Eine Übersicht über das Betriebssystem Inferno

Susanne Wenger

27. Juli 2002

Inhaltsverzeichnis

Zusammenfassung

Inferno wurde entwickelt von Sean Dorward, Rob Pike, Phil Winterbottom, Eric Grosse, Dave Presotto, Dennis Ritchie, Ken Thompson und Howard Trickey von `Bell Labs Computing Sciences Research Center` in New Jersey, USA - am gleichen Ort wo auch Unix, C und C++ entstand. 1996 nach einem Jahr Arbeit an Inferno wurde `Vita Nuova` mit Sitz in York, England von Lucent Technologies' Bell Labs für den Verkauf, Support und Entwicklung von Inferno gegründet. Unter $http://www.lucent.com/inferno$ kann man das hosted Inferno evaluation package für Plan 9, FreeBSD, Linux/x86, Solaris/SPARC, Windows 95/98/ME/NT/2000 herunterladen. Darin sind die Dis virtual machine, Support für das Styx-Netzwerkprotokoll, der Limbo-Compiler, C cross-compilers für 68000, 68020, ARM, Power PC, x86 und Sparc, das Tk user interface toolkit und Limbo-Sourcecode für u.a. folgende Anwendungen und Libraries enthalten:

- Charon Webbrowser
- Acme integrating Entwicklungsumgebung
- Yacc Compiler-Compiler
- sh programmable shell
- plumbing support
- verschiedene Unix-ähnliche Tools: mv, cp, rm, xd, wc, grep, ps, diff, tr, man, ls,..

Inferno ist ein verteiltes, architekturunabhängiges Network OS (NOS). Die Idee von Inferno war die Komplexität und Kosten der Softwareentwicklung zu reduzieren. Um dies zu erreichen offeriert Inferno verschiedene Features. Zum Beispiel ist Inferno gut skalierbar. Desweiteren läuft das Betriebssystem Inferno inkl. grundlegender Anwendungen mit nur 1MB RAM, wobei keine memory-mapping Hardware erforderlich ist. Dies macht Inferno besonders gut geeignet für verteilte Systeme wie network computers (NCs), Internet terminals, set-top boxes und PDAs. Zudem macht die Architektur Inferno zu einem idealen Interface für high-performance Server-Anwendungen. Zum Beispiel erlaubt Inferno einem Internet Service Provider Software zu schreiben, die auf verschiedener Hardware und Betriebssystemen läuft. Dies erreicht Inferno mit Hilfe von verschiedenen Abstraktionsebenen. So bietet Inferno eine virtuelle Maschine (VM) an, um Hardware-Differenzen zu verbergen. Zudem bietet Inferno - im Gegensatz zu Java - noch ein virtuelles Betriebssystem (VOS) und ein virtuelles Netzwerk (VN) an. Ausserdem erlaubt das Styx-Protokoll, dass alle Ressourcen in einem Netzwerk als Dateien dargestellt werden und mit `open`, `close`, `read` und `write` manipuliert werden können. Als Folge davon ist das Entwickeln von Software mit Inferno erleichtert. Desweiteren sind Security- und Authentication-Features in Inferno integriert.

1 Anwendungsgebiet

Infernos Kunden sind u.a. `Sun Microsystems`, `Tatung` und `Philips`. Eine Nachfragesteigerung wird von `Vita Nuova` erwartet, da auch die Diversität von intelligent computing und communications devices zunimmt. So wollen Anwender auf immer verschiedenere Netzwerk-Arten Emails, News und Unterhaltung holen (z.B. via Natel). Bis jetzt haben Entwickler wegen den verschiedenen Hardwarekomponenten immer viel zu viel Zeit mit der Entwicklung von low-level Details gebraucht. Nun ermöglicht Inferno die Entwicklung auf die Main-Features zu konzentrieren.

1.1 Beispiel: Chunghwa Telecom Co.

Inferno 1.0 wird von `Chunghwa Telecom Co.`, Ltd. of Taiwan eingesetzt, um ihre Internet Screenphone in einem landesweiten Internet Service Provider Netzwerk zu unterstützen. Innerhalb von 3 Jahren sollen so 3 Millionen zusätzliche Internet Users gewonnen werden. Dabei soll Inferno den Internet Screenphone Benützer ermöglichen, sich für Services wie electronic mail, web browsing, home shopping und home banking zu registrieren. Der Vorteil und Grund für den Einsatz von Inferno war hier aber hauptsächlich die einfache Integration von Inferno in die bisherige Infrastruktur, die Flexibilität von Inferno und den Support für portable, kleine, netzwerkfähige und verschlüsselte Systeme. Zudem erlaubt Inferno `Chunghwa Telecom Co.` eine Expansion auf den internationalen Markt, da Anwendungen in Limbo auf allen Inferno fähigen Netzwerken laufen.

2 Architektur

Inferno beinhaltet den Inferno-Kernel, das Styx-Protokoll, die Limbo-Programmiersprache und die Dis-virtual machine. Zudem sind noch Treiber für Sprach- und Audio-Anwendungen, open database connectivity (ODBC) für Informix, Microsoft, Sybase und Oracle Datenbanken und eine Reihe von Kommunicationsprotokolle, Netzwerkanwendungen und Graphiklibraries vorhanden.

2.1 Virtuelles Betriebssystem (VOS)

Inferno bietet eine virtuelle Maschine (VM) an, um Unterschiede in der Hardware zu verbergen. Zudem hat Inferno noch ein virtuelles Betriebssystem (VOS) und ein virtuelles Netzwerk (VN). Die Idee eines virtuellen Betriebssystems ist analog zu einer normalen, herkömlichen VM. Inferno's VOS definiert ein Interface zu den Systemservices. Dies kann auf 2 verschiedene Arten geschehen. Falls Inferno auf einem anderen Betriebssystem gehosted ist, werden VOS-Services auf das native OS-Services gemappt. Auf der anderen Seite kann Inferno native laufen - was bzgl. der VOS-Services bedeutet, dass sie direkt vom Kernel implementiert werden müssen.

