

VAM und VX-Amoeba - die virtuelle AMOEBA Maschine und eine neue hybride Betriebssystemumgebung

Dr. rer. nat. Stefan Bosse



14.9.2004

Zusammenfassung

In diesem Artikel soll eine neue hybride Betriebssystemumgebung vorgestellt werden, die auf dem bekannten verteilten Betriebssystem AMOEBA basiert. Die verteilte Kommunikationsumgebung besteht aus zwei Kernkomponenten: dem neuen AMOEBA Kernel VERTEX als "Hardcore"-System, und die virtuelle AMOEBA Maschine VAM, die AMOEBA-Konzepte in bestehende Betriebssysteme, wie z.B. UNIX, integriert. Das Hybrid-System zeichnet sich durch eine hohe Skalierbarkeit, Flexibilität und Portabilität aus.

Schlüsselwörter

Verteilte Betriebssysteme, Netzwerkprotokolle, Virtuelle Maschinen, Funktionale Programmierung

1. Einführung

1.1. AMOEBA

Anfang der 80' Jahre wurde von der Amsterdamer Forschergruppe um Prof. Andrew Tanenbaum das über ein Netzwerk verteilte Betriebssystem AMOEBA zum Leben erweckt [1]. Völlig neue Konzepte und ein für diese Zeit revolutionär einfaches Design machen die Stärken dieses Systems aus. Zentraler Bestandteil dieses Systems ist ein neu entwickeltes Netzwerkprotokoll namens **FLIP** (Fast Local Network Protocol), welches sich u.A. durch optimierten Datendurchsatz und geringe Latenz auszeichnet. Der FLIP-Protokollstack bildet die Grundlage für AMOEBA's **RPC** (Remote Procedure Call) Kommunikationsprotokoll. Praktisch alle Systemfunktionen werden über das vereinheitlichte RPC-Protokoll abgewickelt, wie z.B.

- Dateidienste,
- Prozessmanagement,

- Terminal I/O,
- Veränderung von Serverparametern, Abfragen von Statusinformationen,
- entferntes Laden eines Kernels.

AMOEBA weist einige Parallelen zum verteilten Betriebssystem Plan9 aus den Bell-Labs auf. Jedoch baute Plan9 auf Unix-Konzepten auf, wo hingegen AMOEBA völlig unbeachtet bestehender Betriebssysteme entwickelt wurde.

Wie eingangs erwähnt, bildet das RPC-Protokoll die zentrale Kommunikationsschnittstelle. Das in AMOEBA verwendete RPC-System arbeitet synchron, d.h. eine Nachricht, die von einem Klienten an einen Server gesendet wird, bedarf einer Rückantwort vom Server. Der Klient (bzw. der rufende Thread) bleibt derweilen bis zum Eintreffen der Antwort oder dem Fehlschlagen des RPC blockiert. Das RPC-Interface besteht aus nur drei Funktionen:

```

errstat trans(hdr_req,buf_req,bufsize,hdr_rep,buf_rep,bufsize');
errstat getreq(hdr_req,buf_req,bufsize);
errstat putrep(hdr_rep,buf_rep,bufsize');

```

Die erste Funktion wird vom Klienten, und die beiden letzten werden vom Server, immer paarweise, verwendet. Zu jeder Transaktion gehört ein Transaktionsformular, genannt **Header**. Es wird jeweils eines für den Hin-, und eines für den Rückweg verwendet. Der Header hat folgendes Format:

port h_port
port h_signature
private h_priv
uintn16 h_command/h_status
uint32 h_offset
uint16 h_size
uint16 h_extra

Der Header beinhaltet neben drei vom Nutzer frei wählbaren Einträgen (h_offset, h_size, h_extra, insgesamt 64 Bit) die Zieladresse in Form eines sog. Ports. Ein **Port** besteht aus einer Anzahl von Bytes, in der symbolischen Form geschrieben als:

```
p1:p2:p3:p4:p5:p6
```

Es können eine beliebige Anzahl von Servern auf dem gleichen Port Nachrichten empfangen, sowohl auf dem lokalen Rechner (Multithreading!), als auch auf anderen Rechnern. Welcher Server(thread) eine Nachricht erhält, ist nicht zwangsläufig deterministisch.

Es wird zwischen **privaten und öffentlichen Ports** unterschieden. Der Grund liegt in der notwendigen Unterscheidung zwischen Ports, mit denen ein Server sein Kommunikationsinteresse bekundet, und Klienten, die Kontakt mit dem Server aufnehmen wollen. Würden die sog. get- und putports identisch sein, könnte jeder Klient auch einen Server immitieren, mit fatalen Sicherheitsfolgen.

Zu diesem Zweck wird der eigentliche Serverport (i.A. zufallsgeneriert) mittels der Einwegfunktion `priv2pub` verschlüsselt und veröffentlicht, z.B. im Dateisystem. Dazu später mehr.

Neben dem Header, der noch weitere Informationen zum Auftrag erhält, gibt es einen Transaktionsdatenbereich. Es gibt jeweils einen für den Datentransfer zum Server, und einen für die Rückantwort vom Server zum Klienten (diese können aber auch entfallen - der Header enthält einige frei zur Verfügung stehende Einträge).

Die Aufgabe eines Servers besteht i.A. darin, Objekte zu verwalten und Klienten zur Verfügung zu stellen. Bestes Beispiel ist der Dateiserver, deren Dateien als Objekte ausgetauscht werden. Weitere Beispiele für Objekte in AMOEBA sind:

- Verzeichnisse
- Prozesse
- Speichersegmente
- Terminal Ein- und Ausgabekanäle
- Unix-Pipes usw.

Jedes Objekt ist mit einer sog. **Capability** verknüpft, die aus dem oben beschriebenen Serverport (public), einer Objektnummer, Objektrechten und einem privaten Port, dem Sicherheitsport, besteht. Symbolisch läßt sich eine Capability darstellen durch:

`p1:p2:p3:p4:p5:p6/objnum(rights)/s1:s2:s3:s4:s5:s6`

Zweck des privaten Ports ist die Verifizierung der im Mittelteil der Capability aufgeführten Rechte, die die Nutzung der mit der Capability verknüpften Resource bestimmen (z.B. Dateirechte - Schreiben, Lesen, Löschen...). Das Rechtfeld ist mittels einer Einwegverschlüsselung im privaten Port enthalten. Nur der Server, der den (zufallsgenerierten) Ausgangsport, der bei der Verschlüsselung benutzt wird, kennt, kann mittels des Rechtfelds den öffentlichen Sicherheitsport generieren und mit dem vom Klienten angebotenen vergleichen. Bei Übereinstimmung wird das Rechtfeld als gültig angesehen.

