet bereits auf das Resultat und kann es nun verarbeiten.

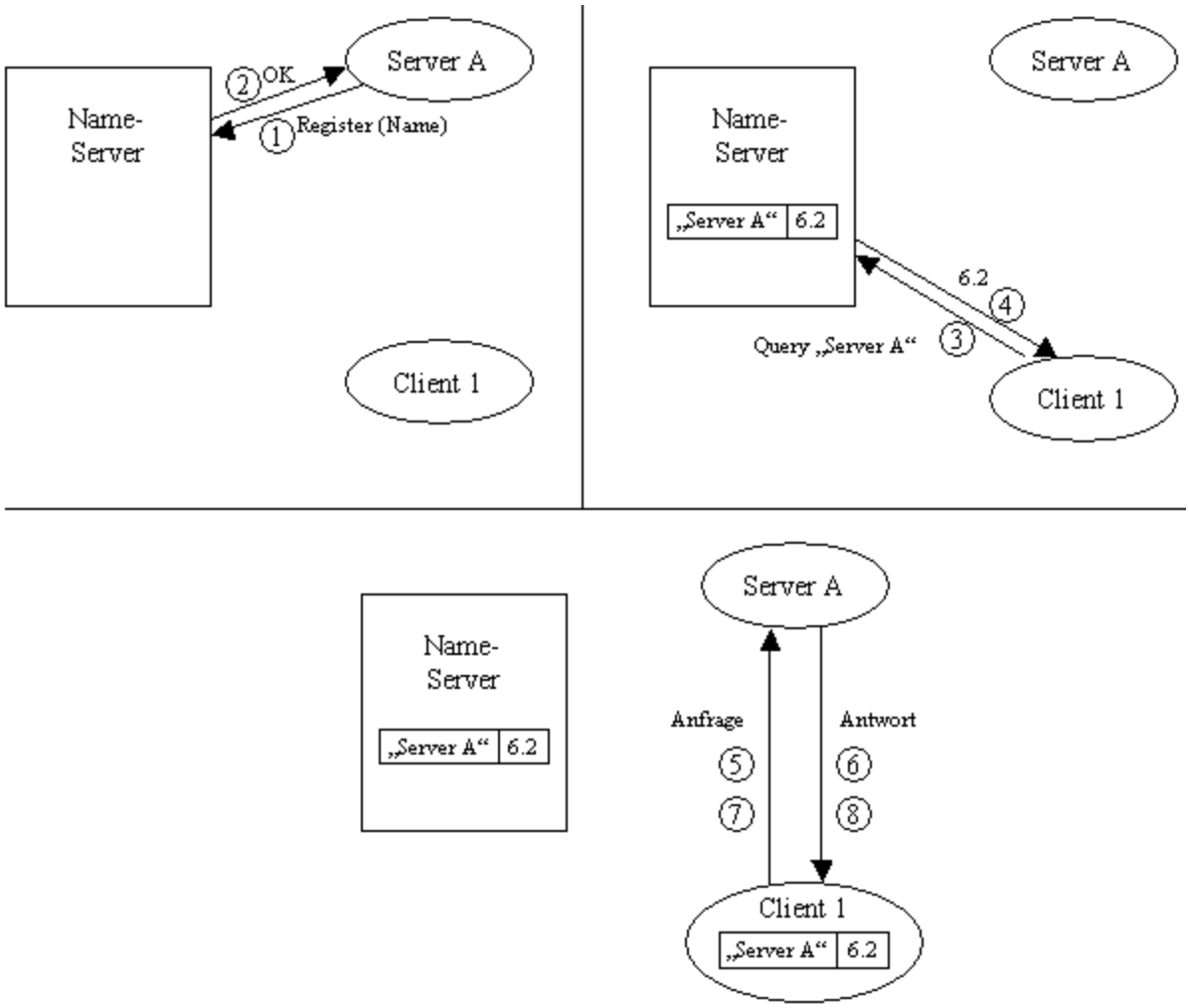


Abbildung 8

Transparente Kommunikationsüberwachung bedeutet nun, dass sich sowohl Client als auch Server genauso wie eben beschrieben verhalten, obwohl ihre Kommunikation überwacht wird. Das heißt, weder Client noch Server bemerken diese Überwachung.

In dem beschriebenem Szenario existiert auch ein Nameserver, bei dem sich die Server registrieren. Allerdings speichert der Nameserver nicht nur die Zuordnung Server-Name⇔Server-ID, sondern er überprüft, ob der Server überwacht werden soll. Wenn das der Fall ist, dann erteilt der Nameserver dem Client-Proxy den Auftrag, einen Stellvertreter-Thread für den Server anzulegen. Der Client-Proxy liefert dem Server die ID des Stellvertreter-Threads zurück. Nun informiert der Nameserver den Server-Proxy über die Zuordnung Server-ID⇔Stellvertreter-ID. Anschließend speichert der Nameserver die Zuordnung Server-Name⇔Stellvertreter-ID und antwortet dem Server mit einer Erfolgsmeldung.

Wenn nun der Client die ID eines Servers erfragt (Abbildung 9, unten), so überprüft der Nameserver, ob der entsprechende Server überwacht wird. Wenn das der Fall ist, so erfolgt die anschließende Client-Server-Kommunikation über die Stellvertreter-Threads. Also benötigt der Client auch noch einen Client-Stellvertreter. Wenn dieser noch nicht existiert, dann veranlasst der Nameserver den Server-Proxy, einen Stellvertreter-Thread für den Client anzulegen und die Stellvertreter-ID zurückzuliefern. Danach teilt der Nameserver dem Client-Proxy die Zuordnung Client-ID⇔Stellvertreter-ID mit. Jetzt beantwortet der Nameserver die Anfrage des Clients und liefert für den Server-Namen die ID des Server-Stellvertreters zurück.

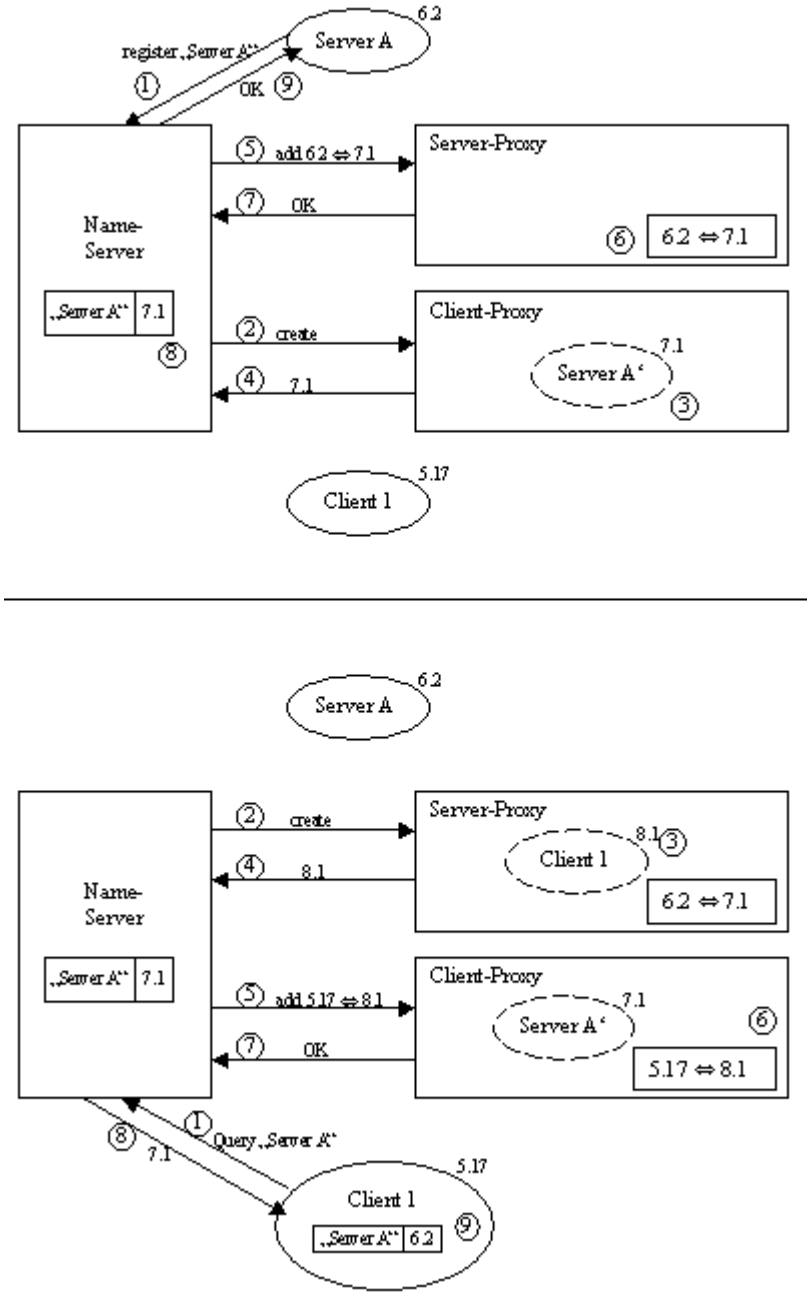


Abbildung 9

Nachdem der Client die (vermeintliche) ID des Servers erhalten hat, kann er mit der Client-Server-Kommunikation beginnen. Das bedeutet, er versendet seine Anfrage einfach an seine Server-ID. Da er aber nicht die ID des Servers, sondern die des Server-Stellvertreters, vom Nameserver erhalten hat, schickt der Client seine Anfrage an den Server-Stellvertreter.

Wenn ein Stellvertreter-Thread eine IPC erhält, so sucht er in seiner Liste mit ID-Paaren nach der ID des Senders. Die entsprechend andere ID ist dann die Empfänger-ID, an die die IPC weitergeleitet wird. In diesem Fall erhält der Server-Stellvertreter also die Anfrage vom Client, schaut in seiner Liste nach und leitet die Anfrage an den Client-Stellvertreters weiter.

Genauso arbeitet auch der Client-Stellvertreter: er sucht die ID des Server-Stellvertreters, findet diese und leitet die Anfrage an den Server weiter.

Der Server erhält nun die Anfrage vom Client-Stellvertreter. Der Server bearbeitet die Anfrage und schickt die Antwort an den anfragenden Thread zurück, also an den Client-Stellvertreter.

Nun schaut der Client-Stellvertreter wieder in der Liste nach und leitet die Antwort an den Server-Stellvertreter weiter. Der Server-Stellvertreter handelt entsprechend und leitet die Antwort an den Client weiter.

Der Client erhält schließlich die Antwort auf seine Anfrage vom Server-Stellvertreter, also von genau dem Thread, an den er die Anfrage gesendet hat.

3.5 Änderung am Nameserver

Die Idee der Stellvertreter-Threads ist: Der Nameserver liefert dem Client nicht die ID des Servers, sondern die ID des Server-Stellvertreters aus. Dies kann man auf zwei verschiedene Arten realisieren: entweder man programmiert einen Teil der Überwachungs-Logik in den Nameserver und den anderen Teil in die Proxys oder man kapselt die komplette Überwachungslogik in den Proxys.