2.2 Layers

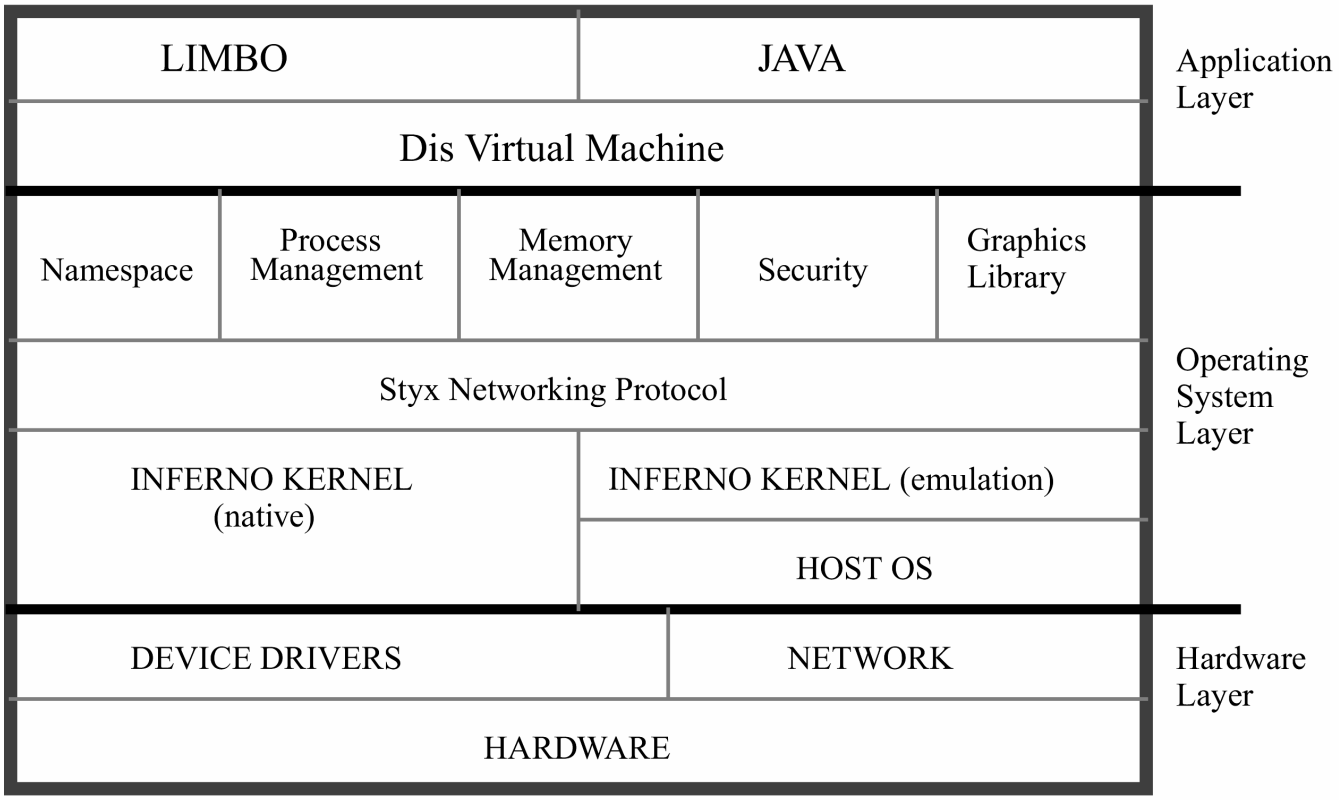

Abbildung 1: Inferno Architecture

2.2.1 Application Layer

Hier laufen Limbo-Anwendungen auf einer virtuellen Machine, genannt Dis. Dazu werden Aufrufe von Limbo an die untere Schicht weitergegeben. Dabei stellt das Inferno System verschiedene Standardinterfaces für Anwendungen bereit. Deshalb spielt es für den Entwickler auch keine Rolle, für welche Hardware er eine Anwendung implementiert. Zum Beispiel ist das Netzwerk abstrahiert, so dass man ein LAN oder eine Modem-Verbindung über die traditionelle Telefonleitung gleich ansprechen kann. Alle Programme laufen auf einer virtuelllen Maschine (Dis), welche für alle Prozessor-Architekturen die gleichen Interfaces zu Verfügung stellt. Dis ist eine einfache 3-Address Machine mit einigen speziellen Operationen, um higher-level Datentypen wie arrays oder strings zu bearbeiten. Garbage collection und das Scheduling von Tasks werden auf low-level Ebene gehandled. Wenn Code in den Speicher zur Ausführung geladen wird, wird der Bytecode in ein effizienteres Format für die Ausführung expandiert. Ein optionaler JIT-Compiler kann Dis-Anweisungs-Streams in nativen Maschinencode übersetzen. Das Resultat ist fast so schnell wie compilierter C-Code, weil die Dis-Anweisungen auf die Instruction-Set Architektur moderner Prozessoren abgestimmt sind.

2.2.2 Operating System Layer

Folgende Funktionalitäten werden im Operating System Layer zur Verfügung gestellt:

- Namespace Management
- Prozess Management
- Speicher Management
- Security
- Graphikfunktionen

In diesem Layer befindet sich auch der Kernel. Der Kernel hat folgende Aufgaben:

- Prozess Erzeugung
- Namespace Management
- Data Streaming
- Netzwerk Protokolle

Desweiteren ist der Kernel für I/O und Hardwareanbindung via Treiber verantwortlich. Dabei stellt ein hosted Inferno ein Spezialfall dar, da hier die Befehle an das native System weitergegeben werden müssen.

2.2.3 Hardware Layer

Der Hardware Layer besteht haupsächlich aus der physikalischen Hardware, dem Netzwerk und den Treibern.

2.3 Memory Management

Inferno hat 2 Arten von Allokationsmechanismen. Der low-level Mechanismus kontrolliert grosse zusammenhängende Blöcke, während der high-level die Blöcke in Pools aufteilt. Dies erlaubt Anwendungen bei knappen Ressourcen das Verhalten des Systems zu bestimmen. Die Allokation von Speicher-Pools wird für

- Allokation von allgemeinen Speicherstrukturen
- Speicher für den Dis Heap
- Speicher für Graphik und Schriften
- Netzwerkbuffer-Pool

zur Verfügung gestellt. Der Speicher wird in Blöcken in einem unbalanced B-Tree gespeichert. Dabei sind die Blätter nach Grösse geordnet. Wenn ein Prozess Speicher angefordert wird der Baum mit einem best-fit Algorithmus nach einem Block der gross genug ist durchsucht. Wenn ein verfügbarer Block 25% grösser als erforderlich ist, wird er gesplittet. Wenn der Speicher wieder freigegeben wird, wird der Block in den Baum zurückgefügt, wobei bei 2 benachbarten freien Blöcken diese zusammengelegt werden. Dieses Memory Management stellt eine schnelle Allokation und eine relativ geringe Fragmentation des Speichers sicher.

2.4 Scheduling

Der Kernel stellt premptives Scheduling für Prozesse, welche Protokollstacks, Alarme, Interrupts,... handeln, zu Verfügung. Dabei wird für Prozesse mit verschiedenen Prioritäten ein Round-Robin-Verfahren angewendet.