1.2. FLIP

Das FLIP-Protokoll wurde zur effizienten Kommunikation zwischen Prozessen (Server/Klienten) in verteilten Systemen entwickelt. FLIP ist ein zuverlässiges Nachrichtenprotokoll, daß sowohl Punkt-zu-Punkt- als auch Multibezug-Kommunikation erlaubt. Ein wichtiges Merkmal ist dabei, daß praktisch kein Netzwerkmanagement, anders als z.B. bei IP, notwendig ist. [5]

Das für den Programmierer essentielle Kommunikationssystem wird durch die aufgesetzte RPC-Schicht implementiert.

Folgende Eigenschaften sind zu nennen:

- FLIP ist unabhängig von Netzwerktechnologien - sowohl Ethernet als busbasierte Systeme (VME-Bus) können zum Einsatz kommen (sogar serielle Schnittstellen können als Netzwerkpunkte eingesetzt werden).
- FLIP Kommunikationsendpunkte werden durch einheitliche und ortsunabhängige 64-Bit Schlüssel gekennzeichnet.
- FLIP arbeitet gleichzeitig als ein Netzwerkrouter, und kann zwischen verschiedenen Netzwerken, auch hinsichtlich der physikalischen Implementierung, vermitteln. FLIP kann die geeignetste Route ermitteln.
- FLIP routet nachrichtenbasierend mittels des 64-Bit Endpunkt-Bezeichners. Aber FLIP arbeitet verbindungslos.
- Die softwareseitige Implementierung von FLIP konnte vom Autor betriebssystemunabhängig realisiert werden, sodaß nicht nur der ursprüngliche Amoeba-Kernel als Träger in Frage kommt, sondern auch UNIX-Systeme.

1.3. AMOEBA Server

1.4. AMOEBA's Schwächen und Stärken

AMOEBA entwickelte sich im Laufe der Zeit zu einem ausgewachsenen Desktop-Betriebssystem, unter anderem mit einer bibliotheksbasierten Unix-Emulation und der graphischen X11-Umgebung. Jedoch ist der Aufwand hoch, nahezu 4000 freie Programmpakete nach AMOEBA zu portieren, wodurch die Eignung als verteiltes "Allround"-Betriebssystem seine Grenzen fand. Zudem erwies sich der tägliche Betrieb eines verteilten Betriebssystems mit massiver Prozesslastenverteilung als problematisch, wenn z.B. X-Windows auf der lokalen Maschine läuft, das xterm-Programm auf einer anderen Maschine B, die Shell auf Maschine C, und der Texteditor auf Maschine D. Weiterhin sind die Konzepte von AMOEBA's Dateisystem im Vergleich zum Unix-Dateisystem für einen alltäglichen "Allround"-Betrieb nur bedingt geeignet, zumindest wenn man Unix-portierte Programme nutzt.

Die Stärken liegen aber in der RPC-Kommunikation und dem FLIP-Protokoll, welches eine einfach zu programmierende, aber sehr leistungsfähige verteilte Systemumgebung ermöglicht. Zudem ist der AMOEBA-Kernel ein moderner Mikro-Kernel, der vollständig auf dem Klient-Server-Modell beruht.

Die Implementierung neuer Gerätetreiber bereitet im Gegensatz zu monolithischen Kernel keine nennenswerten Schwierigkeiten, und dürfte auch Anfängern einen guten Einstieg in diese Materie ermöglichen. Mit dem neuen **VERTEX VX-AMOEBA** Kernel können Gerätetreiber als normale Prozesse außerhalb des Kernels ausgeführt werden. Es wird lediglich eine spezielle System-Capability benötigt, um I/O Ports, ISR und weitere privilegierte Operationen durchführen zu können. Der Vorteil dieser Methode liegt in einer schnellen und sicheren Implementierung von

Gerätetreibern, mit bekannten und definierten Schnittstellen zum Kernel. Der Kernel bleibt übersichtlich, und verschiedene Gerätetreiber können weder den Kernel negativ beeinflussen, noch andere Prozesse.

Ein AMOEBA-Kernel allein kann autonom operieren, stellt aber noch kein vollständiges System dar. Wesentliche Bestandteile wie das Dateisystem oder der Namens- und Verzeichnisdienst können zwar im Kernel integriert werden, was aber dem Konzept eines Mikro-Kernels widerspricht. Daher wurde nach einer Alternative gesucht, um zum einen ein anwendungsorientiertes verteiltes System mit einer einfachen RPC-Programmierung zu ermöglichen, und zum anderen ein komfortables Desktop-Betriebssystem als zentrales Operationsinstrument und Entwicklungsumgebung nutzen zu können. Insbesondere verteilte Meß- und Steuerungssysteme sollten auf einfache Weise aufgebaut und programmiert werden können.

1.5. Das Hybrid-Konzept

Die Lösung bestand in einem Hybrid-System aus folgenden Komponenten:

1. Dem weiterentwickelten AMOEBA-Kernel namens **VERTEX (VX)** mit folgenden Leistungsmerkmalen und Komponenten (siehe dazu auch [2]):
 - Leistungsfähiger Mikro-Kernel
 - Lowlevel Prozeß- und Threadmanagement
 - Netzwerk: FLIP und RPC (FLIP kann auch immer als Router eingesetzt werden)
 - Eine Vielzahl von Gerätetreibern: Serielle- und parallele Schnittstelle, IDE, Floppy, Netzwerk, ...
 - Gerätetreiber im Nutzerprozeßraum: I/O Zugriff, Interrupt-Handler (ISR), ...
 - Leistungsfähiges lokales Kommunikationsinterface IPC mit Unterstützung von "shared memory"-Segmenten
 - Gerätetreiber werden als Server aufgefaßt, und veröffentlichen ihre Dienste (d.h. die Server-Capability) in einem kerneleigenen Verzeichnisdienst.
2. Einer auf bestehenden Betriebssystemen (Unix) aufgesetzten AMOEBA-Umgebung mit folgenden Bestandteilen:
 - Nutzerprozeß-Implementierung des FLIP-Protokollstacks **flipd**, der die RPC-Kommunikation von Unix aus ermöglicht,
 - eine an die Unix-Umgebung angepasste und reduzierte AMOEBA Bibliothek **AMUNIX**, die Basiskonzepte wie RPC, Capabilities und eine Vielzahl von sog. Klient-Server-Stub Prozeduren beinhaltet,
 - eine virtuelle und interaktive Ausführungsumgebung, das **vam** System, die mit der funktionalen Programmiersprache **OCaML** implementiert wurde, bei der der ML-Compiler betriebs- und prozessorunabhängigen Bytecode erzeugt, der von einer virtuellen Maschinen ausgeführt wird,