Überwachungslogik komplett im Proxy

Kapselt man die Überwachungslogik im Proxy, so müssen die Proxys beim Start eine Liste der Server erhalten, die überwacht werden sollen. Der Proxy muss nun den Nameserver abfragen, ob sich ein zu überwachender Server bereits beim Nameserver registriert hat. Wenn das der Fall ist, so erzeugt der Proxy einen Stellvertreter-Thread für den Server. Dieser Stellvertreter-Thread entfernt nun den Nameserver-Eintrag des zu überwachenden Servers und registriert sich selbst unter dessen Namen. Dazu ist keine Änderung des Nameservers notwendig, denn die benötigte Funktionalität ist bereits vorhanden.

Ein Nachteil an dieser Lösung ist, dass der Proxy nicht bemerkt, welcher Client die ID des Servers abfragt und somit nur schwer den Client-Stellvertreter anlegen kann. Dieses Problem lässt sich lösen, indem man entweder doch den Nameserver modifiziert oder indem man bei jeder weiterzuleitenden Nachricht die ID des Senders (Clients) auswertet und ggf. einen Stellvertreter anlegt. Dies erhöht jedoch den Kommunikations-Overhead.

Ein weiterer Nachteil besteht darin, dass der Nameserver beim Start bereits diejenigen Server registriert, die per Konvention eine bestimmte ID besitzen. Zu dieser Zeit empfängt der Nameserver noch keine Anfragen, deshalb können die Proxys diese Server auch noch nicht überwachen. Erst wenn der Nameserver schließlich Anfragen empfängt, haben die Proxys eine Chance, die ID der zu überwachenden Server abzufragen. Hat sich so ein Server registriert, so kann der Proxy nun einen Stellvertreter-Thread anlegen und den Server unter der ID des Stellvertreters registrieren. Jetzt erst wird der Server überwacht. Allerdings sind die Anfragen von einem Client und einem Proxy gleichberechtigt. Es kann also vorkommen, dass ein Client gleich nach Systemstart die ID eines benötigten Servers abfragt und diese erhält. Erst später registriert der Proxy unter dem Server-Namen die ID des Server-Stellvertreters. Somit kann der zeitige Client mit dem Server unüberwacht kommunizieren.

Die eben beschriebene Wettlaufsituation kann genauso im laufenden Betrieb des Nameservers auftreten. Wenn sich ein zu überwachender Server registriert, kann zufällig unmittelbar danach ein Client dessen ID abfragen und unüberwacht mit dem Server kommunizieren. Um die Wahrscheinlichkeit für dieses Szenario möglichst weit zu senken (sie kann nie auf Null sinken), sollte der Proxy in möglichst kurzen Zeitabständen Anfragen an den Nameserver stellen, ob sich die zu überwachenden Server inzwischen registriert haben. Das entspricht einer Denial-of-Service-Attacke gegen den Nameserver und ist nicht akzeptabel, wenn man bedenkt, dass die meisten Server des L4Env den Nameserver benutzen.

Der Vorteil dieser Lösung ist das Vermeiden einer Änderung am Nameserver. Des weiteren ist bei dieser Lösung die komplette Proxy-Logik auch in den Proxys implementiert.

Überwachungslogik im Nameserver

Akzeptiert man Änderungen am Nameserver, so können die Nachteile der eben beschriebenen Lösung beseitigt werden. Dazu erhalten nicht die Proxys, sondern der Nameserver beim Start eine Liste der zu überwachenden Server. Sobald sich nun ein Server registriert, kann der Nameserver überprüfen, ob dieser Server überwacht werden muss. Wenn das der Fall ist, so wird als erstes der Stellvertreter-Thread für den Server angelegt und gleich dessen ID im Nameserver gespeichert. Erst dann versendet der Nameserver eine Antwort an den Server und ist für weitere Anfragen bereit. Es ist also keinem Client möglich, die ID eines zu überwachenden Servers abzufragen, da bei der Registrierung eines zu überwachenden Servers sofort die ID des Server-Stellvertreters gespeichert wird.

Sobald ein Client die ID eines Servers abfragt und die ID eines Server-Stellvertreters erhält, wird auch ein Client-Stellvertreter für diesen Client angelegt (falls noch keiner existiert). Wenn der Client also später mit dem Server kommunizieren möchte, so sendet er seine Anfrage an den Server-Stellvertreter. Dieser braucht nun nicht zu überprüfen, ob der entsprechende Client-Stellvertreter existiert, da er ja schon bei der Nameserver-Anfrage angelegt wurde.

Wertung

In meiner Arbeit werde ich den letzteren Ansatz implementieren. Nur mit diesem Ansatz ist gewährleistet, dass die Kommunikation eines zu überwachenden Servers auch in jedem Fall tatsächlich überwacht wird. Mit dem ersten Ansatz, die Überwachungslogik komplett in den Proxys zu implementieren, besteht ständig die Gefahr der oben beschriebenen Wettlaufsituation. Eine sichere Überwachung der Kommunikation erscheint mir wichtiger, als die Vermeidung einer Änderung des Nameservers.

3.6 Timeout-Semantik

Beim Senden und beim Empfangen einer L4-IPC kann man einen Timeout angeben. Dadurch kann ein Thread bestimmen, wie lange er maximal warten (blockieren) möchte, bis sein (oder einer seiner) Kommunikationspartner kommunikationsbereit ist. Nach Ablauf des Timeouts wird die IPC mit einem entsprechendem Fehlercode abgebrochen.

Es gibt drei Timeout-Arten: einen Null-Timeout, einen endlichen Timeout und einen unendlichen Timeout. Ein Null-Timeout entspricht einer blockierungsfreien IPC-Operation. Das bedeutet, die Nachricht kann entweder sofort an den Empfänger zugestellt bzw. vom Sender empfangen werden oder die IPC-Operation wird sofort mit einem Fehlercode beendet. Ein unendlicher Timeout bedeutet, dass der aufrufende Thread solange blockiert wird, bis die IPC zustande kommt. Falls die IPC niemals zustande kommt (z.B. Absturz des Kommunikationspartners), wird der Thread seine Arbeit auch nie fortsetzen können. Und ein endlicher Timeout bedeutet schließlich, dass der aufrufende Thread bis zum Zustandekommen der IPC, jedoch nicht länger als der spezifizierte Timeout, blockiert wird.

Mit dem Verlust der Synchronität kann auch die Timeout-Semantik nicht erhalten werden. Beispielsweise könnte ein Client einem Server eine IPC mit einem Null-Timeout schicken. Dadurch würde der Client erfahren, ob der Server momentan empfangsbereit ist oder ob er noch arbeitet. Bei Kommunikation über Stellvertreter-Threads ist das nicht möglich, weil die IPC vom Server-Stellvertreter entgegengenommen wird, auch wenn der Server momentan nicht empfangsbereit ist. Die Client-Logik weiß aber nicht über die Stellvertreter Bescheid. Sie nimmt an, direkt mit dem Server zu kommunizieren und schließt aus der erfolgreichen

Kommunikation, dass der Server empfangsbereit ist. Dabei kann es sogar sein, dass der Server mittlerweile schon abgestürzt ist. In diesem Fall würde auch der Server-Stellvertreter die IPC lediglich an den Client-Stellvertreter weiterleiten. Dieser würde nun versuchen, die IPC an den Server weiterzuleiten und würde einen Fehler feststellen. Zu dieser Zeit hat der Client aber bereits seine Nachricht erfolgreich (an den Server-Stellvertreter) zugestellt. Damit wäre für ihn die Kommunikation beendet.

Server sind stets so konstruiert, dass sie ihre Ergebnisse mit einem Null-Timeout versenden, weil sie ansonsten ihre eigene Performance drastisch senken würden. Ein Thread, der die Verfügbarkeit eines Servers angreifen möchte, bräuchte andernfalls nur eine Menge von Anfragen an den Server stellen und einfach keine Empfangsoperation ausführen. Der Server würde sich bei jeder Anfrage für das angegebene Timeout selbst blockieren. Wenn also ein Client ein Ergebnis vom Server haben möchte, so hat der Client dafür zu sorgen, dass er auch empfangsbereit ist.

Nachdem der Server ein Ergebnis versendet hat, wartet er auf neue Anfragen. Hierbei gibt es zwei Möglichkeiten: Entweder der Server empfängt und beantwortet ausschließlich Anfragen oder er verrichtet auch noch zusätzliche Arbeit, wenn längere Zeit keine Anfragen eintreffen. Wenn der Server ausschließlich Anfragen empfängt, so empfängt er mit einem unendlichen Timeout, da er, solange keine Anfragen eintreffen, nichts zu tun hat. Wenn der Server auch noch andere Arbeit verrichten soll, so empfängt er Anfragen mit einem gewissen Timeout. Wenn dieses Timeout verstreicht, ohne dass eine Anfrage eintrifft, so wird der Server wieder aktiv und kann die zusätzliche Arbeit verrichten.

Der Timeout für das Senden der Anfrage vom Client an den Server gibt die Zeit an, die zwischen dem Aufruf der eigenen Send-Operation und der Empfangsbereitschaft des Servers maximal vergehen darf. Der Timeout bei der Abfrage des Ergebnisses gibt nun die Zeit an, die zwischen dem Übertragen der Anfrage bis zum Erhalt des Ergebnisses maximal vergehen soll. Dieser Timeout entspricht also der Zeit, die der Server maximal für die Verarbeitung der Anfrage und der Berechnung des Ergebnisses benötigen soll.

Mit welchem Timeout die Clients ihre Anfragen nun versenden und die Antworten empfangen, ist abhängig von dem Vertrauen, dass sie dem Server entgegenbringen. Wenn ein Client darauf vertraut, dass der Server ihn nicht ignoriert und auch nicht abstürzt, so kann er unendliche Timeouts verwenden. Wenn der Client auf die Ergebnisse des Servers unbedingt angewiesen ist (was der Normalfall ist), kann er ebenfalls unendliche Timeouts verwenden. Außerdem dauert die Berechnung und der Einsatz von endlichen Timeouts länger, als der Einsatz von unendlichen Timeouts. Somit sprechen auch Performance-Gründe für den Einsatz von unendlichen Timeouts. Wenn der Client aber auch ohne das Ergebnis des Servers sinnvoll weiterrechnen kann, so sollte er einen endlichen Timeout benutzen. Denn immer, wenn der Client eine Anfrage mit einem unendlichem Timeout stellt, bedeutet ein Server-Absturz auch einen Absturz des Clients. Genauso verklemmt der Client, wenn der Server die Anfrage des Clients einfach ignoriert und verwirft.

Um die Überwachung mittels Stellvertreter-Threads im Hinblick auf die Timeout-Semantik so transparent wie möglich zu gestalten, sollten die Stellvertreter-Threads nach außen die gleichen Sende-Timeouts benutzen, wie der originale Client bzw. Server. Das ist aber nur in gewissen Grenzen möglich, da die Proxys generisch und nicht anwendungsspezifisch sind.