2.5 Komponenten

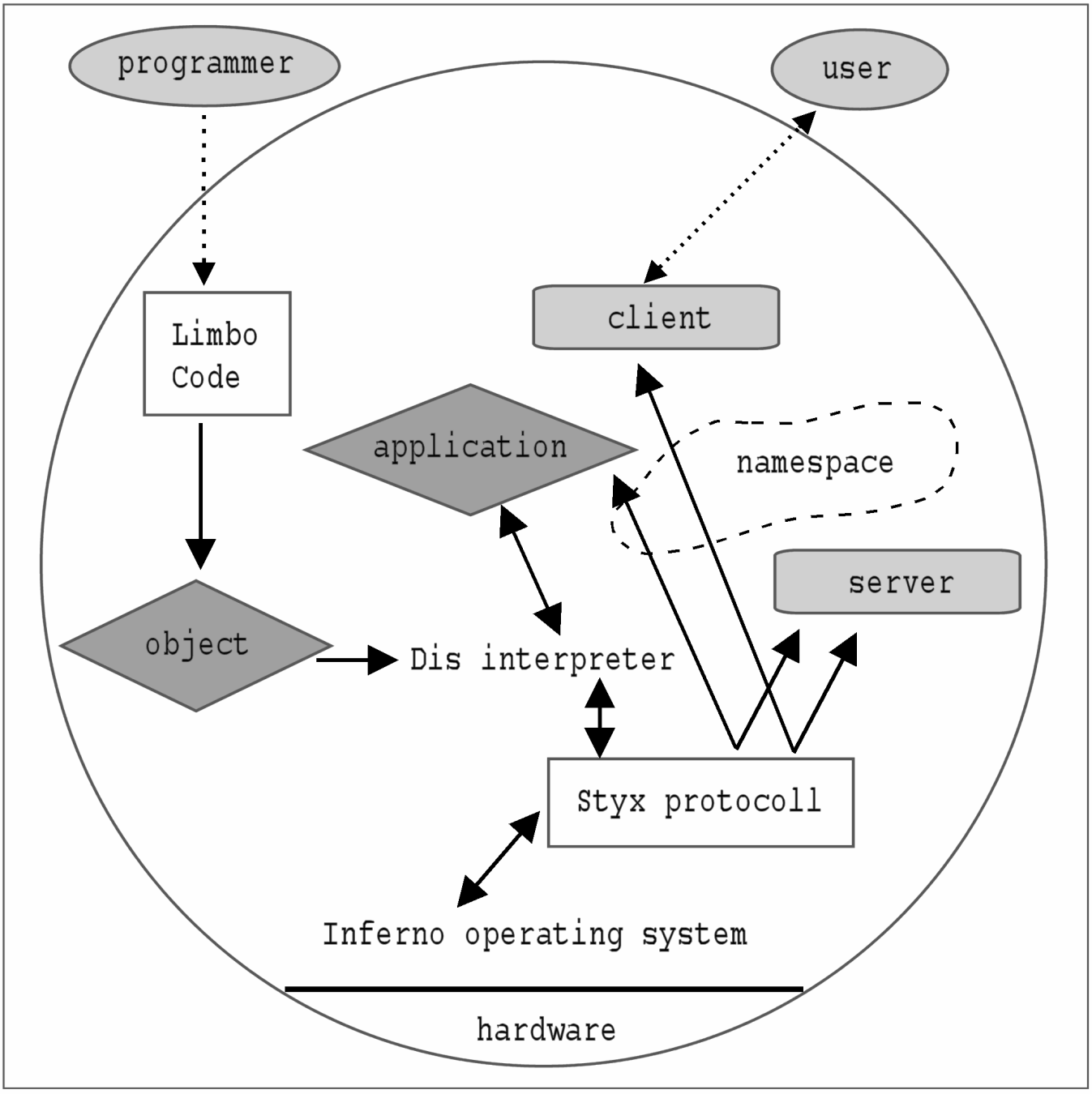

Abbildung 2: Components

3 Namespace

Es gibt viele Eigenschaften, die man von einem Betriebssystem erwartet. Konventionelle Features sind z.B. Threads, Device Access und Network Filesystem. Spezielle Features von Inferno sind zudem, dass alle diese Ressourcen als Dateien dargestellt werden - somit können Dateien also Interfaces zu Peripherie, Programmen, Services, Netzwerk,... sein. Zudem können lokale und remote Filesysteme in dynamisch konfigurierbare, hierarchische Namespaces zugeordnet werden. Dabei versteht man unter Namespace eine hierarchische Sammlung von lokalen oder remote Dateien.

```
perky$ echo hallo > /dev/cons
hallo               # Datei /dev/cons ist Interface zur Konsole
```

3.1 Vorteile

- Dateien sind Interfaces zu Peripherie, Programmdiensten, Netzwerk,... somit wird die Komplexität von verschiedenen Ressourcen reduziert
- Systeminterfaces, welche System calls erfordern werden durch spezielle Dateien dargestellt. Dies führt zu einer Reduktion der Grösse von Inferno.
- Mit `open()`, `read()`, `write()`, `close()`,... kann man auf Dateien, Peripherie, Programme, Services, Netzwerk,... zugreifen, was äusserst einfach und programmierfreundlich ist..
- Allen Dateien können Benutzerrechte zugeordnet werden, was eine Restriktion von Ressourcen für bestimmte Anwendungen erlaubt.
- Weil jedes Filesystem (z.B. Ressourcen) auf jede Node eines Netzwerkes exportiert werden kann, kann das erforderliche File für eine Anwendung entweder lokal oder remote sein.
- Konfiguration und Applikation können gut getrennt werden. Ressourcen erscheinen nur dort, wo sie wirklich gebraucht werden, so dass man nicht die Anwendungen an die Ressourcen anpassen muss.

3.2 Threads und Namespaces

Dabei übernimmt Inferno von Plan 9, dem Vorgänger von Inferno, die Idee eines Prozessspezifischen Namespaces, der vererbt und gemeinsam genutzt wird, dann aber mit `pctl()` gelöscht oder abgetrennt werden kann. Die Shell trennt ihren Namespace beim Start ab, vererbt ihn aber an die erzeugten Kommandos. Folglich ist `cd` ein eigenes Kommando, das sich aber in einem Shell-Skript nicht auf die ursprüngliche Shell auswirken kann:

```
perky$ cd /tmp; pwd
/tmp
perky$ echo 'cd /; pwd' > root
perky$ sh root; pwd
/               cd wirkt sich auf Skript-Shell aus
/tmp            ...aber nicht auf Top-Shell
```

Ebenso können sich die Kommandos `bind`, `mount` und `unmount` auf den Namespace der aufrufenden Shell auswirken. `bind` dient zum Umbau des Namespaces. Verzeichnisse oder Dateien können verdeckt werden, Verzeichnisse können aneinandergefügt werden. Dies erlaubt es Resourcen zu konfigurieren und abzuändern ohne die Anwendung zu beeinflussen. Da man insbesondere private Verzeichnisse mit `/dis` mounten kann, benötigt die Shell kein PATH-Konzept. Kommandos werden vom aktuellen Verzeichnis und von `/dis` aus gesucht; letzteres nur, wenn sie nicht mit `./` beginnen. Im Inferno-Kern gibt es einen Namensraum, für dessen Wurzel `#` steht. Die direkten Unterbäume haben Namen, die aus einem einzigen Zeichen bestehen. Dieses Zeichen wählt einen im Kern gebundenen Server (Device) aus, der seine Resourcen als Dateisystem zur Verfügung stellt. Dabei wird der Namespace im Kernel als eine Liste von Relationen zwischen Channels und internen Listen von Channels dargestellt ebenso wie auch der Filedescriptor als Channel dargestellt wird. Für weitere Informationenen zu `bind`, `mount` und `unmount` vergleiche
$http://www.vorlesungen.uni-osnabrueck.de/informatik/inferno/man/html/md\_sys1.htm.$

3.3 Anwendung: Portieren von Inferno

Um ein Betriebssystem zu portieren, muss man zuerst mindestens die Treiber für Tastatur und Bildschirm implementieren. Mit Inferno jedoch kann man ganz einfach die entsprechenden Geräte über das Netzwerk importieren sobald CPU und Netzwerk unterstützt werden. Zum Beispiel um Tastatur, Bildschirm von einer anderen Maschine zu importieren:

```
mount 182.1.1.3 /n/remote
bind /n/remote/dev/keyboard /dev/keboard
bind /n/remote/dev/draw /dev/draw
bind /n/remote/dev/pointer /dev/pointer
```

Nach diesem Schritt läuft der Kernel nun auch auf dem neuen Gerät und der Benützer kann über den Bildschirm, Maus und Tastatur jeder anderen Maschine, auf der Inferno (hosted oder native) läuft bedienen.

3.4 Anwendung: Time Lapse Photography

Unter $http://www.vitanuova.com/mkt/press/Inferno\_overview.pdf$ findet man auf Seite 3 ein weiteres Beispiel, wie man Namespaces brauchen kann. Dabei werden Windows PC, Digital Camera und Video Recorder und Compaq iPAQ auf sinnvolle Weise verknüpft.

4 Styx Protocol

Inferno repräsentiert Geräte transparent und erlaubt Devices, wie Dateien zu behandeln. Um auf diese Ressourcen zuzugreifen, benutzt man das Kommunikations-Protokoll Styx. Es ist dabei nicht nötig, dass Anwendungen einen Unterschied zwischen lokalen und remote Ressourcen machen. Das Styx-Protokoll liegt oberhalb des Transport Layer. Es läuft über TCP/IP, Point-to-Point Protocol (PPP), ATM und einigen Modem Protokollen. Styx implementiert die zu den Device Switches entsprechenden Befehle wie `open`, `read`,... Jedenfalls führt Inferno lokale Prozeduren auf, wenn die anzusprechende Datei lokal ist, und sonst werden die Prozeduren in entsprechende Styx-Befehle umgewandelt.