- und eine Vielzahl von AMOEBA-Systemprogrammen, die vollständig in ML reimplementiert wurden, und somit sowohl mittels vam unter Unix, aber auch nativ unter AMOEBA ausgeführt werden können. Beispiele:
 - ◇ **AFS**: Atomic-File-System. Ersatz für AMOEBA's bullet Dateiserver.
 - ◇ **DNS**: Directory- and Name-Service. Ersatz für AMOEBA's soap Verzeichnis- und Namensserver.
 - ◇ **vash**: Die Unix-AMOEBA-Integrations-Shell

1.6. Eigenschaften

Folgende Eigenschaften lassen sich mit dem neuen Hybrid-Konzept erzielen:

- Portabilität bezüglich verschiedener Prozessoren. Dies trifft zum einen auf den VERTEX-Kernel zu, da dieser durch sein Mikrokernkonzept leicht auf neue Prozessor- und Systemarchitekturen portiert werden kann, und zum anderen auf die VAM-Umgebung, da diese im wesentlichen prozessor- und systemunabhängig ist.
- Portabilität bezüglich der Ausführungsumgebung von VAM. VAM stellt AMOEBA- und Unix-Konzepte betriebssystemübergreifend zur Verfügung, so daß Applikation, wie z.B. Dateiserver, immer die gleiche Schnittstelle sehen.
- Hohe Skalierbarkeit. Der VERTEX-Kernel hat geringe Anforderungen an die Leistungsfähigkeit eines Systems. Wenige Megabyte RAM-Speicher und Prozessorleistungen ab 10 MIPS lassen eine gute Performance erwarten. Es können praktisch beliebig viele Rechnerknoten verbinden. Entweder nativ oder mittels VAM.
- Modularität. Es werden nur die Komponenten tatsächlich ausgeführt, die für den Betrieb notwendig sind. Ein minimales VAM-System besteht z.B. aus: dem FLIP-Protokollstack Server, dem Datei- und Namensserver, und einigen shells. Es können dynamisch weitere Maschinen in das Gesamtsystem hinzugefügt werden.

In den folgenden Kapiteln sollen die einzelnen Komponenten und Schichten dieses Hybrid-Konzepts näher vorgestellt werden.

2. Der VX-AMOEBA-Kernel

Der VERTEX-AMOEBA-Kernel als zentrale Operationseinheit im AMOEBA-Betriebssystem hat einige entscheidende Eigenschaften gegenüber vielen am Markt etablierten Betriebssystemen:

- Mikrokern-Konzept:
 - ◇ einfache und vereinfachte Interaktion zwischen den Kernelmodulen und Prozessen nach dem Klienten-Server-Modell und Nachrichtenaustausch,
 - ◇ Gerätetreiber können als normale Nutzerprozesse implementiert werden (mit Interruptunterstützung auf await/wakeup Basis),

- ◇ kurze I/O Antwortzeiten,
 - ◇ Kernel- und Prozessthreads mit identischer Schnittstelle,
 - ◇ zweistufiger, prioritätenbasierter Scheduler.
- Hoch performante Klienten-Server-Kommunikation:
 - ◇ Remote Procedure Call (RPC): wird von Prozessen und Kernelmodulen zur lokalen **und** netzwerkübergreifenden Kommunikation zwischen verteilten Prozessen verwendet. Die Kommunikation erfolgt synchroin und nachrichtenbasiert mit dem 'Fast Local Network Protocol' (FLIP). Es können Datenfragmente bis zu einer Größe von 1GByte mit einer einzigen Transaktion übertragen werden.
 - ◇ Local Interprocess Procedure Call (IPC): wird ausschliesslich zur lokalen Kommunikation zwischen im Nutzerraum implementierte Gerätetreiber und Nutzerprozessen verwendet. Ähnlich dem RPC-System aufgebaut, bietet es auf Durchsatz und niedrige Latenz optimierte nachrichtenbasierte Kommunikation. So werden auch geteilte Datensegmente unterstützt, über die verschiedene Prozesse Daten 'austauschen' können.
 - Universeller Protokollstack FLIP:
 - ◇ FLIP ist zentraler Bestandteil der Kommunikation in einem Netzwerkcluster.
 - ◇ FLIP arbeitet nach dem Routerprinzip. Es können verschiedene Netzwerke verbunden werden.
 - ◇ FLIP ist nicht auf die Ethernet-Hardwareschicht beschränkt.
 - ◇ FLIP benötigt keine Konfiguration. Kommunikationsendpunkte werden mittels Broadcasting ermittelt.
 - Skriptausführung mittels STD_EXEC Methode: VM42

Einzelne Kernelsever unterstützen die Ausführung von Forth-Skripten. Ein nützliches Werkzeug für die schnelle Entwicklung von Gerätetreibern und zum Hardwaretest.
 - Kernel Administration:
 - ◇ Entferntes Starten einer Kerneldatei: mittels des RPC-Systems kann ein laufender VERTEX-AMOEBE-Kernel eine neue Kernelbilddatei empfangen, und diese an seiner Stelle ausführen, so daß ein neuer Kernel gestartet werden kann.
 - ◇ Sämtliche Kernel-Statusinformationen inklusive dem Kernel-Log-Puffer können über das Netzwerk via RPC und der KSTAT Methode abgefragt werden.
 - ◇ Kernellaufzeitparameter können mittels der STD_PARAM Methode gesetzt und gelesen werden.
 - Debugging

1. Thread- und Prozeßtracing

Über den Kernel Systemserver läßt sich eine zeitaufgelöste Spur von Thread- und Prozeßereignissen, wie z.B. einem Threadwechsl im Scheduler, aufzeichnen. Mittels einer Ereignismaske lassen sich bestimmte Arten von Ereignissen aus- und einblenden. Die Ereignisspur kann dann remote ausgelesen und analysiert sowie eventuell grafisch aufbereitet werden.

2. FLIP-Netzwerkpakettracing

Sämtlicher Netzwerkverkehr des FLIP Protokolls läßt sich zeitaufgelöst aufzeichnen.

Der VERTEX-Kernel ist zum größten Teil in C programmiert. Der Anteil an Assembler-Code beträgt weniger als 5%. Zur Zeit wird nur die i86-Architektur unterstützt. Die plattformspezifischen Anteile sind vom plattformunabhängigen Teil separiert. Insbesondere der zentrale FLIP Protokollstack wurde überarbeitet und prozessor- und betriebssystemunabhängig gemacht.