Die Kommunikation zwischen den Stellvertreter-Threads erfolgt in meiner Implementation mit einem unendlichen Timeout. Die Zustellung der IPC von einem Server-Stellvertreter zu einem Client erfolgt mit einem Null-Timeout, die Zustellung der IPC von einem Client-Stellvertreter zum Server dagegen mit einem endlichen Timeout von 30 Sekunden. Wenn die Client-Stellvertreter die IPC an den Server mit einem unendlichen Timeout weiterleiten würden, bestünde die Gefahr, dass aufgrund des Absturzes eines Servers sehr viele Stellvertreter-

Threads abstürzen und somit auch Clients blockieren. Dieses Szenario ist in Abbildung 10 dargestellt.

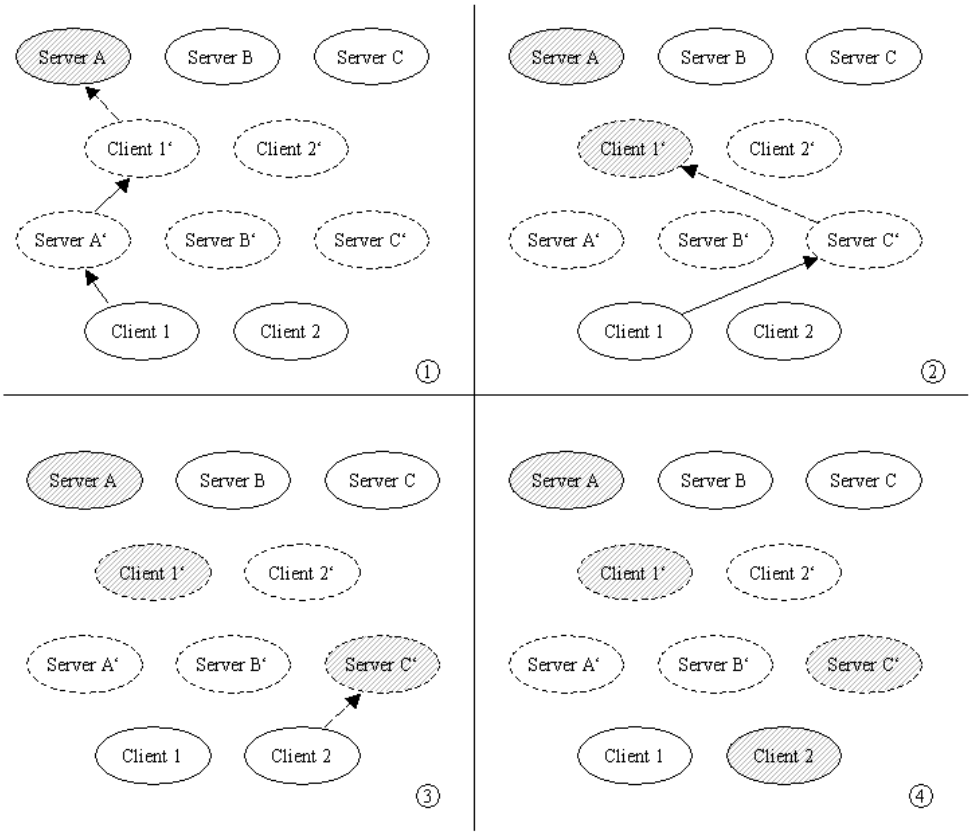


Abbildung 10, Verklemmungsgefahr

Angenommen, Server A ist abgestürzt und Client 1 möchte an ihn eine Anfrage stellen (1). Wenn die Stellvertreter mit unendlichem Timeout ihre Nachrichten zustellen, stürzt daraufhin der Client-Stellvertreter 1' ab. Wenn Client 1 seine Serveranfrage mit einem endlichen Timeout versendet hat, so wird er nach einer gewissen Zeit wieder aktiv und kann nun eine Anfrage an Server C stellen (2). Daraufhin stürzt der Server-Stellvertreter C' ab. Falls nun Client 2 eine Anfrage an Server C mit einem unendlichen Timeout stellt (3), so stürzt auch Client 2 ab (4), obwohl er eine Anfrage an einen bereiten Server gestellt hat. Demzufolge dürfen nicht alle Stellvertreter-Threads mit unendlichem Timeout senden.

Bei dem endlichen Timeout besteht die Gefahr, dass eine aus welchen Gründen auch immer (z.B. Kommunikation über das Netzwerk) verzögerte IPC einfach verworfen wird. Da dies nicht mit einer möglichst transparenten Überwachung der Kommunikation vereinbar ist, sollte dieser Timeout so groß gewählt werden, dass dadurch keine IPC zwischen den Proxys verloren geht. Den Timeout von 30 Sekunden habe ich unter der Annahme gewählt, dass innerhalb von 30 Sekunden jede IPC zugestellt wurde. Benutzt man Anwendungen, bei denen das nicht der Fall ist, so sollte man den Timeout entsprechend korrigieren.

Der Empfangs-Timeout der Stellvertreter-Threads ist generell unendlich. Solange keine IPC eintrifft, hat der Stellvertreter auch keine anderen Aufgaben und kann deshalb auch weiter auf IPC warten.

4 Implementierung

Im Kapitel 3.2. habe ich verschiedene Implementierungsansätze diskutiert und mich für das Konzept der Stellvertreter-Threads entschieden. Dieses Konzept habe ich in Kapitel 3.4 vertieft. Die Funktionsweise besteht darin, dass der Nameserver den anfragenden Clients nicht die ID des Servers aushändigt, sondern die eines Server-Stellvertreters.

In Kapitel 3.5 habe ich 2 mögliche Techniken gezeigt und mich im Interesse einer zuverlässigen Kommunikationsüberwachung für eine Änderung des Nameservers entschieden.

4.1 Implementation der Proxys

Mein Proxy soll auf Anfrage des Nameservers Stellvertreter-Threads bereitstellen. Diese Stellvertreter-Threads erhalten dann IPC, welche von ihnen entgegengenommen und weitergeleitet werden soll. Dazu müssen die Stellvertreter-Threads Kenntnis davon haben, an welchen Thread die IPC von einem gegebenem Thread weitergeleitet werden soll. Ausgehend von diesen Anforderungen habe ich mich für das folgende Design eines Proxys entschieden:

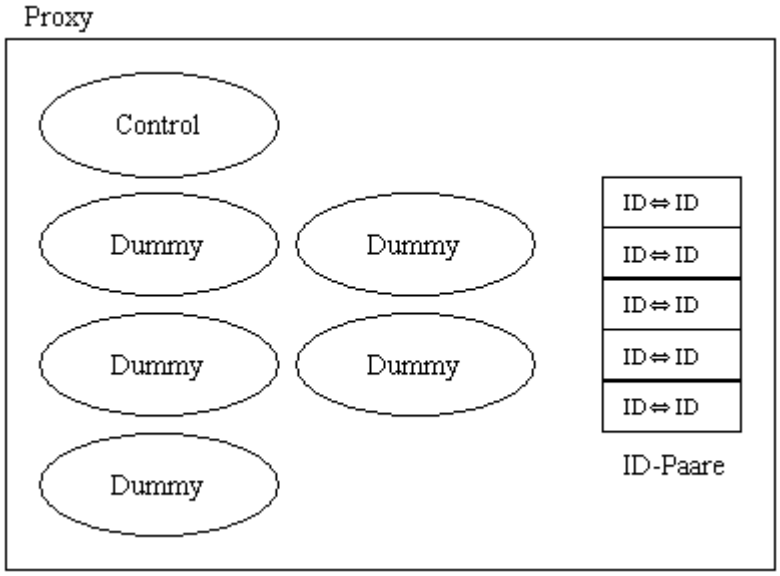


Abbildung 11, Layout eines Proxys

Control-Thread

Der Control-Thread wird beim Laden des Proxys gestartet. Er registriert sich beim Nameserver als Proxy und nimmt alle Anfragen an den Proxy entgegen. Momentan handelt es sich dabei nur um die beiden Anfragen des Nameservers: „Anlegen eines neuen Stellvertreter-Threads“ und „Eintragen eines ID-Paares“. Falls kontrolliert werden soll, ob die zu überwachenden Server noch arbeiten oder schon beendet wurden, so kann man die zwei Anfragen „Beenden eines Stellvertreter-Threads“ und „Entfernen eines ID-Paares“ hinzufügen. Außerdem werde ich für die Leistungsbewertung die Anfrage „verbrauchte Rechenzeit“ hinzufügen, welche die bisher verbrauchte Rechenzeit des angegebenen Stellvertreter-Threads zurückliefert. Wenn man die Stellvertreter-Threads später ergänzt, so dass sie nicht nur IPC weiterleiten, sondern auch eine Arbeit verrichten, dann sollte man die Stellvertreter-Threads konfigu-

rieren. Solch eine Konfiguration könnte man dem Control-Thread mitteilen, und dieser nimmt dann die erforderlichen Änderungen vor. Die vermeintlich bessere Lösung, die Konfiguration direkt an den Stellvertreter-Thread zu senden, scheidet aus, denn der Stellvertreter-Thread kann nicht unterscheiden, ob es sich bei einer IPC um eine Konfiguration handelt, die er bearbeiten soll, oder ob es sich um eine IPC einer zu überwachenden Anwendung handelt (die zufällig einer Konfiguration gleicht), die er weiterleiten soll.

ID-Paare

Da die Stellvertreter-Threads die empfangene IPC weiterleiten sollen, muss ihnen der Ziel-Thread bekannt sein. Dazu existiert im Proxy eine Liste mit ID-Paaren. Wenn ein Stellvertreter-Thread eine IPC von einem beliebigen Thread empfangen hat, so sucht er in der Liste mit den ID-Paaren die ID des Senders. Die jeweils andere ID ist die ID des Empfängers, an den die IPC weitergeleitet werden muss.

Alle ID-Paare gelten grundsätzlich für alle Stellvertreter-Threads.

Dummy-Thread

Die Stellvertreter-Threads in meiner Anwendung leiten die empfangene IPC nur weiter. Sie verrichten keine Arbeit im eigentlichen Sinne. Aus diesem Grund habe ich die Stellvertreter-Threads in meiner Implementation Dummy-Threads genannt. (Dummy soll hier im Sinne von Platzhalter verwendet werden.)

Ein großes Problem bei der Kommunikationsüberwachung ist die Zwischenspeicherung der IPC. Da generische Proxys keine Kenntnis von der Kommunikation haben, die sie überwachen, müssen sie Empfangspuffer angeben, die jede mögliche Nachricht aufnehmen können. Eine Long-IPC kann Nachrichten mit einer Größe von mehr als 130 Megabyte übertragen. Pro Proxy kann man bis zu 127 Stellvertreter-Threads erzeugen, so dass alle Stellvertreter-Threads eines Proxys zusammen mehr als 16 Gigabyte Nachrichtenpuffer benötigen. Das zeigt, dass es schon bei der Long-IPC zu Speicherproblemen kommen kann. Dieses Problem lässt sich lösen, indem man nicht alle 127 möglichen Stellvertreter-Threads benutzt, sondern nur solange Stellvertreter-Threads anlegt, solange genügend Speicher vorhanden ist. Wenn mehr Stellvertreter benötigt werden, so muss eine weitere L4-Task reserviert werden. (Wenn 4 Gigabyte Speicher zur Verfügung stehen, so könnten maximal 31 Stellvertreter-Threads gestartet werden.)