4.1 Format

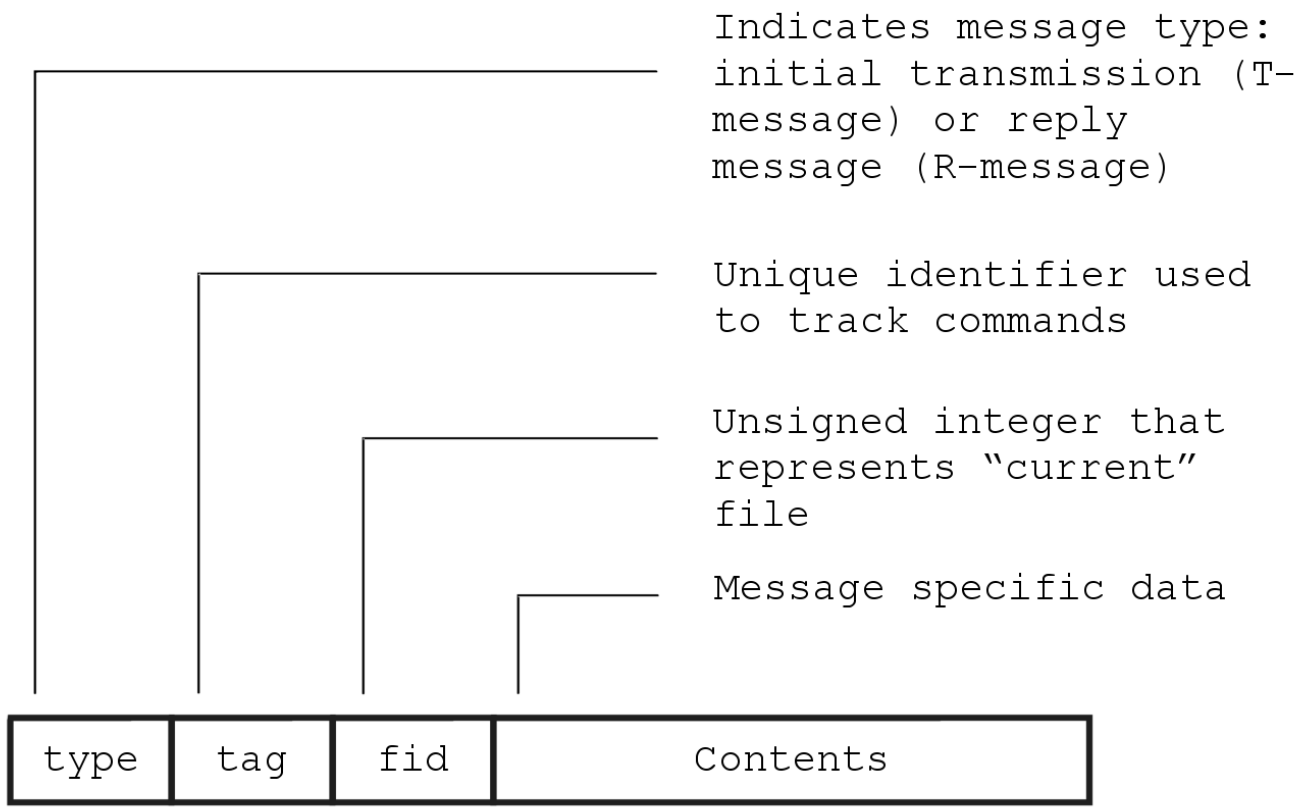

Abbildung 3: Styx

Das erste Byte bezeichnet den Message-Type, welche einer der beiden in `styx.m` definierten Konstanten ist. Normalerweise handelt es sich um eine T-Message, d.h um eine Anfrage des Klienten an den Server, oder um eine R-Message, d.h um eine Anfrage des Servers an den Klient. Das `tag` Feld wird vom Klient festgelegt und muss die Message eindeutig identifizieren. Zudem hat die Antwort einen höheren `tag`-Wert - ausser ein Fehler tritt auf (`Rerror`). Dieses Feld ist 2 Bytes lang. Danach kommt nochmals ein 2 Byte langes `fid`-Feld. Dieses Feld identifiziert die Datei, auf welche der Styx-Befehl ausgeführt werden soll. Am Schluss kommt noch ein Feld variabler Länge, welches alle nötigen Argumente für die Befehle beinhaltet

4.2 Styx-Transaktion

Ein Inferno-Server bietet seinen Klienten Dateisysteme über eine bidirektionale, sichere Kommunikationsleitung an, auf der mit Styx-Nachrichten verhandelt wird. Der Klient schickt eine T-Nachricht, der Server antwortet mit einer R-Nachricht des gleichen Typs oder mit `Rerror`. Beide Nachrichten zusammen bilden eine Transaktion. Jede Nachricht enthält ein Etikett `tag`, das der Klient für die T-Nachricht innerhalb der offenen Transaktionen eindeutig erfindet, und das dann die R-Nachricht verknüpft. Da nicht unbedingt synchron geantwortet werden muss, sind diese Etiketten nötig. Der Server hat für jedes Objekt eine permanent eindeutige Bezeichnung `qid`, die sich aus einem Pfad und einer Version zusammensetzt. Der Klient liefert eine Bezeichnung `fid`, unter der dann die Zugriffe erfolgen; nach `clunk` kann eine `fid` wieder neu

eingeführt werden. Systemaufrufe wie `chdir()` werden vom Kernel durch Folgen von Transaktionen realisiert. Transaktionen führen im Kernel zu Funktionsaufrufe an die internen Server (Devices). Ein Device ist `#M`, das dann mit Messages lokale oder fremde Prozesse anspricht.

4.3 Features

Jede Styx-Meldung kann optional eine kryptographische Signatur enthalten und/oder verschlüsselt sein. Dies kommt den Anforderungen nach einem sicheren Netzwerk entgegen - verbundene Inferno-Knoten werden authentifiziert durch public-key-Encryption Schemata. Zudem kann festgestellt werden, ob die Pakete in der richtigen Reihenfolge sind und gegebenenfalls die Pakete richtig anordnen.

5 Dis VM

Die virtuelle Maschine Dis stellt eine Umgebung zur platformunabhänigen Entwicklung und Ausführung von Software zur Verfügung. Eigentlich handelt es sich um eine einfache 3-Address Machine mit einigen speziellen Operationen, um higher-level Datentypen wie arrays oder strings zu bearbeiten.

5.1 Befehlssatz

Der Befehlssatz der Dis entspricht derjenigen vieler üblicher Prozessoren. Anweisungen sind von der Form

```
OP   src1,   src2,   dst
```

Die `src1`- und `dst`- Rechengrössen spezifizieren Adressen oder beliebig grosse Konstanten, während die `src2` auf kleinere Konstanten oder indirekte Adressen mit kleinen Offsets beschränkt ist, um Code zu sparen. Jeder Operand spezifiziert eine Adresse entweder im Stackframe oder in den globalen Daten des Moduls. Die Typen der Operanden werden durch die Instruktion festgelegt. Grundlegende Typen sind Wort (32-bit signed), (64-bit signed), Byte (8-bit unsigned), Reale (64-bit Gleitkomma IEEE) und Pointer (implementation-dependent). Der Befehlssatz folgt dem Beispiel der CISC-Prozessoren und stellt Operationen für Arithmetik und Datenmanipulation zur Verfügung. Zudem sind noch Instruktionen zur Allokation von Speicher, Prozess-Generierung,-Kommunikation und -Synchronisation vorhanden.

5.2 Leistungsfähigkeit

Garbage collection und das Scheduling von Tasks werden auf low-level Ebene gehandled. Im Gegensatz zur Java VM's stack-based Architektur soll Dis' memory-to-memory Architektur die Übersetzung von Bytecode in Maschinencode einfacher machen. Die Dis und der JIT compiler wurden beide in ANSI-C geschrieben, um Portabilität und Performance zu gewährleisten. Laut Lucent soll die Leistung des JIT Compilers etwa 30%-50% Geschwindigkeit einer reinen C-Anwendung erreichen, weil viele Dis-Anweisungen in eine einzige Pentium-Instruktion übersetzt werden bzw. die Instruction-Set Architektur moderner Prozessoren abgestimmt sind. Als weiterer Vorteil gilt, dass der Dis Interpreter kleiner als die Java VM ist und eine weitere Leistungssteigerung dadurch erreicht wird, dass der Bytecode zuerst beim Laden in ein spezielles Format umgewandelt wird und erst danach compiliert/interpretiert wird.