2.1. Kernel Scheduler

Der VERTEX-Kernel Scheduler arbeitet als ein zweistufiger prioritätengesteuerter Scheduler. Es gibt sowohl Prozess- als auch Threadprioritäten. Die Prozessprioritäten haben ein höheres Gewicht als Threadprioritäten. Daher kann ein Prozeß seine Threadprioritäten frei wählen. Die Prozesspriorität kann von normalen Prozessen nur verringert werden. Der Kernel wird als Prozeß behandelt, und hat die höchste Priorität. Nur Nutzerprozesse, die erweiterte Rechte (Gerätetreiber) besitzen, können ihre Prozeßpriorität frei wählen.

Es gibt folgende Thread- und Prozeßprioritäten:

1. Prozeßprioritäten:

HIGH Höchstmögliche Priorität. Nur der Kernel und Gerätetreiber im Nutzeraddressraum können diese Priorität annehmen.

NORM Normale Priorität. Standardeinstellung.

LOW Niedrigste Priorität für Hintergrundprozesse.

2. Jeder Prozeß hat seine eigenen Threadprioritäten:

HIGH Höchstmögliche Priorität.

NORM Normale Priorität. Standardeinstellung.

LOW Niedrigste Priorität für Hintergrundprozesse.

Kernelthreads werden grundsätzlich nicht-preemptiv geschaltet. Prozesse werden preemptiv geschaltet, und besitzen ein Zeitquantum. Bei Erreichen des Zeitquantums findet ein Prozeßwechsel statt. Nutzerprozeßthreads können optional preemptiv geschaltet werden.

3. AMUNIX

Amunix besteht aus dem FLIP-Protokoll Daemon **flipd** und einer angepassten AMOEBA Bibliothek **AMUNIX** mit einer dazugehörigen Entwicklungsumgebung. Der FLIP-Daemon hat die Aufgabe, rohe Netzwerkpakete, die dem FLIP-Protokoll zugehören, zu empfangen, zu verarbeiten, der höheren RPC-Schicht weiterzuleiten, sofern diese bestimmungsgemäß sind, und Pakete zu versenden, die im Rahmen der FLIP-Administration und durch RPC-Operationen generiert werden. Der Daemon läuft als privilegierter Prozeß im Unix-Nutzeradressraum. Abbildung 1 zeigt den schematischen Aufbau der bereits vom Originalsystem der Vrije-Universiteit bekannten Amunix-Umgebung. Neu ist aber der Betrieb des FLIP-Daemons als reiner Nutzerprozeß mit dem Vorteil einer hohen Portabilität. Zu diesem Zweck wurde FLIP vollständig neu strukturiert, und alle betriebsystemabhängigen Teile isoliert, so daß für die Unix- und der nativen AMOEBA-Implementierung der gleiche Quellenkern verwendet werden kann. Die einzige betriebsystemabhängige Schnittstelle liegt in der direkten Verbindung zur unteren Netzwerkschicht des Kernels. Bei Linux wird diese über Raw-Sockets, und bei FreeBSD über den Packetfilter BPF realisiert. Als Zukunftsoption ist in der Abbildung schon die Nutzerthread-Implementierung Uthreads eingefügt. Der FLIP-Daemon übernimmt dann zusätzliche Aufgaben im Threadmanagement, da dieses mit der RPC-Schicht Verknüpfungspunkte besitzt.

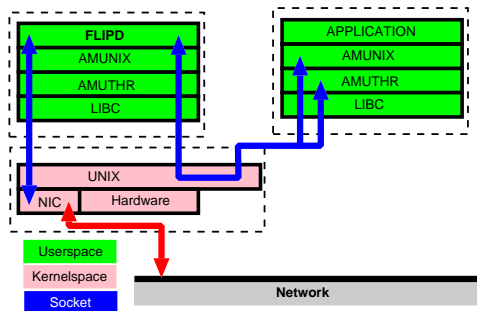


Abbildung 1: Schichtenaufbau bei der Amunix-Umgebung.

Server- und Klientenprogramme, im folgenden Amunix-Programme genannt, kommunizieren in der Amunix-Umgebung mittels Unix-Sockets. Diese Kommunikation bleibt aber durch die RPC-Schicht verborgen. Das Amunix-System kann daher als Translationsschicht zwischen AMOEBA und Unix verstanden werden. Der FLIP-Daemon veröffentlicht in einem festgelegten Unix-Verzeichnis, z.B. `/flip`, einen allgemeinen Kontroll-Socket. Dieser dient der ersten Kontaktaufnahme zwischen Amunix-Programmen, von denen eine Vielzahl gleichzeitig und unabhängig arbeiten können, mit dem FLIP-Daemon. Es wird dann für jeden Thread eines Amunix-Programms ein eigener Socket aufgebaut. Bei einer expliziten Terminierung eines Threads oder implizit durch Prozeßterminierung werden die Socketverbindungen wieder geschlossen. Zusätzlich wird der FLIP-Daemon ein Cleanup durchführen. AMOEBA's Thread-Implementierung ist unter Amunix vollständig nutzbar, und mit der gleichen Schnittstelle ausgestattet.

Da es zunächst noch keinen Verzeichnisdienst gibt, können Capabilities unter Unix als normale Dateien (betriebsystem- und architekturunabhängig) abgelegt und gelesen werden, so daß

einfache Namens-Capability Zuordnungen möglich sind.

Da nur die AMUNIX-Bibliothek in ein Programm eingebunden werden muß, können in einem Programm AMOEBA- und Unix-Konzepte nahtlos ineinander übergehen, genauso wie unter dem nativen AMOEBA-System, nur eben umgekehrt. Bestehende Programme können so auf einfache Weise mit einer verteilten Kommunikationsumgebung ergänzt werden.

3.1. AMUTHR-Implementierung

Um unter UNIX Amoeba-Programme mittels der AMUNIX-Umgebung zum Leben zu erwecken, wird ein geeignetes Threadpaket benötigt, was aber im Gegensatz zum nativen Vertex-Kernel nicht im Kernel, sondern Bestandteil eines jeden AMUNIX-Prozesses ist. Um möglichst gleiches Verhalten wie bei nativen Amoeba-Programmen zu erzielen, wurde kurzerhand das Kernel-threadmodul in den UNIX-Nutzerprozeßraum portiert. Lediglich die Teile des Prozessmanagements wurden entfernt. Ansonsten blieb der Quellcode unverändert. Im Vergleich zu früher verwendeten Posix-Threadpaketen konnte eine signifikante Leistungssteigerung im Zusammenhang mit dem FLIP-System sowohl hinsichtlich des Datendurchsatzes als auch der Latenz erreicht werden (etwa zwei Größenordnungen!).

3.2. FLIP-Implementierung

Im Gegensatz zum nativen Vertex-Kernel wird unter UNIX der FLIP-Protokollstack als reiner Nutzerprozeß implementiert.