Die in dieser Arbeit implementierten Proxys akzeptieren nur Short-IPC und Long-IPC, aber keine Flexpage-IPC. Mit einer Flexpage-IPC kann man einen kompletten Adressraum übertragen. Damit ist es unmöglich, eine solche IPC zwischenzuspeichern. Selbst wenn man pro L4-Task nur einen Stellvertreter-Thread anlegt, besitzt dieser Stellvertreter-Thread genau einen Adressraum, in dem er auch eigene Daten und eigenen Code speichern muss. Es ist also nicht mehr genügend virtueller Speicher vorhanden, um jede Flexpage-IPC zwischenzuspeichern. Die einzig mögliche Lösung des Problems ist die Variante, die weiterzuleitende Nachricht nicht mehr zwischenzuspeichern. Dazu braucht der Proxy Zugriff auf den Adressraum des Senders, um die Nachricht schon dort zu überprüfen und dann ungepuffert an den Empfänger übertragen zu können. Durch den Zugriff auf einen fremden Adressraum werden allerdings Kernänderungen zwingend notwendig. Hier bietet sich das in Kapitel 3.2 beschriebene Konzept der transparenten Monitore an.

Bei der Implementation des Weiterleitens einer Long-IPC ist mir aufgefallen, dass die L4-Spezifikation [1] an einer Stelle nicht ganz genau ist. Wenn man eine Long-IPC sendet und eine empfängt, so existiert zu jedem String ein Deskriptor (l4_strdope_t). Dieser Deskriptor enthält ein vierelementiges Feld, welches aus zwei Zeigern und zwei Datenwörtern besteht. Die Zeiger referenzieren den zu sendenden bzw. den zu empfangenen String. Die Datenwörter heißen „rcv string size“ und „snd string size“, eine Erklärung ist in [1] nicht zu finden. Wenn man die Bedeutung vom Namen ableitet, so könnte man meinen, dass vor dem Senden in „snd string size“ die Größe des zu sendenden Strings eingetragen wird und nach dem Empfangen in „rcv string size“ die Größe des empfangenen Strings steht. Tatsächlich steht die Größe des empfangenen Strings aber in „snd string size“ (zumindest bei der Fiasco-Implementierung). Dieses Verhalten erleichtert die Arbeit, da beim Weiterleiten einer IPC die Größe des Strings nicht erst nach „snd string size“ kopiert werden muss und so Rechenzeit gespart wird, allerdings sollte dieses Verhalten dokumentiert werden.

Meine Implementierung nutzt die in Kapitel 3.6 vorgeschlagenen Timeouts. Untereinander kommunizieren die Dummy-Threads mit einem unendlichen Timeout. Der Server-Stellvertreter liefert die IPC an den Client mit einem Null-Timeout aus und der Client-Stellvertreter liefert jede IPC an den Server mit einem Timeout von 30 Sekunden aus. Falls dieser Timeout für eine zu überwachende Anwendung zu knapp gewählt wurde, sollte der Timeout unbedingt erhöht werden (gen_proxy/server/include/proxy_def.h), um dem Verlust von IPC durch Timeouts vorzubeugen.

Bei der Erweiterung meiner Arbeit für den Einsatz in verteilten Anwendungen brauchen die Timeouts nicht geändert werden. Wenn die Zeit für die Netzwerkübertragung in die Timeouts der Dummy-Threads einfließt, dann nur in den Timeout für die Kommunikation untereinander.

Falls ein Empfänger auf eine Long-IPC wartet, aber eine Short-IPC eintrifft, so empfängt er zwar eine Short-IPC, bekommt aber den Eindruck, er habe eine Long-IPC empfangen, die aus genau 2 Datenwörtern besteht.

Die Dummy-Threads warten auf irgendeine Long-IPC. Nach deren Erhalt bestimmen sie den nächsten Empfänger und leiten die IPC weiter. An dieser Stelle habe ich eine Short-IPC Optimierung eingefügt: mit der Optimierung prüfen die Dummy-Threads vor dem Weiterleiten, ob die Long-IPC aus genau zwei Datenwörtern besteht. Wenn das der Fall ist, dann leiten sie die IPC als Short-IPC weiter und verkürzen somit die Übertragungszeit.

4.2 Änderungen am Nameserver

Bevor am Nameserver irgendwelche Veränderungen vorgenommen wurden, sollte er kopiert und umbenannt werden. Da die „Building Infrastructure for DROPS“ (BID) standardmäßig (Modus sigma0) den originalen Nameserver verlinkt, empfiehlt es sich, den modifizierten Nameserver als Standard einzutragen. (Dazu muss die Datei l4/mk/modes.inc entsprechend angepasst werden.) Somit braucht man beim Übersetzen aller Anwendungen nur noch die Stellen verändern, an denen die Nameserver-Bibliothek explizit verlinkt oder referenziert wird. Diese Stellen kann man durch einfaches Suchen finden.

Die im Folgenden beschriebenen Änderungen am Nameserver habe ich soweit wie möglich in einer eigenen Datei (modified_names/server/src/proxy.c) gespeichert. Außerdem kann man bei der Konfiguration des Nameservers einstellen, ob er mit oder ohne Proxy-Unterstützung compiliert werden soll.

Die vorgenommenen Änderungen unterteile ich im Folgenden in zwei Arten: dem Hinzufügen der Grundfunktionalität (um Kommunikation zu überwachen) und zusätzliche Funktionalität, um die zu überwachenden Server zur Laufzeit zu verändern.

Hinzufügen der Grundfunktionalität

Die Grundfunktionalität für das Überwachen von Kommunikation beinhaltet die in Kapitel 3.5 beschriebene Funktionalität.

Beim Start des Nameservers wird ihm über die Kommandozeile mitgeteilt, welche Server er überwachen soll. Da im originalem Nameserver schon eine Funktion zum Auslesen der Kommandozeilenparameter existiert, brauchte diese nur ergänzt werden. Um die zu überwachenden Server zu speichern, wurde dem Nameserver eine interne Liste hinzugefügt.

Als nächstes muss dem Nameserver bekannt sein, welche ID die beiden Proxys besitzen, damit der Nameserver Anfragen an sie stellen kann. Dazu registrieren sich die beiden Proxys beim Nameserver (wie alle anderen Server im System auch). Dadurch weiß der Nameserver, ab wann die Proxys einsatzbereit sind. Eine andere Möglichkeit wäre, dass der Nameserver während der Initialisierung wartet, bis sich beide Proxys bei ihm angemeldet haben. Damit wären die Proxys zwar in jedem Fall einsatzbereit, sobald der Nameserver seine Arbeit beginnt, allerdings würde der Nameserver keine Anfragen empfangen, solange die Proxys nicht einsatzbereit sind. In meiner Implementation kann der Nameserver selbst dann alle Anfragen empfangen und bearbeiten, wenn die Proxys noch nicht gestartet wurden. Natürlich wird die Kommunikation dann nicht überwacht, aber die ursprüngliche Nameserver-Funktionalität ist zumindest gewährleistet.

Weiterhin wurde in Kapitel 3.5 beschrieben, dass die Funktionalität des Nameservers beim Registrieren von Servern erweitert werden muss. Sobald sich ein Server registriert, überprüft der Nameserver, ob es sich um einen zu überwachenden Server handelt. Wenn das der Fall ist, so existieren nun 2 Alternativen: Wenn die Proxys noch nicht einsatzbereit sind, so können die Server auch noch nicht überwacht werden. Stattdessen merkt sich der Nameserver in einer weiteren internen Liste diesen „zu früh gestarteten“ Server. Falls beide Proxys einsatzbereit sind, so stellt der Nameserver eine Anfrage an den Client-Proxy: „Erzeugen eines Stellvertreter-Threads“. Nachdem der Server-Stellvertreter erzeugt wurde, liefert der Client-Proxy dessen ID an den Nameserver zurück. Der Nameserver kennt nun die originale Server-ID und die ID des Server-Stellvertreters. Nun informiert der Nameserver den Server-Proxy über dieses ID-Paar. Dieser Vorgang ist in Abbildung 9 dargestellt.

Wenn es sich beim Registrieren eines Servers um den zweiten Proxy handelt, sind die Proxys ab diesem Moment einsatzbereit. Wenn der Nameserver allerdings sofort beginnt, Stellvertreter für die bereits registrierten Server anzulegen, tritt ein Deadlock auf: Der Proxy registriert sich beim Nameserver und wartet auf eine Antwort. Solange der Nameserver die Anfrage des Proxys nicht beantwortet hat, kann er also auch seinerseits keine Anfrage an den Proxy stellen. Deshalb legt der Nameserver zu diesem Zeitpunkt noch keine Stellvertreter an, sondern merkt sich dies nur vor und beantwortet die Anfrage des Proxys. Sobald die nächste Anfrage beim Nameserver eintrifft (von welchem Thread auch immer) legt der Nameserver erst einmal für jeden bereits registrierten (und zu überwachenden) Server einen Stellvertreter an und informiert den Server-Proxy über jedes ID-Paar.

Wenn ein Client die ID eines zu überwachenden Servers erfragt, so muss wieder unterschieden werden, ob die Proxys einsatzbereit sind oder nicht. Wenn sie noch nicht einsatzbereit sind, so führt der Nameserver seine ursprüngliche Funktionalität aus und gibt dem Client die originale Server-ID zurück. In diesem Fall kann die Kommunikation nicht überwacht werden. Allerdings kann der Nameserver die Anfrage des Clients nicht solange zurückstellen, bis beide Proxys einsatzbereit sind. Wenn die Proxys z.B. nicht gestartet wer-

den, würde das bedeuten, jeder Client blockiert, sobald er nach der ID eines zu überwachenden Servers fragt. Falls nun ein Client nach der ID eines zu überwachenden Servers fragt und die Proxys einsatzbereit sind, so muss auch dieser Client (genau) einen Stellvertreter besitzen. Dazu führt der Nameserver eine dritte interne Liste, auf der alle Clients vermerkt sind, die bereits einen Stellvertreter-Thread besitzen. Falls der Client noch nicht in dieser Liste vermerkt wurde, so wird im Server-Proxy ein Stellvertreter für ihn angelegt, dem Client-Proxy das ID-Paar mitgeteilt und der Client der Liste hinzugefügt. Anschließend wird die Anfrage des Clients mit der ID des Server-Stellvertreters beantwortet.

Dynamisches Überwachen

Nachdem der Nameserver nun wie beschrieben modifiziert wurde, kann die Kommunikation der beim Start angegebenen Server überwacht werden. Anschließend habe ich den Nameserver noch so ergänzt, dass man zur Laufzeit die zu überwachenden Server ändern kann. Dazu habe ich den Nameserver um zwei Anfragen ergänzt: „Zu überwachenden Server hinzufügen“ und „Server nicht mehr überwachen“. Außerdem habe ich die Nameserver-Einträge ergänzt, um die Information zu speichern, ob der Server überwacht wird. Das größte Problem an dieser Stelle war, dass die beiden neuen Nameserver-Anfragen zu jedem Zeitpunkt gestellt werden können. Das heißt, es muss geprüft werden, ob beide Proxys schon einsatzbereit sind, ob sich der entsprechende Server schon registriert hat und ob dieser Server bisher oder schon einmal früher überwacht wurde. Für jede Kombination dieser Zustände muss entsprechend reagiert werden. Vor allem darf für jeden Server nur maximal ein Server-Stellvertreter existieren. Außerdem darf der Server-Stellvertreter nicht gelöscht werden, nur weil der Server nicht mehr überwacht wird. Immerhin könnten noch Clients die ID des Server-Stellvertreters besitzen und mit diesem kommunizieren. Das bedeutet wiederum, dass dieser Stellvertreter bei einer erneuten Überwachung wiedergefunden werden muss.