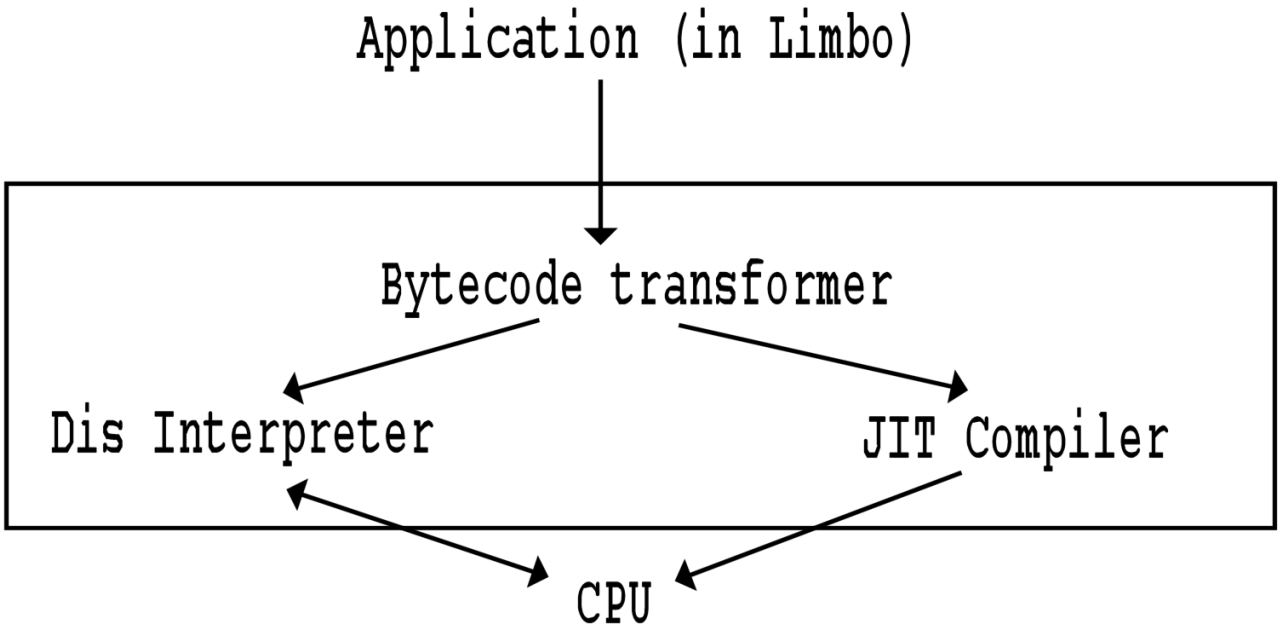

Abbildung 4: Dis Interpreter

5.3 Speicher Management

Nur eine Instanz von Dis kontrolliert den `heap`. Weil keine Locks im `heap` existieren können auch nicht mehrere Instanzen Speicher teilen. Alle Module und Daten teilen einen `data heap`. Dabei werden Module-Instanzen und Stackframe durch den `heap` alloziert. Jedes `heap`-Objekt wird dann mit dem Filedescriptor assoziiert.

5.4 Programmausführung

Mehrere Threads werden im Round-Robin-Verfahren abgearbeitet. Dabei können die Threads folgende Zusatände annehmen: Die Ausführung eines Threads wird unterbrochen falls der

alt	Inter-thread communication processing
broken	Thread has crashed
delete	Remove from queue
exiting	Thread instructions completed
ready	Available to execute instructions
receive	Ready to receive a value from another thread
release	Remove from queue to complete a kernel call
send	Ready to transmit a value to another thread

Tabelle 1: Zustände von Threads

Thread im Zustand `exiting`, `send`, `receive` ist oder sein Quantum abgearbeitet hat. Daraus ergibt sich folgendes Zustandsdiagramm:

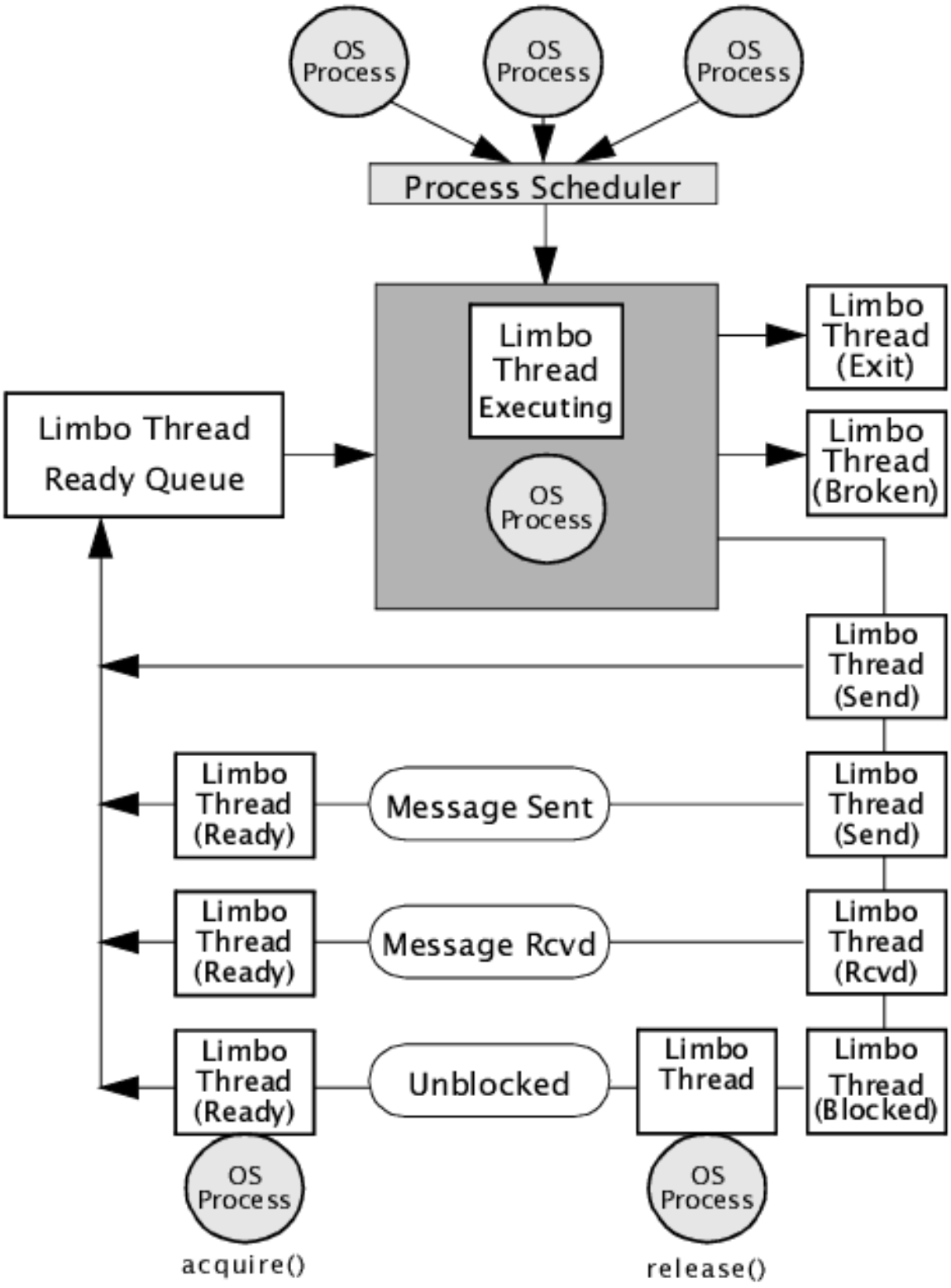

Abbildung 5: Zustandsdiagramm

6 Limbo

Der Kernel von Inferno wurde in C geschrieben, was das ganze Betriebssystem portabel macht. Die higher-level Module und Libraries wurden dann aber in Limbo geschrieben, einer extra für Inferno entwickelte Programmiersprache. Mit Limbo kann man nicht nur Anwendungen für Inferno schreiben, sondern auch Interfaces zu Services bereitstellen, die nicht native auf Inferno laufen müssen.
Limbo ist

- prozedural
- concurrent
- modular
- block-structured
- strongly typed
- dynamic

Die Syntax der Expressions und Statements ist ähnlich wie in C, die Syntax der Deklaration wie in Pascal. Insbesondere ist Limbo nicht objekt-orientiert - wie das bei Java der Fall ist - und beinhaltet auch keine Sicherheitsfeatures. Mit Limbo ist im System auch gerade u.a. die Tk Graphiklibrary, File Manipulation, Mathfunktionen, Lineare Algebra, Verschlüsselung, String-parsing integriert. Zu Limbo gehört zudem ein Editor, ein Debugger und ein Compiler.