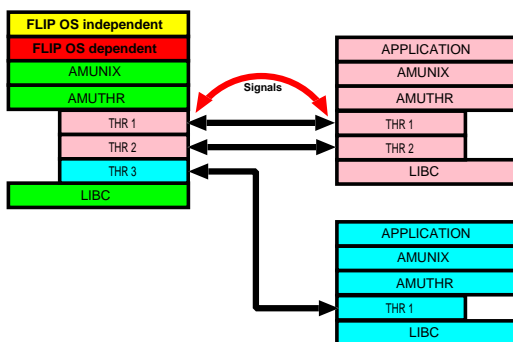


Abbildung 2: Verknüpfung von FLIP mit AMUNIX-Prozessen.

Abbildung 2 zeigt die Anbindung von AMUNIX-Prozessen an den FLIP-Daemon. Jeder AMUNIX-Prozessthread ist (sofern er das RPC-System nutzt) über eine UNIX -Socket-Verbindung mit einem eigenen Thread im FLIPD-Prozess angebunden. Über diese Socket-Verbindung wird mittels Nachrichtenaustausches eine RPC-Operation im FLIPD-Prozess durchgeführt. Neben der reinen RPC-Kommunikation müssen zusätzlich RPC-Threadsignale (SIG_INT) von FLIPD an den AMUNIX-Prozess übertragen werden, und dort in geeigneter Weise durch AMUTHR umgesetzt werden. Beim umgekehrten Weg, wenn z.B. der AMUNIX-Prozess ein UNIX-Signal wie SIG_TERM erhält, muß FLIPD seinerseits RPC-Signale an anderer Prozesse ver-

senden, die mit dem signalisierten AMUNIX-Prozess gerade eine RPC-Transaktion durchführen (siehe auch [6], Kapitel Threadsignale).

4. AMCROSS

Neben der AMUNIX-Entwicklungsumgebung gibt es noch eine sogenannte Crosscompiling-Entwicklungsumgebung, **amcross** genannt, mit der native AMOEBA-Programme, Bibliotheken und Kernel unter Unix übersetzt, gebunden und in das native AMOEBA-Ausführungsformat konvertiert werden können.

5. VAM und ML

Da es bei der Wahl des Unix-Betriebssystems und der Rechnerarchitektur keine Einschränkungen geben sollte, erweist sich die AMUNIX-Entwicklungsumgebung als zu unflexibel, da für jedes Hostbetriebssystem und jede Rechnerarchitektur getrennte Binärdateien erstellt werden müssen. Bei der amcross-Entwicklungsumgebung kommt hingegen dieser Umstand nicht zum tragen, da der VERTEX-AMOEBA-Kernel zur Zeit nur für die i86-Architektur zur Verfügung steht.

Der Einsatz einer **virtuellen Maschine**, die sowohl mit der AMUNIX-Umgebung arbeitet, als auch direkt vom nativen AMOEBA-Kernel ausgeführt werden kann, wurde daher in Erwägung gezogen.

Der häufig verwendete JAVA-Ansatz wurde aus Gründen der relativ niedrigen Leistungsfähigkeit und unverhältnismäßig hohen Komplexität sowie durch Portabilitätsaspekte verworfen. Als vielversprechender Kandidat kristallisierte sich das **OCaML-Projekt** [3] vom französischen INRIA-Institut heraus.

Das OCaML-System, ein Dialekt der funktionalen Programmiersprache ML, besteht aus folgenden Einzelkomponenten [4]:

- Dem **schnellen und optimierenden Compiler**, der das ML-Programm in betriebssystem- und architekturunabhängigen Bytecode übersetzt, selber in ML programmiert,
- der eigentlichen **virtuellen Maschine**, die den Bytecode ausführt, basierend auf einer Stackmaschine, und in C programmiert, im folgenden mit VM abgekürzt,
- und einer **flexiblen interaktiven Entwicklungsumgebung** (shell), die dem Nutzer einen Interpreter vorspiegelt, tatsächlich aber Skripte und Befehlszeilenkommandos direkt in Bytecode übersetzt (beinhaltet daher den Compiler), und, da selber wieder als Bytecodeprogramm ausgeführt, zum Hauptprogramm hinzufügt und ausführt.

Das 'O' im Name steht für eine objektorientierte Ergänzung des ML-Kerns.

Die virtuelle Maschine ist auf die ML-Programmiersprache abgestimmt, und erreicht im Zusammenspiel mit dem optimierenden Compiler eine herausragende Performance. Nach eigenen Erfahrungen kann man davon ausgehen, daß ein CaML-Programm etwa um den Faktor 3-5

langsamer läuft, bzw. um diesen Faktor mehr Rechenleistung benötigt, als ein vergleichbares optimiertes C-Programm.

Im Gegensatz zu C-Programmen ist bei OCaml der Programmierer nicht für das Speichermanagement verantwortlich. Stattdessen wird das Speichermanagement durch die virtuelle Maschine organisiert. Um nicht benutzten Speicher wieder freigeben zu können, läuft im "Hintergrund" der sogenannte "Garbage-Collector", der mittels eines ausgeklügelten Algorithmuses, der unterschiedliche Bearbeitungsstufen- und tiefen kennt, retardiert Speicher freigibt.

Die virtuelle Maschine kann durch eine einfache ML-C-Schnittstelle erweitert werden, und z.B. Funktionen der AMUNIX-Bibliothek nutzen (RPC...)[4].

Das ursprüngliche OCaml-System (ursprünglich Version 3.01, der Zeit 3.05) wurde mit der AMUNIX-Entwicklungsumgebung verschmolzen und führte zu der Entwicklungsumgebung **VAM** mit einer Vielzahl von Hilfs- und Systemprogrammen, die eine virtuelle AMOEBA-Umgebung in einem Unix-System ermöglicht. Zudem wurden einige Änderungen am Compiler, der VM und dem interaktiven System vorgenommen. Das interaktive System wurde neu gestaltet und erweitert und führte zu dem Programm **vam**, welches als interaktive AMOEBA-Unix-ML-Shell operiert. Das VAM-System enthält folgende ML-Module:

Bytebuf Low-level Puffermagament. Diese Modul bietet Unterstützung für physikalische und logische Datenbereiche, wie sie z.B. vom RPC-Modul verwendet werden. Der Unterschied zum String-Modul besteht in der Auslagerung des benötigten Speicherbereichs vom ML-Heap, der von der VM verwaltet wird, in den dynamischen Heap des Programms, wodurch das Puffermagament flexibler ausgelegt werden konnte. So können Fenster von physikalischen Speicherbereichen als logische Speicherbereiche verwendet werden.