4.3 Testanwendung

Arithmetik-Server

Zu Beginn meiner Arbeit habe ich einen kleinen Arithmetik-Server implementiert. Dieser erzeugt 6 Threads. Jeder Thread registriert sich beim Nameserver mit seinem Namen und übernimmt dann eine Rechenart. Die ersten 4 Arithmetik-Server führen die 4 Grundrechenoperationen aus. Dazu warten sie auf eine Short-IPC. In den Datenwörtern sind die Operanden gespeichert. Der betroffene Server berechnet das Ergebnis und sendet es als Short-IPC zurück. Der fünfte Arithmetik-Server wartet auf eine Long-IPC, welche aus mehreren Datenwörtern besteht. Diese Datenwörter werden von dem Server als Summanden aufgefasst und addiert. Das Ergebnis wird wieder in einer Short-IPC zurückgesendet. Der sechste Server empfängt als Anfragen Long-IPC, die aus mehreren Datenwörtern und indirekten Strings bestehen. Die empfangenen Strings werden mehrmals auf dem Bildschirm ausgegeben und es wird eine Erfolgsmeldung als Short-IPC zurückgesendet.

Anschließend habe ich noch eine kleine Testanwendung implementiert, welche als Client dieser Arithmetikserver fungiert. Dieses Programm fragt den Nameserver nach den IDs aller 6 Arithmetik-Server. Danach stellt es einige Anfragen an diese Server.

Mit dieser Testanwendung lässt sich die Funktionsweise und die Transparenz der Kommunikationsüberwachung sehr einfach zeigen. Die Arithmetik-Server können sowohl überwacht als auch nicht überwacht werden, ohne dass sie dazu verändert werden müssen.

Log-Server Beispiel

Nachdem ich die Implementierung und die Leistungsbewertung meiner Proxys (und der Nameserver-Änderung) abgeschlossen hatte, habe ich den Log-Server überwacht. Für diesen sind schon mehrere Clients als Beispiel-Anwendungen implementiert. Die Beispiel-Anwendung, die ich verwende, erfragt beim Nameserver die ID des Log-Servers. Anschließend erzeugt die Beispiel-Anwendung mehrere Threads, welche alle die ID des Log-Servers benutzen, um ihre Ausgaben zu loggen.

Ich habe diese Beispiel-Anwendung gewählt, da sie nicht dem in Kapitel 3.1 beschriebenem Szenario der Client-Server-Kommunikation entspricht. Bei diesem Szenario erfragt jeder Thread die ID eines Servers beim Nameserver. Danach kommuniziert der Thread mit dem Server beliebig oft. In der gewählten Beispiel-Anwendung erfragt allerdings nur ein Thread die ID des Servers beim Nameserver. Anschließend erzeugt dieser Thread mehrere neue Threads und teilt sich mit diesen die ID des Servers. Nun versuchen alle Threads, auch die, die nicht beim Nameserver angefragt haben, mit dem Server zu kommunizieren.

In Kapitel 3.4 wurde bereits die Funktionsweise der Kommunikationsüberwachung beschrieben: ein Client-Stellvertreter wird dann angelegt, wenn ein Client beim Nameserver die ID eines zu überwachenden Servers erfragt und dieser Client noch keinen Stellvertreter besitzt. In der Beispiel-Anwendung haben alle Threads bis auf den ersten keine Anfrage an den Nameserver gestellt. Somit existiert auch kein Client-Stellvertreter für diese Threads. Weiterhin hat der erste Thread bei seiner Anfrage an den Nameserver nicht die wirkliche ID des Log-Servers erhalten, sondern die ID des Server-Stellvertreters. Schließlich wird der Log-Server ja überwacht. Demzufolge kommunizieren alle Threads der Beispiel-Anwendung mit einem Server-Stellvertreter. Aber nur für den ersten Thread existiert auch ein Client-Stellvertreter, an den die IPC weitergeleitet werden kann. Senden die anderen Threads nun eine IPC an den Server-Stellvertreter, so ist dem Server-Stellvertreter nicht bekannt, an welchen Thread im System er die IPC weiterleiten soll.

Das beschriebene Problem lässt sich lösen, wenn die Stellvertreter-Threads in die Lage versetzt werden, selbst neue Stellvertreter-Threads anzulegen. Dazu habe ich die bisher im Nameserver ergänzte Funktionalität teilweise auch in die Stellvertreter-Threads eingefügt. Wenn ein Stellvertreter-Thread eine IPC erhält, so sucht er in seiner Liste mit den ID-Paaren die ID des Senders. Die andere ID ist dann die ID des Empfängers, an den die IPC weitergeleitet werden muss. Findet der Stellvertreter-Thread den Sender allerdings nicht in der Liste, so ist der oben beschriebene Fall eingetreten. Nun versendet der Stellvertreter-Thread die Anfrage „Anlegen eines neuen Stellvertreter-Threads“ an den Control-Thread des jeweils anderen Proxys und erhält die ID des erzeugten Threads. Die erhaltene ID und die ID des Senders teilt der Stellvertreter-Thread nun seinem eigenem Control-Thread in der Anfrage „Eintragen eines ID-Paares“ mit. Anschließend muss der Nameserver noch informiert werden, für welchen Client soeben ein Stellvertreter-Thread angelegt wurde. Das ist erforderlich, damit der Nameserver für diesen Client nicht noch einen weiteren Stellvertreter-Thread anlegt, falls dieser Client eine ID beim Nameserver erfragt.

Mit diesen Änderungen ist das beschriebene Problem gelöst. Der Stellvertreter-Thread sorgt nun beim Erhalt einer IPC von einem unbekanntem Thread selbst dafür, dass für diesen Thread ein Stellvertreter angelegt wird.

5 Leistungsbewertung

5.1 Testszenario

Für die Leistungsbewertung der generischen Proxys sind hauptsächlich 2 Fragen interessant:

- Um wie viel ist eine überwachte Kommunikation langsamer als eine nicht überwachte?
- Es wurde die Funktionalität des Nameservers erweitert. Inwieweit haben sich dadurch dessen Antwortzeiten geändert?

Die Antwortzeiten des Nameservers verändern sich hierbei nicht nur für die Threads, deren Kommunikation überwacht wird, sondern für alle Threads, die den Nameserver benutzen. Schließlich muss der Nameserver bei jeder Anfrage erst einmal überprüfen, ob es sich um Threads handelt, deren Kommunikation überwacht wird, oder um andere Threads, bei denen lediglich die ursprüngliche Funktionalität ausgeführt werden muss.

Durch meine Arbeit wurden drei Nameserver-Funktionen erweitert. Zum einen die Funktionalität „Server registrieren“ und „Server entfernen“. Die Veränderung dieser Nameserver-Funktionen habe ich nicht ausgemessen, da sie nur sehr selten ausgeführt werden: der Server registriert sich nur einmal, bevor er Anfragen empfängt, und entfernt später seinen Eintrag wieder, bevor er sich beendet (oder entfernt sich nicht, weil er sich nicht beendet). Eine eventuelle Laufzeitverlängerung ist hier ohne Belang.

Die dritte Nameserver-Funktion, die ich in meiner Arbeit verändert habe, ist das Auflösen eines Server-Namen durch einen Client. Diese Nameserver-Funktionalität wird weit häufiger in Anspruch genommen, als die beiden eben beschriebenen. Wenn man z.B. einen Server betrachtet, so nutzt er, wie eben beschrieben, in seiner Lebenszeit einmal die Funktionalität „Server registrieren“ und bis zu einmal die Funktionalität „Server entfernen“. Aber jeder Client, der irgendeine Funktionalität des Servers in Anspruch nehmen möchte, muss sich vorher die ID des Servers mittels der Nameserver-Funktionalität „Server-Namen auflösen“ besorgen. Aus diesem Grund werde ich die Dauer dieser Nameserver-Funktionalität ausmessen.

Für die Leistungsbewertung benutze ich die im Kapitel 4.3 eingeführten Arithmetik-Server. Allerdings habe ich die Server-Funktionalität entfernt, d.h. die Server empfangen lediglich eine Anfrage, prüfen diese auf Fehler und schicken einfach eine Erfolgsmeldung zurück.

Zusätzlich habe ich noch eine weitere kleine Testanwendung implementiert, welche den Nameserver nach der ID der 6 Arithmetik-Server fragt. Direkt vor und nach jeder Anfrage beim Nameserver wird der Zeitstempel des Prozessors ausgelesen und mit Hilfe der Taktfrequenz die Dauer der Nameserver-Anfrage berechnet.

Danach werden 1000 Anfragen einem Arithmetik-Server (entsprechend des jeweiligen Testfalles) gestellt. Die Dauer dieser Anfragen wird ebenfalls mit Hilfe des Prozessor-Zeitstempels gemessen.

Anschließend wird noch die verbrauchte Rechenzeit aller Threads beider Proxys ausgegeben. Diese Rechenzeit wird mit Hilfe des L4-Systemaufrufes „thread_schedule“ bestimmt. Mit diesem Systemaufruf erhält man die Rechenzeit, die der angegebene Thread seit seinem Start verbraucht hat. Findet eine IPC zwischen zwei Threads statt, so wird die dafür benötigte Rechenzeit auf Sender und Empfänger aufgeteilt.

Für die Leistungsbewertung benutze ich ein Computersystem mit einem AMD-K6-2/350 Prozessor und 128 MB Hauptspeicher als Testrechner.

5.2 Anfrage an den Nameserver

Wie im letzten Kapitel bereits erwähnt, wurde bei jedem Durchlauf des Testprogramms die Dauer der Nameserver-Funktionalität „Server-Namen auflösen“ für jeden der 6 Arithmetik-Server ermittelt. Dabei habe ich die Dauer der Namensauflösung für die folgenden 3 Testfälle gemessen:

1. Nameserver ohne Proxys konfiguriert (entspricht dem Original-Nameserver)
2. Nameserver mit Proxys konfiguriert, aber Proxys nicht gestartet
3. Nameserver mit Proxys konfiguriert, Proxys gestartet (entspricht Überwachung)

In allen 3 Testfällen wurde das Testprogramm mehrmals ausgeführt und das Endergebnis ist der arithmetische Mittelwert aller Messergebnisse.