6.1 Hello World

```
1   implement Command;
2   include "sys.m";
3   include "draw.m";
4   sys:    Sys;
5   Command: module
6       init: fn (ctxt: ref Draw->Context, argv: list of string);
7   };
8   # The canonical "Hello world" program, enhanced
9   init(ctxt: ref Draw->Context, argv: list of string)
10  {
11      sys = load Sys Sys->PATH;
12      sys->print("hello world\n");
13      for (; argv!=nil; argv = tl argv)
14          sys->print("%s ", hd argv);
15      sys->print("\n");
16  }
```

6.2 Features

6.2.1 Typen

Limbo ist strongly typed, d.h alle Variablen und Funktionen müssen explizit deklariert werden. Auch wird das Typechecking zur Compilierzeit und Moduleladezeit durchgeführt. Primitive

Datentypen sind

- int
- real
- string (Unicode)

Zudem werden folgende Datentypen unterstützt:

- Strings
- Listen
- dynamische Arrays
- Tuples
- Channels
- Definition von abstrakten Datentypen

6.2.2 Garbage Collection

Garbage Collection wird automatisch gemacht, wenn die letzte Referenz auf einem Datentyp gelöscht wird. Dies bedeutet das keine zyklischen Datenstrukturen möglich sind, ausser man spezifiziert dies explizit mit dem Keyword `cyclic`. Dann wird die Datenstruktur gelöscht, wenn nur noch interne Referenzen vorhanden sind.

6.2.3 Lexical Scoping

Limbo benützt lexical scoping, d.h. dass die Sichtbarkeit von Variablen durch die Struktur des Programms festgelegt wird. Genauer bedeutet dies, dass Variablen nur in dem Block sichtbar sind, wo sie definiert worden sind, und in den Unterblöcken dieses Blocks. Variablen, die auf Module-level deklariert werden, sind global für dieses Module. Der Scope der Variable wird dann zur Laufzeit ausgewertet.

6.2.4 Channels

Channels erlauben die Kommunikation zwischen (remote oder lokalen) Prozessen bzw. Anwendungen. Ein Channel ist ein Speicherplatz für einen Wert mit definiertem Typ, der jedoch nur für Sende- und Empfangsoperationen verwendbar ist. Channel haben Referenz-Semantik. Es gibt folgende Operationen:

```
typ: type (int, string);    # Typ-Vereinbarung
x: chan of typ;             # Vereinbarung
x = chan of typ             # Erzeugung
x <-= (10, "hallo")         # Senden, Platz ist dann belegt
(i, s) := <- x              # Empfangen, Platz ist danach leer
```

Die Pointe besteht darin, dass das Senden und Empfangen in verschiedenen Threads erfolgen muss, denn beide Operationen blockieren, bis die Übertragung wirklich erfolgen kann, das heisst, bis sich ein Sender und ein Empfänger am Channel treffen. `alt { ... }` erwartet Channels in seinen Etiketten, kontrolliert den ersten in jedem Etikett und wählt einen Block, dessen Etikett nicht blockiert.

```
# watching several channels simultaneously for input and output
outchan := chan of string;
inchan := chan of int;
alt
{
    i :=<- inchan => sys->print("Received %d\n", i);
    outchan <-= "message" => sys->print("Message sent\n");
}
```

6.2.5 Multitasking

Multitasking (premptiv) wird erreicht durch `spawn`, welches einen asynchronen, unabhängigen Thread erzeugt, der den gleichen Speicher teilt. Synchronisation von Threads ist nicht explizit implementiert, jedoch lässt sich eine Synchronisation mit Hilfe von Channels, die gemeinsam auf globale Variablen zugreifen, simulieren.

```
# mon := Monitor.new()
# mon.lock()            # first call succeeds, subsequent block
# mon.test()            # returns true if attempt to lock successful
# mon.unlock()          # releases lock established by lock/test
# mon.destroy()         # terminates partner thread
Mon: module {
    PATH: con "mon.dis";
    Monitor: adt {
        ch: chan of int;        #  synchronizing channel
        ctl: chan of int;       #  to abort monitor thread
        fix: chan of int;       ## repair a bug
        new: fn (): ref Monitor;
        lock: fn (m: self ref Monitor);
        test: fn (m: self ref Monitor): int;
        unlock: fn (m: self ref Monitor);
        destroy: fn (m: self ref Monitor);
    };
};
```

Zur Synchronisation erzeugt man eine Instanz von Monitor:

```
implement Mon;
include "mon.m";
Monitor.new (): ref Monitor {
    result := ref Monitor(chan of int, nil, nil); ## Instanz erzeugen
    spawn main(result);                           #  Partner-Thread erzeugen
    return result;
}
```

Dabei entsteht ein Thread, der abwechselnd als Partner für `lock()`, `test()` und `unlock()` zur Verfügung steht und der durch `destroy()` terminiert wird. Leider muss man einen internen Fehler korrigieren (##): Wenn in zwei `alt` versucht wird, zum gleichen Channel zu senden, und wenn ein `alt` wie in `test()` vermeiden kann zu senden, versucht das andere alt anschliessend nicht mehr zu senden.

6.3 Module

Ein Limbo-Programm besteht aus der Interaktion von verschiedenen Modulen. Da Vererbung jedoch nicht unterstützt wird kann Limbo nicht als objekt-orientiert bezeichnet werden. Ein Modul besteht aus einer Deklaration (*.m-Datei), welches das Interface definiert, und einer Implementation (*.b-Datei). Dabei können die Interfaces verschieden implementiert werden. Der Limbo-Compiler generiert dann hardwareunabhängigen Byte-Code (*.dis-File). Alle Standard-Libraries wurden als Module implementiert. Einige davon wie `Sys`, `Draw`, `Math` und `Tk` sind sogar im Kernel integriert, da sie für fast alle Programme gebraucht werden.

7 Sicherheit

Sicherheit kann in 3 Bereiche unterteilt werden: Sicherheit der Verbindung, Resource Control und System-Integrität:

- Sicherheit der Verbindung wird erreicht durch (optionale) Verschlüsselung. Keys werden mit dem public-key Mechanismus (Diffie-Hellman) ausgetauscht und dabei werden symmetrische Mechanismen wie MD5, SHA, RC4 und DES angewendet.
- Resource Control wird durch anwendungsspezifische Namespaces erreicht, welche für andere Anwendungen restriktiert werden kann. Somit kann keine Anwendung Ressourcen ansprechen, die nicht schon im Namespaces sind.
- System Integrität wird durch die Dis VM erreicht, welche eine kontrollierte Umgebung für Anwendungen zur Verfügung stellt. Zudem können Anwendungen und Module mit einer Signatur versehen werden. Diese Signatur kann dann vom System vor der Ausführung überprüft werden.

7.1 Sicherheitslücken

Verschlüsselung und Authentifizierung ist zwar schön, ein Problem ist jedoch bei hosted Inferno zu bemerken. Da Inferno in diesem Fall auf das Hostbetriebssystem baut, können sich Sockets unter Druck so verhalten wie diejenigen des Hostbetriebssystem. Weitere Probleme gibt es auch dadurch, dass wenn man Inferno hosted über einem anderen Betriebssystem (Plan 9, Win 32, Unix,...) ausführt, wird der Emulator ohne Login gestartet und der User auf Inferno entspricht dem User auf dem Hostbetriebssystem. Insbesondere ermöglicht dies, dass man ohne einen oder mit einem beliebigen Usernamen eingeloggt sein kann.