AMOEBA, Buf, Stdcom, Stderr, Ar, Circbuf, Virtcirc Dieses Module stellen die Basisfunktionalität des AMOEBA-Systems zur Verfügung, wie z.B. Capabilitymanagement und architekturunabhängige Pufferfunktionen (`buf_put_XX` und `buf_get_XX`), Ringpuffer etc.

RPC Das Remote-Procedure-Call Modul, welches über Unix-Sockets an den FLIP-Daemon ankoppelt und die verteilte Kommunikation ermöglicht.

Afs_client, Dns_client, Name, Dir, Disk Klientenschnittstellen für das neue AMOEBA-File-System (AFS) und dem Directory-and Name-Service (DNS), sowie low-level Zugriff auf AMOEBA's virtual disk system (zur Zeit nur nativ im Kernel verfügbar). Weiterhin abstraktere Module für Verzeichnis- und Namensauflösungen.

OCaml Sämtliche vom Standard-OCaml-System bekannten Module, insbesondere die der Standardbibliothek wie String, List, Array, etc. sowie dem Compiler (in Bibliothek-Form).

Threads, Mutex, Sema Diese Module bieten Unterstützung für Multithreading in VAM. Die Programmierschnittstelle entspricht genau der, die unter dem nativen AMOEBA-System zur Verfügung steht.

Unix Das Gegenstück zum AMOEBA-Modul-System: die Schnittstelle zum Unix-Betriebssystem. Entspricht im wesentlichen dem Unix-Modul der Standarddistribution von OCaml.

Xlib, WXlib Reimplementierung der X-Windows-Basisbibliothek von Farbice Le Fessant (Projekt Para/SOR, INRIA Rocquencour) in ML. Auf diese Modul setzt das Widget-Modul wxlib auf. Es bietet einfache, objektorientierte, Programmierung von Fenstern und graphischen Applikationen, vergleichbar mit TCL/TK oder der Qt-Bibliothek.

Abbildung 3 zeigt den schematischen Aufbau des VAM-Systems, bestehend aus dem Protokollstack FLIP und der AMUNIX-Translationsschicht, einer oder mehrerer VAM-Maschinen, entweder als interaktives Skript-System vam oder als eigenständige Applikation ausgeführt.

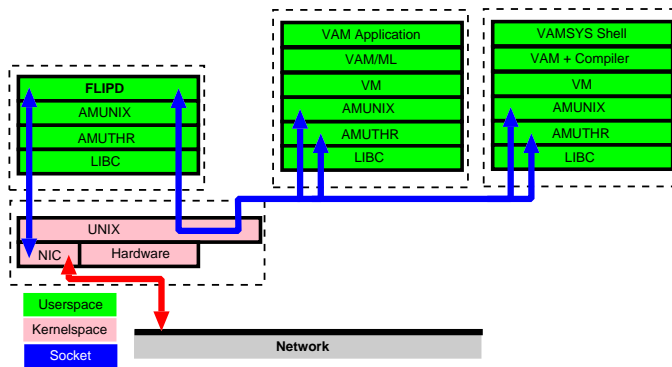


Abbildung 3: Schichtenaufbau des VAM-Systems.

VAM stellt nicht nur eine Betriebssystemumgebung, sondern auch eine im wesentlichen portable Entwicklungsumgebung für verteilte Systeme und Anwendungen dar. VAM ermöglicht die Programmierung und den Betrieb von verteilten heterogenen Systemen, kombiniert mit vorhandenen komfortablen Arbeitsumgebungen wie UNIX und X11.

6. Vergleich mit anderen Systemen

6.1. Inferno

6.2. PVM und MPI

7. Minimales VAM-System

Abbildung 4 zeigt ein Beispiel für ein VAM-AMOEBASystem, bestehend aus folgenden Komponenten:

- Dezidierte Barebone-Rechner (Embedded Systems oder PCs), die mit einem nativen AMOEBA-VERTEX-Kernel operieren, der z.B. von einer Festplatte, einer Compact-Flash-Card oder von einer Floppy gestartet werden kann. Weder Terminal noch Festplatten sind zum Betrieb dieser Systeme erforderlich.

- Unix-Maschinen, die mit der VAM- und AMUNIX-Schicht die Verbindung zu den AMOEBA-Systemen herstellen.
- Alle Rechner sind über 10- oder 100MBit/s Ethernet gekoppelt.

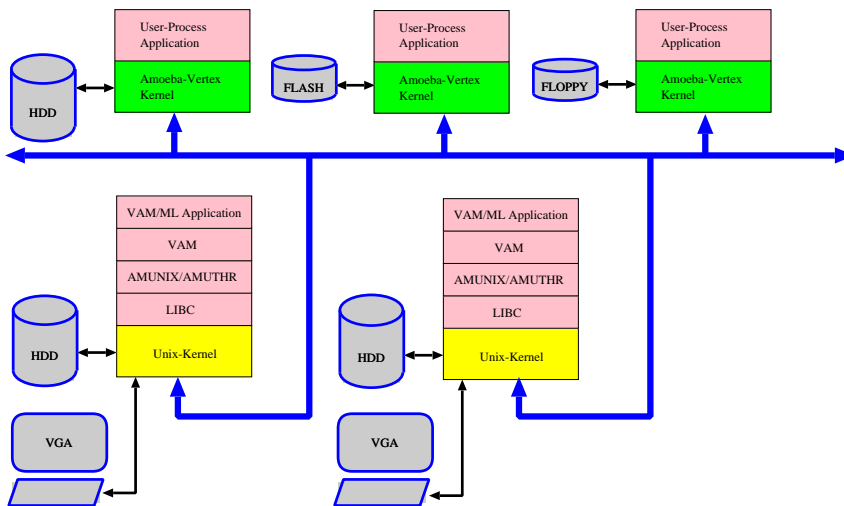


Abbildung 4: Ein Beispiel für ein VAM-AMOEBASystem bestehend aus verschiedenen Rechner- und Programmkomponenten.

8. Leistungsfähigkeit

8.1. Natives RPC

Zunächst sollen Meßergebnisse einen Eindruck der Leistungsfähigkeit von AMOEBA's FLIP-Protokollstack vermitteln. Dazu wurden zwei PC-Rechner mit einem nativen VERTEX-AMOEBASystem eingesetzt. Auf dem einen Rechner wurde ein Test-Server-, und auf dem anderen ein Test-Klienten-Programm als normale Nutzerprozesse ausgeführt. Der Klient transferierte eine vorgegebene Datenmenge mit einer festgelegten Transaktionspuffergröße zum Server, bzw. umgekehrt vom Server- zum Klient-Programm.