Bei jedem Durchlauf war auffällig, dass die erste Nameserver-Anfrage wesentlich länger gedauert hat, als alle weiteren Anfragen. Das liegt daran, dass der Nameserver bei der ersten Anfrage noch Seitenfehler auslöst und der entsprechende Speicher in den Hauptspeicher eingelagert werden muss. Bei jeder weiteren Anfrage ist der benötigte Speicher nun bereits im Hauptspeicher vorhanden und die Dauer der Anfrage ist dementsprechend kürzer. Aus diesem Grund habe ich jeweils die Dauer der ersten Namensauflösung und die Dauer aller weiteren Namensauflösungen zusammengefasst. Es wurden folgende Ergebnisse erzielt:

Testfall	1. Anfrage	2.-6. Anfrage
1 Original-Nameserver	63	28,3
2 keine Proxies gestartet	63	29,8
3 Kommunikation überwacht	929	30,8

Dauer der Namensauflösung, in Mikrosekunden

Es verlängert sich wesentlich die Dauer der ersten Nameserver-Anfrage. Das liegt weniger an den Seitenfehlern, die auch bei überwachter Kommunikation auftreten, als vielmehr an dem bereits beschriebenem Verhalten des Nameservers. Wenn ein Client die ID eines zu überwachenden Servers abfragt, so muss auch für den Client ein Stellvertreter-Thread im Server-Proxy angelegt werden. Außerdem muss der Client-Proxy über die Zuordnung der originalen ID zur Stellvertreter-ID informiert werden.

Bei den restlichen Nameserver-Anfragen erhöht sich die Dauer nur unwesentlich (kleiner 9%). Das liegt daran, dass der Nameserver überprüft, ob der Server, dessen ID erfragt wird, überwacht wird und außerdem prüfen muss, ob für den anfragenden Client bereits ein Stellvertreter-Thread existiert. Allerdings existiert eben dieser Stellvertreter-Thread bereits und muss nicht mehr angelegt werden.

Meiner Meinung nach ist die längere Dauer der ersten Nameserver-Anfrage kein großes Problem, da diese Dauer vom Nutzer nicht wahrgenommen wird und nur einmal pro Client auftritt, unabhängig davon, ob der Client mit mehreren verschiedenen Servern kommuniziert (dann ist die Dauer jeder weiteren Anfrage wieder gering), oder ob er mit dem ersten Server häufiger kommuniziert (dann muss der Client dessen Namen überhaupt nicht mehr auflösen).

Beim Einsatz in verteilten Systemen ist das auch kein Problem, da die Übertragung der IPC über das Netzwerk ohnehin länger dauert.

5.3 Kommunikationsdauer

Bei der Messung der Kommunikationsdauer zwischen Client und Server habe ich zwischen 5 Testfällen unterschieden:

a. Short-IPC (mit Short-IPC-Optimierung der Stellvertreter-Threads)
b. Short-IPC (ohne Short-IPC-Optimierung der Stellvertreter-Threads)
c. Long-IPC (3,0)
d. Long-IPC (16,0)
e. Long-IPC (16,8)

Jeden dieser 5 Testfälle habe ich sowohl mit überwachter Kommunikation (also über Stellvertreter-Threads) als auch mit unüberwachter (also direkter) Kommunikation durchgeführt, bis auf Testfall a. Diesen habe ich nur mit überwachter Kommunikation durchgeführt, da der einzige Unterschied zu Testfall b die Short-IPC-Optimierung durch die Stellvertreter-Threads ist. Demzufolge sind Testfall a und b ohne Stellvertreter-Threads identisch.

Insgesamt gibt es also 9 Versuche. Diese habe ich alle mehrmals durchgeführt und das Ergebnis als arithmetischen Mittelwert berechnet. Außerdem habe ich jeweils 1000 Serveranfragen pro Durchlauf durchgeführt. (Bei Testfall a und b habe ich 1000 Anfragen pro Arithmetik-Server, also insgesamt 4000 Anfragen, ausgeführt.)

In der folgenden Tabelle sind die gemessenen Antwortzeiten (pro Serveranfrage in Mikrosekunden) angegeben, sowohl mit überwachter als auch mit unüberwachter Kommunikation. Außerdem habe ich bei der überwachten Kommunikation die Rechenzeit aller beteiligten Stellvertreter-Threads gemessen.

Testfall	Serveranfrage, direkt	Serveranfrage, überwacht	Rechenzeit Stellvertreter
a	6,8	35,8	28,2
b	6,8	49,6	41,7
c	13,6	53,4	39,5
d	13,9	54,1	39,8
e	18,4	78,5	57,1

Dauer einer Serveranfrage, in Mikrosekunden

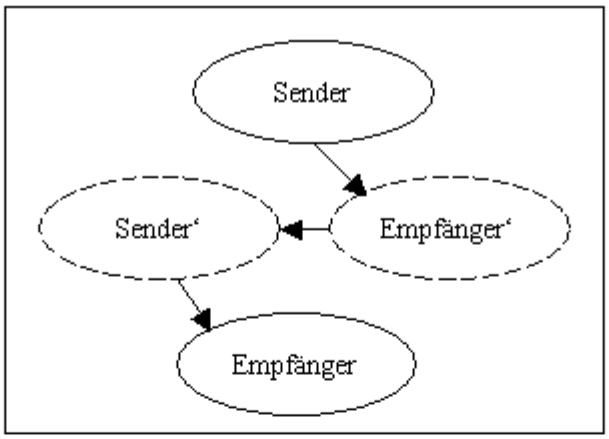
Wie man sieht, benötigt die überwachte Serveranfrage wesentlich mehr Zeit als die unüberwachte. Das war zu erwarten, da bei der überwachten Serveranfrage nicht 2, sondern 6 IPC-Operationen ausgeführt werden und außerdem noch die Rechenzeit der Stellvertreter-Threads hinzukommt.

Bei Testfall b kann man erkennen, dass sich die Dauer einer Serveranfrage bei einer Short-IPC unverhältnismäßig stark erhöht. Das liegt daran, dass eine Short-IPC auf dem Testrechner ca. 7 Mikrosekunden schneller ist, als eine Long-IPC. Da die Stellvertreter-Threads nicht wissen, ob sie eine Short-IPC oder eine Long-IPC als nächstes empfangen, warten sie auf eine Long-IPC. Demzufolge werden bei der direkten Kommunikation 2 Short-IPC versendet, bei der überwachten Kommunikation dagegen 2 Short-IPC und 4 Long-IPC. Im Testfall a wird die Short-IPC optimiert, das bedeutet, es werden bei der überwachten Serveranfrage 6 Short-IPC versendet.

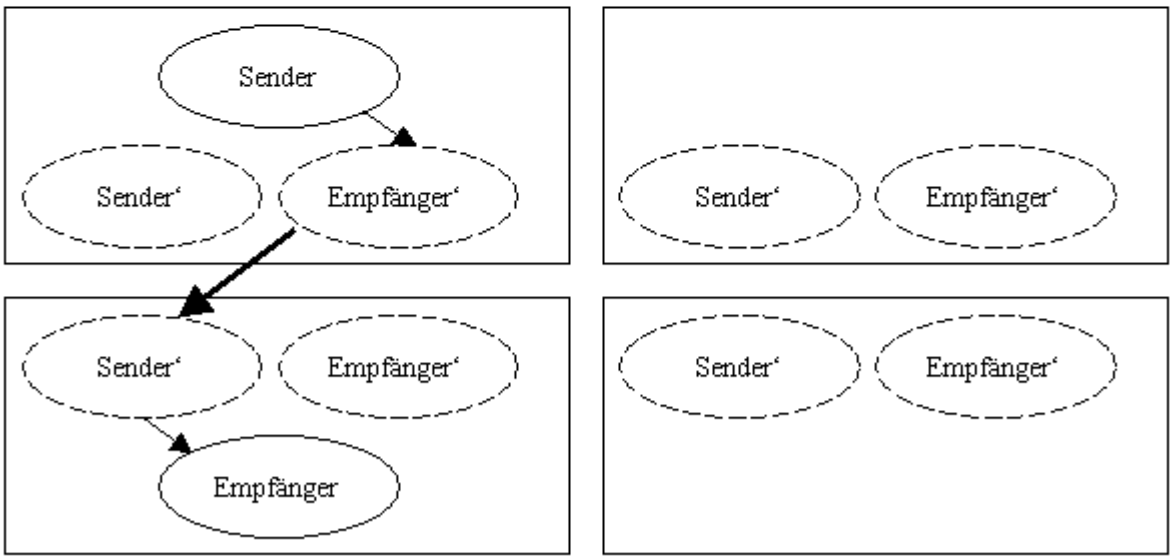
6 Ausblick

6.1 Verteilte Anwendungen

Um auf meiner Arbeit basierend netzwerktransparent zu kommunizieren, muss man die Funktionalität der Netzwerkkommunikation in die Stellvertreter-Threads einfügen. Dazu kann man das Netzwerktreiberprojekt Oshkosh verwenden. Dieses Projekt portiert Netzwerktreiber von Linux auf Fiasco und stellt somit bereits für viele Netzwerkkarten Treiber zur Verfügung. In der Arbeit „Netzwerktransparente IPC für L4/Fiasco“ [4] wurde ein Prototyp für netzwerktransparente IPC implementiert.



lokaler Fall



verteilter Fall

Abbildung 12

Mit meinen Arbeitsergebnissen kann Prozessmigration in einem verteilten System gut realisiert werden. Dazu startet man für jeden Thread einen Stellvertreter auf jedem Rechner im Netzwerk. Auf diese Weise kann man Prozesse (und damit Threads) beliebig migrieren, es

muss nur allen Stellvertreter-Threads bekannt sein, auf welchem Rechner sich der originale (zu überwachende) Thread befindet. Alle anderen Threads benötigen darüber keine Kenntnis. Im Sender-Empfänger-Beispiel würde die Kommunikationsweiterleitung wie folgt ablaufen: Der Sender kennt nur den lokalen Empfänger-Stellvertreter und schickt diesem die IPC (wie im lokalen Fall). Der Empfänger-Stellvertreter muss nun wissen, auf welchem Rechner sich der wirkliche Empfänger befindet. An den Sender-Stellvertreter auf diesem Rechner wird die IPC weitergeleitet. Dieser Sender-Stellvertreter stellt nun dem Empfänger die IPC zu (wieder analog dem lokalen Fall). Wichtig ist hier, dass der Sender seine IPC immer an den lokalen Empfänger-Stellvertreter sendet, und der Empfänger immer die IPC vom lokalen Sender-Stellvertreter empfängt. Die Threads einer Anwendung brauchen also nicht über die Verteilung und somit über die Migration Bescheid wissen.

In diesem Beispiel muss der Empfänger-Stellvertreter alle Sender-Stellvertreter kennen. Da der Sender und der Empfänger nicht die einzigen beiden Threads im System sind, die miteinander kommunizieren, müsste jeder Stellvertreter-Thread jeden anderen Stellvertreter-Thread auf jedem Rechner kennen. Das Problem lässt sich lösen, indem der Empfänger-Stellvertreter einen Broadcast an alle Empfänger-Stellvertreter sendet. Beim Erhalt solch eines Broadcasts überprüfen die Empfänger-Stellvertreter, ob der Empfänger auf ihrem Rechner (also lokal) läuft. Wenn das der Fall ist, so wird die IPC an den lokalen Sender-Stellvertreter weitergeleitet, ansonsten wird sie verworfen. Der Empfänger-Stellvertreter muss also nur Kenntnis von allen anderen Empfänger-Stellvertretern haben. Und der Empfänger-Stellvertreter muss nicht wissen, auf welchem Rechner sich der Empfänger befindet, sondern nur, ob er sich auf dem gleichen Rechner befindet. Diese Lösung sollte allerdings nur eingesetzt werden, wenn ein broadcast-fähiges Netzwerk vorhanden ist.