7.1.1 Problem: Hosted Inferno auf Windows

Falls das Hostbetriebssystem Windows 95/98 ist, werden nicht einmal die Systemdateien geschützt. Und selbst wenn, kann man sich Zugriff auf das Windowssystem verschaffen. Dazu

muss man den default Pfad des Namespaces auf `C:/USERS/INFERNO` setzten. Nun vollführt man folgende Schritte:

```
stacey; cat /dev/user
inferno
stacey; mount tcp!jessica /n/remote
stacey; cd /n/remote/usr/dalai/keyring
stacey; lc
default
stacey; cp default /usr/inferno
stacey;
```

Jetzt kann man als `dalai` einloggen. Zudem können wir analog auch Zugriff auf das Passwort-File `/keydb/password` erlangen. Desweiteren gibt es mehrere Tools wie z.B. `clogon` - erlaubt den Usernamen einmal pro Sitzung zu wechseln - oder `hellfire` - Passwort-Cracker, welcher remote Zugriff ermöglicht.

7.2 Beispiel Server

Bei Systemaufrufen wie `open()`, `read()`, `write()` usw. verkehrt der Kern mit den Devices über Styx in Form von lokalen Funktionsaufrufen. Eines der Devices ist `#M`. Es kann nur vom Kern angesprochen werden und verwandelt die lokalen Funktionsaufrufe in Styx-Messages an Prozesse, für die `#M` Dateiverbindungen (`ref FD`) hält. `#M` erhält diese Dateiverbindungen durch den Systemaufruf `mount()`:

```
include "sys.m";
sys := load Sys Sys->PATH;
if (sys->mount(fd, "/n/fs", sys->MAFTER|sys->MCREATE, "") < 0)
    sys->print("mount failed: %r\n");
```

Das Kommando `mount` beschafft sich den nötigen `ref FD` als Netzverbindung zum Port `styx` (typisch 6666) eines Hosts:

```
perky$ mount -ac net!$FILESERVER /n/fs
```

Statt `$FILESERVER` kann auch ein DNS-Name angegeben werden. `mount` erlaubt allerdings keine explizite Port-Angabe. Die Flags sind identisch zu `bind`. Zusätzlich kann man mit `-C` einen Sicherheitsalgorithmus für das Secure Socket Layer verlangen: `md5` oder `sha` für Digesting, `rc4` für Verschlüsselung, in den USA auch DES. Das Gegenstück zum Systemaufruf `mount()` ist der Systemaufruf `export()` am anderen Ende der Dateiverbindung:

```
if (sys->export(fd, sys->EXPWAIT) < 0)
    sys->print("export failed: %r\n");
```

Diese Funktion stellt ihren Namensraum komplett an `#M` und von dort an dem bei `mount()` angegebenen Katalog zur Verfügung. Vor Aufruf wird man tunlichst mit `newns()` einen reduzierten Namensraum einrichten.

7.2.1 Authentifizierung

Da bei `mount` normalerweise zwei Systeme kommunizieren, ergibt sich die Frage der gegenseitigen Identifikation. Inferno löst dieses Problem mit der Certification Authority, das heisst, mit `logind`. Server und Klient sind der CA unabhängig voneinander persönlich bekannt und haben mit ihr heimlich je ein Passwort vereinbart. Mit Hilfe ihrer Passwörter beschaffen sie sich über das Netz mit `getauthinfo` von `logind` jeweils ein Zertifikat, das ihre eigene Identität verbrieft. Die Zertifikate werden in `keyring/default` beim Betreiber des Servers und `keyring/net!machine` beim Benutzer des Klienten gespeichert, der `mount` ausführen will. Beim `mount`-Kommando wird mit der Funktion `auth()` Information aus den Zertifikaten ausgetauscht. Die Funktion ist symmetrisch für Server und Klient:

```
include "keyring.m";
kr := load Keyring Keyring->PATH;
if (kr == nil)
    sys->print("cannot load Keyring module\n");
authinfo := kr->readauthinfo("/usr/axel/keyring/net!$FILESERVER");
if (authinfo != nil)
    (username, secret) := kr->auth(fd, authinfo, 0);
```

Der Server erfährt dabei den Namen, mit dem der Klient bei der CA identifiziert ist. Diesen Namen verwendet der Server als Grundlage der Zugriffsrechte für den Klienten.

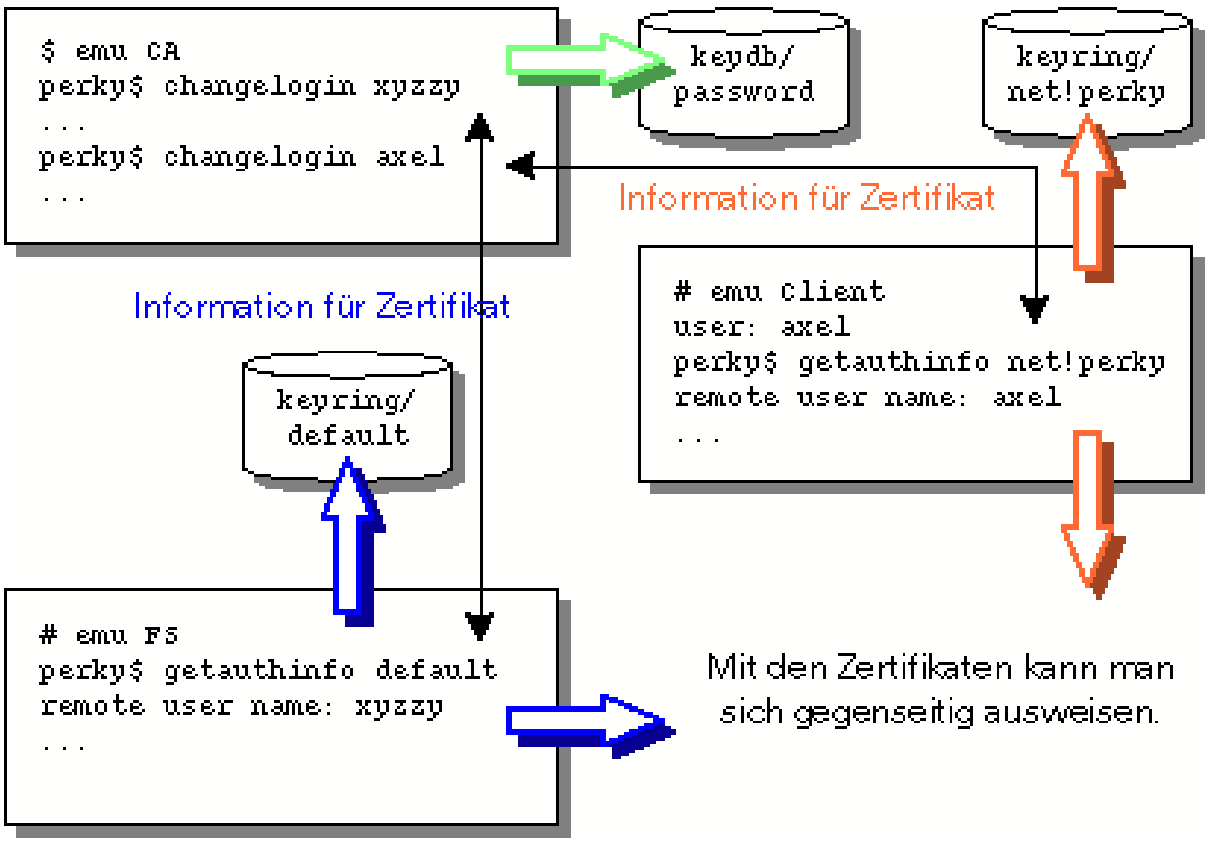

Abbildung 6: Authentifikation

7.2.2 Secure Socket Layer

Das Secure Socket Layer ist ein Filter, der im Stil einer Pipeline auf einen `ref FD` aufgeschaltet werden kann:

```
include "sys.m";
include "security.m";
sys := load Sys Sys->PATH;
if (sys->bind("#D", "/n/ssl", Sys->MREPL) < 0)
    ...
ssl := load SSL SSL->PATH;
(err, conn) := ssl->connect(fd);
if (err != nil)
    sys->print("cannot push SSL: %s\n", err);
else if ((err = ssl->secret(conn, secret, secret)) != nil)
    sys->print("cannot set secret: %s\n", err);
else if (sys->fprint(conn.cfd, "alg clear") < 0)
    sys->print("cannot set alg clear: %r\n");
else
    return conn.dfd;
```

`fd` ist die ursprüngliche Verbindung, das Resultat ist die sichere Verbindung. `secret` ist ein temporärer Schlüssel, der im Zuge von `auth()` beschafft wurde. `clear` ist vermutlich nicht der Algorithmus, den man verwenden würde.