Tabelle Tab. 5 zeigt Ergebnisse von Leistungsmessungen mit zwei PC-Rechnern und 100MBit/s-Ethernet. Zum Einsatz kamen 3Com Netzwerkkarten, für die ein an das AMOEBA-FLIP Protokoll angepasster Hochleistungstreiber entwickelt wurde. Diese Netzwerkkarten sind zum Busmaster-DMA Betrieb fähig, und können empfangene Netzwerkpakete direkt in den Hauptspeicher, und zu versendende Pakete direkt aus dem Speicher laden. Tabelle Tab. 6 zeigt zum Vergleich Meßwerte, die mit einem deutlich leistungsschwächeren erzielt wurden (Rechner 1). Trotzdem zeigt sich keine signifikante Abnahme der Datendurchsatzrate und nur eine geringfügige Zunahme der Latenzzeit eines Null-Daten-RPC's. Die Datentransferraten liegen im Sättigungsbereich vom verwendeten Netzwerkmedium. Es zeigt somit, daß der FLIP-Protokollstack sehr effizient arbeitet, und nur einen geringen Overhead produziert, was sich auch

in den Latenzzeiten niederschlägt. Der Übergang zum Gigabit-Ethernet scheint daher sinnvoll. Nochmals hervorzuheben ist, daß die Übertragung von Nutzer- zu Nutzerprozeß stattfand.

Rechner	Transfer	Durchsatz [MByte/sec]	Latenz [μ s]
1. AMD-Duron 650MHz, 64 MB, 3Com 905CX (100 MBit/sec)	1 \rightarrow 2	11,2	130
2. Intel-Celeron, 700MHz, 64 MB, 3Com 905CX (100 MBit/sec)	2 \rightarrow 1	11,2	136

Tabelle 5: Erzielte Leistungsdaten mit dem FLIP RPC-6 Protokoll. Die Latenz ist die Dauer eines Null-Daten RPC's. Es wurden 200 MByte Daten in 1 MByte großen Fragmenten übertragen.

Rechner	Transfer	Durchsatz [MByte/sec]	Latenz [μ s]
1. Cyrix P1 100MHz, 32 MB 3Com 905B (100 MBit/s)	1 \rightarrow 2	10,5	170
2. Intel-Celeron, 700MHz, 64 MB, 3Com 905CX (100 MBit/sec)	2 \rightarrow 1	9,54	170

Tabelle 6: Erzielte Leistungsdaten mit dem FLIP RPC-6 Protokoll. Die Latenz ist die Dauer eines Null-Daten RPC's. Es wurden 200 MByte Daten in 1 MByte großen Fragmenten übertragen.

8.2. AMUNIX und VAM

Tabelle Tab. 7 zeigt die erzielten Leistungsdaten für den Fall zweier VAM/ML- und zweier AMUNIX/C-Programme, die alle auf der gleichen Maschine ausgeführt wurden. Wie oben beschrieben, findet in diesem Fall der RPC-Datentransfer über UNIX-Sockets mit dem FLIPD-Programmm statt. Es zeigt sich eine deutliche Reduzierung des Datendurchsatzes und ein Anstieg der Latenz gegenüber dem nativen AMOEBA-Fall, jedoch keine signifikante Reduzierung durch die Verwendung einer virtuellen Maschine. D.h. der Flaschenhals ist das FLIPD-Konzept. Es werden aber dennoch akzeptable Werte erzielt, die eine leistungsfähige Kommunikationsumgebung erwarten lassen.

Rechner	Transfer	Durchsatz [MByte/sec]	Latenz [μ s]
Intel-Celeron 700MHz, 128 MB, FreeBSD-4.7,VAM-1.7+AMUNIX wie oben, nur AMUNIX	1 \rightarrow 2	11,2	1700
	1 \rightarrow 2	11,9	1700

Tabelle 7: Erzielte Leistungsdaten für den lokalen Fall mit VAM-1.6 und AMUNIX. Die Latenz ist die Dauer eines Null-Daten RPC's. Es wurden 200 MByte Daten in 1 MByte großen Fragmenten übertragen.

Rechner	Transfer	Durchsatz [MByte/sec]	Latenz [μ s]
1. Cyrix P1 100MHz, 32 MB 3Com 905B (100 MBit/s), Vertex-Amoeba 2. Intel-Celeron, 700MHz, 64 MB, 3Com 905CX (100 MBit/sec) VAM-1.7+AMUNIX	1 \rightarrow 2	10,5	170
	2 \rightarrow 1	9,54	170

Tabelle 8: Erzielte Leistungsdaten mit dem FLIP RPC-6 Protokoll. Die Latenz ist die Dauer eines Null-Daten RPC's. Es wurden 200 MByte Daten in 1 MByte großen Fragmenten übertragen.

9. Sicherheit

Sicherheit in Amoeba wird im wesentlichen durch das Kommunikationsprotokoll FLIP erreicht. Zwei Kernpunkte von FLIP sind hervorzuheben:

1. Nachrichten können vom Absender als "empfindlich" markiert werden ("Security Bit").
2. Nachrichten, die von FLIP transportiert werden, können aufgrund Hintergrundwissen als unsicher markiert werden ("Unsafe bit").

Die RPC-Implementation basiert auf einem geteilten Schlüsselkonzept zwischen Klienten und Server. Es gibt keine speziellen Authorisierungs- und Authentifizierungsserver in Amoeba. Dadurch wird die Klient-Server-Kommunikation effizient und einfach gehalten. Nachrichten werden unverschlüsselt.

Das unauthorisierte Überdecken von Servern wird durch einen mittels Einwegverschlüsselung erzielten öffentlichen Schlüssel verhindert. Der Server benutzt nur den ihm bekannten Schlüssel, und veröffentlicht den Einwegverschlüsselten. Bei jeder Klientenanfrage wird der öffentliche mit dem privaten Schlüssel verglichen.