6.2 Sicherheits- und Testanwendungen, Protokollierung

Des Weiteren kann man die hier beschriebenen Ergebnisse ohne größere Änderungen für Sicherheitsanwendungen einsetzen. Dazu müssen lediglich die Stellvertreter-Threads so modifiziert werden, dass sie nicht nur IPC weiterleiten, sondern diese auch überprüfen und gegebenenfalls verwerfen.

In Sicherheitsanwendungen könnte es zu einem Problem werden, dass vertrauensunwürdige Komponenten IDs erraten und dann unüberwacht kommunizieren können. Diese Sicherheitslücke ist aber nicht durch meine Arbeit, sondern durch die L4V2-Spezifikation entstanden. Um diese Lücke zu schließen, wären Kernänderungen nötig, so dass nicht mehr jeder Thread mit jedem Thread kommunizieren kann. Nach dieser Kernänderung würde es sich allerdings nicht mehr um eine V2-Kernschnittstelle handeln.

Die Kommunikationsüberwachung eignet sich auch gut zum Testen. Man kann gezielt die Kommunikation einer Komponente überwachen und somit die korrekte Funktionsweise überprüfen.

Außerdem kann man IPC sehr einfach protokollieren, indem man die Stellvertreter-Threads derart modifiziert, dass sie in einer Datei speichern, welche IPC sie weiterleiten. Diese Protokollierung eignet sich besonders gut im Zusammenhang mit Sicherheits- und Testanwendungen.

6.3 Messung der Verteilung der Antwortzeit

Speziell für den Einsatz in Echtzeitsystemen kann man meine Ergebnisse auch erweitern, um die Antwortzeiten-Verteilung von Servern zu bestimmen. Dazu verändert man den Client-Stellvertreter so, dass er beim Weiterleiten einer Anfrage den Zeitstempel des Prozessors liest und speichert. Beim Weiterleiten der Antwort wird nun erneut der Zeitstempel des Prozessors ausgelesen, daraus die Antwortzeit ermittelt und protokolliert. Nach längerer Laufzeit erhält man so die Verteilung der Antwortzeit des überwachten Servers.

Der Vorteil dieser Lösung ist, dass man die Verteilung der Antwortzeit eines Servers nicht separat messen muss, sondern die Messergebnisse im laufenden Betrieb des Servers mit anfallen. Außerdem braucht man nicht möglichst realistische Testfälle zu planen und zu simulieren, da man bereits in der Realität misst.

Der Nachteil dieser Lösung ist der Einsatz in einer Echtzeitumgebung an sich. Dazu muss der Proxy genau ausgemessen werden. Schließlich muss dessen Rechenzeit mit eingeplant werden. Außerdem können Clients ihre Rechenzeit den Servern zur Verfügung stellen, die ihre Anfrage bearbeiten. Dies kann an Prioritäten gebunden sein. Auch wird die Vererbung der Rechenzeit in vielen Echtzeitsystemen bei der Einplanung mit beachtet. Wenn man nun Kommunikation überwacht, also 2 Stellvertreter-Threads zwischen Client und Server schaltet, muss sichergestellt werden, dass diese Vererbung der Rechenzeit (und deren korrekte Einplanung) nicht gestört wird.

6.4 Fehlerprotokoll

In Kapitel 3.2 wurde bereits dargelegt, dass eine Kommunikationsüberwachung mittels Stellvertreter-Threads zum Verlust der Synchronität führt. Dadurch lassen sich Fehler beim Zustellen einer IPC an den Empfänger nicht mehr an den Sender ausliefern. Eine Möglichkeit, dies zu verhindern, ist, die IPC nicht einfach weiterzuleiten, sondern ein geeignetes Kommunikationsprotokoll zu verwenden. Die Verwendung eines Fehlerprotokolls führt jedoch dazu, dass die zu überwachenden Anwendungen leicht verändert werden müssen. Um diese Änderungen möglichst gering zu halten, sollte man unterscheiden, ob es sich um eine Sender-Empfänger-Kommunikation oder um eine Client-Server-Kommunikation handelt.

Sender-Empfänger-Kommunikation

Die einfachste Möglichkeit besteht darin, jede weitergeleitete IPC zu quittieren. Für das Versenden einer Nachricht sieht das Szenario wie folgt aus: Der Sender verschickt eine Nachricht an den Empfänger-Stellvertreter und wartet auf die Quittung. Der Empfänger-Stellvertreter leitet die Nachricht an den Sender-Stellvertreter weiter und wartet seinerseits auf eine Quittung. Der Sender-Stellvertreter stellt die Nachricht nun dem Empfänger endgültig zu. Dieser quittiert den Erhalt der Nachricht. Der Sender-Stellvertreter erhält also im Fall der erfolgreichen Zustellung der Nachricht eine Quittung, ansonsten erhält er einen Fehlercode. In beiden Fällen kann er eine entsprechende Quittung an den Empfänger-Stellvertreter weiterleiten, welcher seinerseits die Quittung an den Sender weiterleitet.

Die zu überwachende Anwendung muss dabei lediglich wie folgt modifiziert werden: Der Sender muss auf eine Quittung warten und der Empfänger muss eine Quittung versenden. Nachdem diese Modifikation an der Anwendung (und an den Proxys) vorgenommen wurde, ist der Einsatz der Proxys wieder transparent: Entweder die Kommunikation wird überwacht und das oben beschriebene Szenario tritt auf oder die Kommunikation wird nicht überwacht.

In dem Fall schickt der Empfänger die Quittung direkt an den Sender. Durch das Fehlerprotokoll verdoppelt sich die Anzahl der übertragenen IPC.

Client-Server-Kommunikation

Die Client-Server-Kommunikation besteht aus einer Anfrage vom Client an den Server und aus einer Antwort vom Server an den Client. Damit kann man eine Client-Server-Kommunikation auf zwei Sender-Empfänger-Kommunikationen abbilden. Allerdings sind dann für eine Client-Server-Kommunikation 12 IPC notwendig und der Server darf seine Anfrage nicht mit einem Null-Timeout versenden. Ansonsten könnte (unter der Annahme, der Server besitzt eine mindestens genauso hohe Priorität wie der Client-Stellvertreter) folgende Situation auftreten: Der Server erhält die Anfrage und versendet eine Quittung an den Client-Stellvertreter. Bevor der Client-Stellvertreter diese Quittung weiterleitet, ist der Server bereits mit der Berechnung fertig und versendet das Ergebnis mit einem Null-Timeout an den Client-Stellvertreter. Dieser ist aber noch nicht empfangsbereit und die IPC wird mit einem Timeout-Fehler unterbrochen. Daraufhin verwirft der Server das Ergebnis und wartet auf die nächste Anfrage.

Man kann das Fehlerprotokoll noch verbessern, indem man es an die Client-Server-Kommunikation anpasst und nur die Anfrage quittiert. Das Versenden des Ergebnisses braucht nicht quittiert werden, da dem Server egal ist, ob der Client einen genügend großen Empfangspuffer zur Verfügung stellt. Ist dies nicht der Fall, so erhält der Client ohnehin eine Fehlermeldung und kann die Anfrage noch einmal wiederholen. Dieses angepasste Fehlerprotokoll benötigt nun nur noch 9 IPC pro Client-Server-Kommunikation und nach dem Ändern der Anwendung ist der Einsatz der Proxys wieder transparent.

Wenn es bei einem Serverabsturz für den Client keinen Unterschied darstellt, ob der Server vor dem Erhalt oder während der Bearbeitung der Anfrage abgestürzt ist, so kann auch die separate Quittung der Anfrage eingespart werden. Diese kann der Server an die Antwort anhängen. Somit sind für eine Client-Server-Kommunikation nur noch 6 IPC notwendig.

Ergebnis

Wird überwachte Kommunikation quittiert, um Fehler auch an den Sender zu übertragen, so muss man die Anwendung ändern. Das folgt schon zwangsläufig aus der Tatsache, dass der Sender seine Nachricht verschickt und nach dieser Kommunikationsbeziehung noch einmal auf eine Quittung wartet. Die Überwachung der Kommunikation ist also nicht mehr transparent. Man muss die Anwendung in der Hinsicht modifizieren, dass der Empfänger eine Quittung an den Sender verschickt bzw. der Server eine Quittung an den Client verschickt. Außerdem darf der Server nicht mehr mit einem Null-Timeout antworten.

Diese Änderungen betreffen aber fast ausschließlich die zu überwachende Anwendung und ihre Clients. Die einzige Modifikation an den Proxys besteht in der Fehlerausgabe: Tritt bei der Übertragung einer IPC ein Fehler auf, so wird dieser nicht ignoriert oder auf dem Bildschirm ausgegeben, sondern es wird eine IPC mit einem Fehlercode zurückgesendet. Für die weitere Übertragung der IPC mit dem Fehlercode bzw. für die Übertragung einer Quittung an den Sender bzw. Client ist keinerlei Modifikation der Proxys notwendig.

6.5 Zusammenlegung von Client- und Server-Proxy

Die Nachrichtenübertragung zwischen dem Client- und dem Server-Proxy kann noch verbessert werden, falls es sich dabei nicht um eine Netzwerkübertragung handelt. Bei jeder überwachten Kommunikation findet eine IPC zwischen zwei Stellvertreter-Threads statt. Dabei gehört ein Stellvertreter stets zum Client-Proxy und ein Stellvertreter zum Server-Proxy. Werden nun der Client- und der Server-Proxy in einer L4-Task zusammenfasst, könnten alle Stellvertreter-Threads alle Nachrichten-Puffer gemeinsam verwenden und somit ein Kopieren vermeiden. Dadurch lässt sich auch die Übertragungszeit etwas senken. Allerdings muss der Proxy dann beachten, welche Puffer gerade durch welche Threads belegt sind und ab wann die Puffer wieder verwendet werden können. In der momentanen Implementation ist das kein Problem, da jeder Stellvertreter-Thread seinen eigenen Puffer besitzt.

Durch das Zusammenlegen von Client- und Server-Proxy in eine Task wird sich aber kein Stellvertreter-Thread und auch keine IPC einsparen lassen. Die Gründe hierfür wurden in Kapitel 3.4 bereits dargelegt. Auch eine Unterscheidung zwischen Client-Proxy und Server-Proxy sollte weiterhin möglich bleiben. So könnte man die Proxys weiterhin unterschiedlich konfigurieren, was bei vielen möglichen Einsatzszenarien erforderlich ist.

Des Weiteren ist der Einsatz des von Rene Reusner in seinem Großen Beleg [5] entwickelten Local-IPC hier nicht sinnvoll, da die Dauer der Nachrichtenübertragung erst bei mindestens 3-4 aufeinanderfolgenden Local-IPCs verkürzt wird.