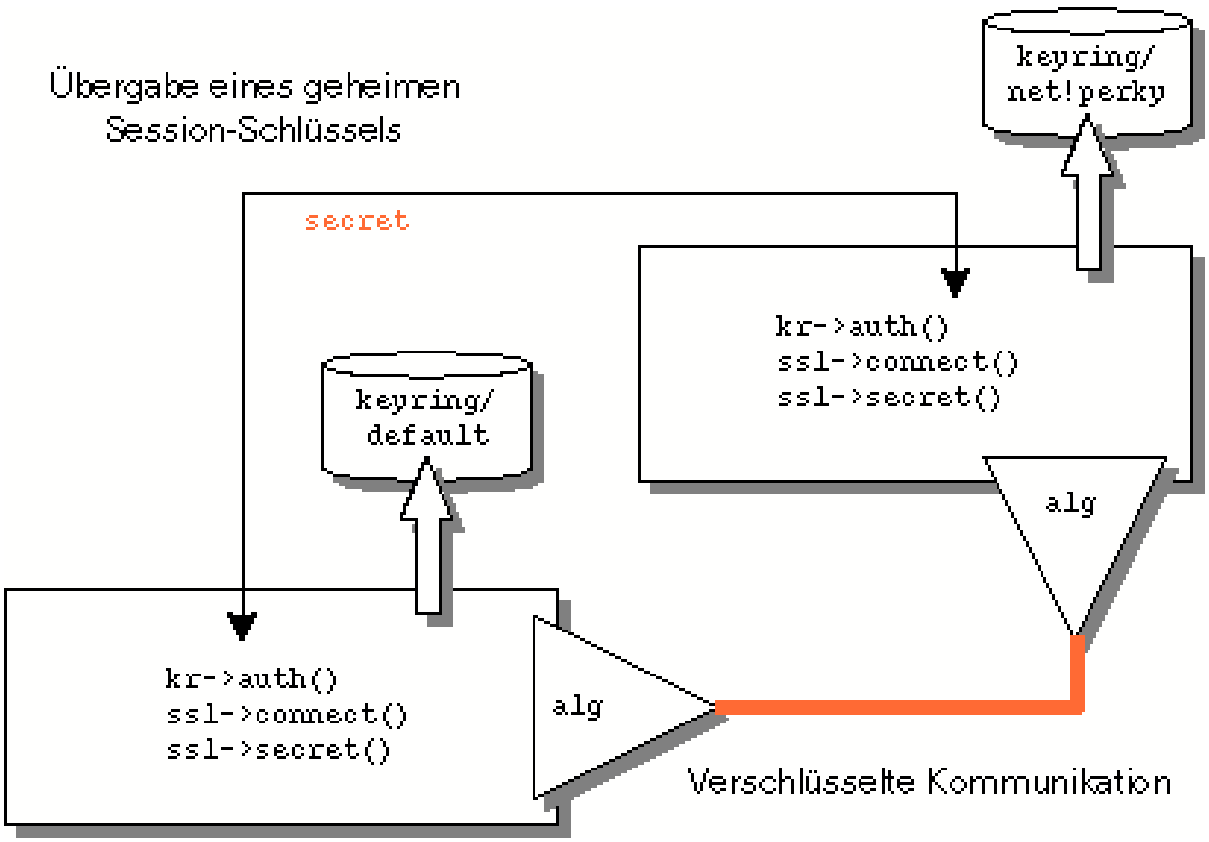

Abbildung 7: Secure Socket Layer

8 Vergleich

8.1 Überblick über verschiedene Systeme

	Erlang	Jini/Java	IBM MQSeries	XML/SOAP	General RTOS	Linux	Embedded Linux	Microsoft Net	Inferno	Corba
Virtual Machine	○	✓	○	○	○	○	○	✓	✓	○
C/C++Language Support	○	✓	✓	○	✓	✓	✓	✓	✓[1]	✓
Concurrent Programming Language	✓	✓	○	○	○	○	○	○	✓	○
Protocol for Distribution	○	○	✓	✓	○	○	○	○	✓	✓
Native OS	○	○	○	○	✓	✓	✓	✓	✓	○
Hosted Environment	✓	✓	✓	✓	○	○	○	○	✓	✓
Embedded Environment	○	✓[2]	○	○	✓	○	✓	○	✓	✓
Royalty Free Distribution	○	○	○	○	○	✓	✓[3]	✓	✓	✓
Source Code Visibility	✓	○	○	○	○	✓	✓	○	✓	○

Abbildung 8: Vergleich verschiedener Systeme

1. Build-in modules and devices drivers are written in C, but concurrent applications are usually implemented in Limbo.

2. In practice embedded Java environments require significant resources. the European STB standard which uses Java has a minimum specification of 16Mb RAM.

3. Many embedded Linux offerings charge per-piece royality fees for the OS and do not give access to the full source code.

8.2 Inferno/Limbo vs. JavaOS/Java

	Inferno/Limbo:	JavaOS/Java:
Security	Built-in authentication and encryption at OS level, but not automatic machine protection security.	Machine protection security is built-in. Encryption is being added.
Resource access	One file system accesses everything from data to network resources.	File system accesses local data. Network data must be accessed through the server.
Minimum Sized Machine to Run applications	512 kilobytes of RAM and 512 kilobytes of ROM.	512 kilobytes of ROM and 128 kilobytes of RAM
Object-oriented	No	Yes
Virtual Machine	Dis	Java Virtual Machine

Abbildung 9: Inferno vs. JavaOS

Für einen ausführlicheren Vergleich von Java, ActiveX und Inferno siehe *http* : *//www.cs.yale.edu/homes/asmith/pap_crit.html*.

Literatur

[1] http://www.vitanuova.com/inferno/

[2] http://www.vitanuova.com/inferno/papers.html

[3] http://www.cs.hut.fi/ kny/inferno/

[4] http://www.internetweek.com/101496/633first.htm

[5] http://www.cs.rit.edu/ ats/inferno/

[6] http://www.cs.unc.edu/ rajaram/reports/survey.html

[7] http://cs-www.bu.edu/ftp/pub/lctes98/lctes98-b4.ps.gz

[8] http://www.eecs.utoledo.edu/DocumentRepository/Thesis/VenuAndra/Andra-thesis.pdf

[9] http://choices.cs.uiuc.edu/2k/papers/MS-naming.ps.gz

[10] http://www.vorlesungen.uni-osnabrueck.de/informatik/inferno/

[11] http://casl.csa.iisc.ernet.in/OperatingSystems/Inferno/

[12] http://www.cs.rit.edu/ ats/inferno/

[13] http://www.vitanuova.com/inferno/papers/limbo.html

[14] http://casl.csa.iisc.ernet.in/OperatingSystems/Inferno/

[15] http://inferno.bell-labs.com/inferno/limbo.html

[16] http://www.cs.yale.edu/homes/asmith/pap_crit.html