A. Beispiel

Im folgenden wird ein kurzes Beispiel für einen einfachen VAM-Server gegeben, wie er in ML programmiert wird. Das gezeigte Code-Fragment bildet einen einfachen Terminal-Server, der Zeichen auf der gewählten Konsole ausgibt und von dieser einliest und Klientenprogrammen zur Verfügung stellt.

```
(*
** Simple terminal server.
*)

open AMOEBA
open Stderr
open Stdcom
open Stdcom2
open Cmdreg
open Rpc
open Thread
open Ar
open Io
open Bytebuf
open Buf

let tty_READ    = Command (tty_FIRST_COM+52)
let tty_WRITE  = Command (tty_FIRST_COM+53)

let tty_REQBUFSZ = 1024

type tty_server = {
  mutable tty_getport: port;
  mutable tty_putport: port;
  mutable tty_checkfield: port;
}

let tty_dying = ref false
(*
** The server thread
*)
let tty_srv ~server
           ~sema
           ~nthreads
           ~inbuf_size
           ~outbuf_size
           =
```

```
tty_dying := false;
let initial = ref false in
  (*
  ** The exit request must be served. Send all other server threads the
  ** STD_EXIT request, too.
  *)
  let on_exit cap =
    let stat = if (!tty_dying = false) then
      begin
        tty_dying := true;
        (*
        ** Tell the other threads we're
        ** dying.
        *)
        for i = 1 to (nthreads-1)
        do
          ignore(std_exit cap);
          Sema.sema_up sema;
        done;
        std_OK
      end
    else
      std_OK
    in
    stat
  in
  try
  begin
    (*
    ** Generic input and output rpc buffers
    *)
    let ibuf = buf_create inbuf_size in      (* request buffer *)
    let obuf = buf_create outbuf_size in    (* reply buffer  *)

    (*
    ** Get a new transaction header
    *)
    let hdr_rep = header_new () in
    let replen = ref 0 in

    let getport = server.tty_getport in
    let putport = server.tty_putport in
    let checkfield = server.tty_checkfield in
```

```
while (true)
do
try
begin
  (*
  ** Wait for client requests...
  *)
  let stat,reqlen,hdr_req = getreq (getport,ibuf,inbuf_size) in

  replen := 0;
  hdr_rep.h_size <- 0;
  hdr_rep.h_status <- std_OK;
  hdr_rep.h_priv <- priv_copy nilpriv;

  let priv =hdr_req.h_priv in
  (
    match hdr_req.h_command with
    (*
    ** Terminal write request
    *)
    | com when (com = tty_WRITE) ->
    begin
      let Objnum obj = prv_number hdr_req.h_priv in
      let Rights_bits rights = prv_rights hdr_req.h_priv in

      let stat = ref std_OK in
      let pos = ref 0 in
      let size = ref (min hdr_req.h_size inbuf_size) in

      if (prv_decode ~prv:priv ~rand:checkfield
          = true) then
        begin
          stat := tty_write ~buf:ibuf
                        ~size:!size;
        end
      else
        stat := std_DENIED;
        hdr_rep.h_size <- !size;
        hdr_rep.h_status <- !stat;
        replen := !pos;
      end;
    end;

    (*
    ** STD_INFO request
```

```
*)

| com when (com = std_INFO) ->
begin
  let Objnum obj = prv_number hdr_req.h_priv in
  let Rights_bits rights = prv_rights hdr_req.h_priv in

  let stat = ref std_OK in
  let pos = ref 0 in
  if (prv_decode ~prv:priv ~rand:checkfield
      = true) then
    begin
      let pos' =
        buf_put_string
          ~buf:obuf
          ~pos:0
          ~str:"+ (terminal)"
      in
      pos := pos';
    end
  else
    stat := std_DENIED;

  hdr_rep.h_size <- !pos;
  hdr_rep.h_status <- !stat;
  replen := !pos;
end;

(*
** Close the server thread.
** Only possible with the super capability.
**
*)
| com when (com = std_EXIT) ->
begin
  initial := not (!tty_dying);
  let rights = prv_rights hdr_req.h_priv in
  let Objnum obj = prv_number hdr_req.h_priv in
  if (obj = 0 &&
      (prv_decode ~prv:hdr_req.h_priv
                  ~rand:checkfield)
      ) = true then
    begin
      let stat = on_exit { cap_port = hdr_req.h_port;
```

```

                                cap_priv = priv} in
      hdr_rep.h_status <- stat;
      ignore( putrep (hdr_rep,obuf,!replen));
      raise Exit;
    end
  else
  begin
    hdr_rep.h_status <- std_DENIED;
  end;
end;
| _ ->
begin
  let Command com = hdr_req.h_command in
    hdr_rep.h_status <- std_COMBAD;
  end;
);

(*
** Send a reply.
*)
ignore( putrep (hdr_rep,obuf,!replen));
end
with
| Buf_overflow ->
begin
  replen := 0;
  hdr_rep.h_status <- std_ARGBAD;
  ignore( putrep (hdr_rep,obuf,!replen));
end;
done;
end
with
| Exit ->
  print_string "TTY: Server thread exited normally.";
  print_newline ();
  if (!initial) then
    Sema.sema_up sema

let _ =
  (*
  ** Create the server ports
  *)
  let getport = uniqport () in
  let putport = priv2pub getport in

```

```

let checkfield = uniqport () in

let srv = {
  tty_getport = getport;
  tty_putport = putport;
  tty_checkfield = checkfield;
} in

(*
** Create the public server capability
*)
let pubcap = {cap_port = putport;
              cap_priv = prv_encode ~obj:(Objnum 0)
                                ~rights:prv_all_rights
                                ~rand:checkfield
} in
(*
** Publish the server capability in the Amoeba filesystem
*)
let stat,cap = name_lookup !capfile in
if (stat = std_OK) then
begin
  let stat = name_delete !capfile in
  if (stat <> std_OK) then
    failwith "can't delete old cap";
end;
let stat = name_append !capfile pubcap in
if (stat <> std_OK) then
  failwith "can't append cap";

let sema = Sema.sema_create 0 in

(*
** Start server threads
*)
for i = 1 to !nthr
do
  ignore(thread_create (fun () ->
    tty_srv ~server:srv
           ~sema:sema
           ~nthreads:!nthr
           ~inbuf_size:tty_REQBUFSZ
           ~outbuf_size:tty_REQBUFSZ
    ) ());

```

```
done;  
(*  
** Wait for server thread to be finished  
*)  
for i = 1 to !nthr  
do  
    Sema.sema_down sema;  
done;
```

Referenzen

- [1] A. S. Tanenbaum, R. van Renesse, H. van Staveren, S. J. Mullender, J. Jansen, G. van Rossum
Experiences with the AMOEBA Distributed Operating System
Commun. ACM, vol. 33, pp. 46-63, Dec. 1990
- [2] A. S. Tanenbaum, M. F. Kaashoek
The AMOEBA Microkernel
1994
- [3] <http://caml.inria.fr>
- [4] E. Chailoux, P. Manoury, B. Pagano
Developing Applications With Objective Caml
preliminary
- [5] M. F. Kaashoek, R. van Renesse, H. van Staveren, H., A. S. Tanenbaum
FLIP: an Internetwork Protocol for Supporting Distributed Systems
ACM Transactions on Computer Systems, pp. 73-106, Feb. 1993
- [6] Amoeba-5.3 Programming Manual, Vrije Universiteit, 1996