6.6 Multithreaded Server

Als Multithreaded Server bezeichnet man Server, die ihre Arbeit nicht – wie bisher beschrieben – mit nur einem Thread verrichten, sondern die mehrere Threads verwenden. Dazu registriert sich ein Thread als Server. Nun können Clients diese ID beim Nameserver erfragen und richten an diesen Thread ihre Anfragen. Der registrierte Server-Thread beantwortet jedoch nicht die Anfrage, sondern teilt dem Client die ID eines anderen Server-Threads mit. Im weiteren Verlauf stellt der Client nun alle seine Anfragen an diesem Server-Thread. Dadurch verteilt sich die Last auf alle Server-Threads.

Multithreaded Servers entsprechen ebenfalls nicht dem in Kapitel 3.1 beschriebenem Szenario der Client-Server-Kommunikation. Sie können auch nicht mit generischen Proxys überwacht werden. Wenn der Client die ID eines anderen Server-Threads erhält, so kommuniziert der Client mit diesem Thread. Er wird also gar nicht mehr mit dem Proxy-Framework kommuniziert, so dass auch im Nachhinein Stellvertreter-Threads nicht angelegt werden können.

Das Problem der Überwachung von Multithreaded Servern ist nur mit anwendungsspezifischen Proxys zu lösen. Die Proxys müssen das Protokoll kennen, mit dem die ID des zukünftigen Server-Threads übertragen wird. Dann kann die ID ausgelesen werden und für diesen Server-Thread kann (bei Bedarf) ein Server-Stellvertreter angelegt werden. Dem Client wird die ID des Server-Stellvertreters mitgeteilt.

6.7 L4-Sec

L4-Sec ist eine neue Kernschnittstelle von L4. Sie wird derzeit entwickelt und kann sich noch entsprechend ändern. Ich beziehe mich hier auf den Stand vom 22.02.2005.

Meine Arbeit baut auf der Kernschnittstelle L4V2 auf. Ich werde im Folgenden beschreiben, mit welchen Änderungen und Konsequenzen man die generischen Proxys auf L4-Sec portieren kann.

Probleme beim Client-Server-Szenario

Die beiden wesentlichen Unterschiede zwischen L4V2 und L4-Sec sind die Einführung von Capabilitys (= Rechte) und Kommunikationsendpunkten. Threads können nicht mehr eine IPC an beliebige Threads schicken, sondern es können beliebig viele Threads eine IPC an einen Endpunkt schicken. Von diesem Endpunkt können beliebig viele Threads empfangen. Wenn mehrere Threads sende- oder empfangsbereit sind, so ist es dem Betriebssystem überlassen, welcher als nächster kommunizieren darf. Für das Senden und das Empfangen sind Capabilitys notwendig.

Daraus folgt nun, dass eine Client-Server-Kommunikation in der Art, wie sie in diesem Beleg beschrieben wurde und auf L4V2 üblich war, nicht mehr zu realisieren ist. Der Server hatte sich beim Nameserver registriert und dann auf Anfragen gewartet. Beim Eintreffen einer Anfrage erhielt der Server die ID des Clients, an die er die Antwort später zurücksenden konnte. In L4-Sec würde sich der Server genauso beim Nameserver registrieren – allerdings nicht seine ID hinterlassen, sondern den Endpunkt angeben, von dem er Anfragen empfängt. Nun könnte der Client diesen Endpunkt beim Nameserver erfragen und die Anfrage an diesen stellen. Allerdings empfängt der Server die Anfrage, kann sie bearbeiten, weiß aber nicht, an welchen Endpunkt er die Antwort schicken soll. Wenn er sie in Analogie zum L4V2-Szenario an den Endpunkt schickt, von dem er die Anfrage empfangen hat, dann sendet der Server die Antwort an den Endpunkt, von dem er selbst Anfragen empfängt.

Badges

Um eine Nachricht an einen Endpunkt zu versenden, benötigt ein Thread ein Senderecht zu diesem Endpunkt. Beim Senden gibt der Thread dieses Senderecht mit einer CapabilityID an, außerdem spezifiziert er noch einen Timeout und einen SourceIdentifier. Dieser SourceIdentifier ist eine frei wählbare Zahl.

Zu jedem Senderecht an einen Endpunkt gehört auch ein Badge; das ist ein Bitstring, mit dem der SourceIdentifier vom höherwertigsten Bit aus überschrieben wird. Das bedeutet, wenn das Badge genauso lang wie der SourceIdentifier ist, so wird dieser durch das Badge ersetzt. Wenn das Badge nur halb so lang wie der SourceIdentifier ist, so wird nur die höherwertige Hälfte überschrieben. Die niederwertige Hälfte könnte der sendende Thread frei wählen. Der (mit dem Badge überschriebene) SourceIdentifier wird beim Empfangen der IPC als Rückgabewert an den Empfänger übermittelt.

Das Badge selbst und das Überschreiben des SourceIdentifier mit dem Badge ist durch das Betriebssystem geschützt, die Anwendung kann daran also nichts ändern. Wenn die Anwendung allerdings das Senderecht an einen anderen Thread weitergibt, so kann sie dem Badge weitere Bit-Stellen hinzufügen.

Mit Hilfe der Badges kann man nun ein Client-Server-Szenario für L4-Sec entwickeln: Der Server registriert seinen Empfangs-Endpunkt beim Nameserver. Dabei erhält der Nameserver ein Senderecht zu diesem Endpunkt mit leerem Badge.

Wenn der Client beim Nameserver nach einem bestimmten Server fragt, so erhält der Client eine Kopie des Senderechtes an den Endpunkt des Servers. Allerdings fügt der Nameserver ein komplettes Badge hinzu, welches dem Empfangsendpunkt des Clients entspricht.

Wenn der Client eine Anfrage an den Server stellt, wird auf diese Weise der SourceIdentifier automatisch mit dem Badge überschrieben. Das bedeutet, der SourceIdentifier (den der Server empfängt) spezifiziert den Empfangsendpunkt des Clients, an welchen die Antwort gesendet werden kann.

Kommunikationsüberwachung

Dieses Client-Server-Szenario kann nun ähnlich überwacht werden, wie in meiner bisherigen Arbeit beschrieben: Der Server registriert sich beim Nameserver, daraufhin wird ein Server-Stellvertreter angelegt. Beim Nameserver wird der Endpunkt des Server-Stellvertreters gespeichert. Sobald der Client einen Endpunkt beim Nameserver erfragt, wird für diesen Client auch ein Stellvertreter erzeugt.

Die Proxys speichern auf diese Weise nicht mehr ID-Paare, sondern die Zuordnung SourceIdentifier(Client)⇒Endpunkt(Client-Stellvertreter). Bei der Weiterleitung der IPC müssten die Stellvertreter lediglich den SourceIdentifier ändern, schließlich soll der Server ja nicht dem Client, sondern dem Client-Stellvertreter antworten. Auf diese Weise kann die Kommunikation im Wesentlichen wie bisher überwacht werden.

Da es nicht möglich ist, von mehreren Endpunkten gleichzeitig zu empfangen, kann man die Anzahl der Stellvertreter leider nicht reduzieren. Ansonsten müsste man die Endpunkte wie oben beschrieben erzeugen, bräuchte aber nur einen Thread, der alle Endpunkte abhört und eintreffende IPC entsprechend weiterleitet.

Multithreaded Server

In L4-Sec ist es sehr einfach, Multithreaded Server zu realisieren. Es ist nicht mehr nötig, wie in Kapitel 6.6 beschrieben, über ein Protokoll IDs auszutauschen. Da mehrere Threads von einem Endpunkt empfangen können, braucht man nur den Server-Thread vervielfältigen. Somit empfangen alle Server-Threads von einem (dem im Nameserver registrierten) Endpunkt. Trifft eine Anfrage ein, so kann das Betriebssystem einem der wartenden Server-Threads diese Anfrage zustellen.

So realisierte Multithreaded Server können ohne Änderung von den beschriebenen Proxys überwacht werden, denn es muss nur ein Endpunkt überwacht werden. Wie viele Threads von diesem Endpunkt empfangen, ist dabei unerheblich.

7 Zusammenfassung

Die Kommunikation dynamisch verteilter Anwendungen mit Hilfe von generischen Proxys ist möglich, solange keine Multithreaded Server beteiligt sind. In dieser Arbeit wurden verschiedene Konzepte gezeigt, Kommunikation zu überwachen. Eines dieser Konzepte wurde ohne Kernänderung implementiert und ist somit direkt auf Fiasco lauffähig.

Wenn man Kernänderungen vermeiden will, so kann man nicht jede IPC übertragen; es müssen Kompromisse in Bezug auf die Nachrichtengröße eingegangen werden. Man kann jede mögliche Short- und Long-IPC mit generischen Proxys übertragen, wenn man nicht mehr als 31 Stellvertreter pro Task erzeugt. Wenn mehr Stellvertreter benötigt werden, so kann man die Stellvertreter eines Proxys auch auf mehrere Tasks aufteilen. Bei der Übertragung einer Flexpage-IPC ist eine Limitierung der Größe unbedingt notwendig. Dies wurde in Kapitel 4.1 gezeigt.

Falls man Kernänderungen akzeptiert, so ist es möglich, IPC beliebiger Größe zu übertragen. Dazu kann die IPC bereits im Adressraum des Senders ausgewertet werden und anschließend direkt dem Sender zugestellt werden.

Multithreaded Server können von generischen Proxys nicht ohne Kernänderung überwacht werden. Wenn man Kernänderungen erlaubt, so kann man nicht nur einen Thread, sondern automatisch alle Threads in der gleichen Task überwachen.

Wenn man auf Kernänderungen verzichtet, dann hat ein Proxy nicht das Recht, IPC zu modifizieren. Dadurch ist der Proxy darauf angewiesen, dass die Anwendung „von sich aus“ mit dem Proxy kommuniziert. Das ist bei Servern, welche aus einem Thread bestehen, mit der Änderung des Nameservers möglich, nicht aber bei Multithreaded Servern. Hier kann eine Überwachung der IPC nur durch anwendungsspezifische Proxys, wie in Kapitel 4.4 beschrieben, durchgeführt werden.

Nutzt man dagegen die Kernschnittstelle L4-Sec, so kann man nicht mehr Threads, sondern Kommunikationsendpunkte überwachen. Somit ist es dann möglich, Multithreaded Server auch ohne Kernänderung zu überwachen.

8 Literatur

[1] Jochen Liedke: „L4 Reference Manual“
September 1996

[2] Lehrstuhl für Betriebssysteme: „L4Env – An Environment for L4 Applications“
TU Dresden, Fakultät Informatik, Juni 2003

[3] Jork Löser: „Building Infrastructure for DROPS (BID) Tutorial“
TU Dresden, Fakultät Informatik, Juni 2003

[4] Sebastian Lehmann: „Netzwerktransparente IPC für L4/Fiasco“
TU Dresden, Fakultät Informatik, Diplom August 2002

[5] René Reusner: „Implementierung von Local-IPC auf L4/Fiasco“
TU Dresden, Fakultät Informatik, Großer Beleg November 2004

[6] T. Jaeger, J.E. Tidswell, A. Gefflaut, Y. Park, J. Liedtke, K. Elphinstone:
„Synchronous IPC over Transparent Monitors“
Proceedings of the Ninth ACM/SIGOPS European Workshop, 2000

[7] Jan Glauber: „Checkpointing als Basis für transparente Service-Migration unter Linux”
TU Dresden, Fakultät Informatik, Diplom Januar 2002