VAMNET: the Functional Approach to Distributed Programming.

Dr. Stefan Bosse
BSSLAB - Independent Research Laboratory, Bremen, Germany

31 October 2005

Abstract

This article gives a design overview of a new reliable distributed operating system environment, combining the world of functional and distributed programming using a virtual machine approach for hiding system dependencies, offering rapid prototyping facilities. The basic operating system concepts used are derived from the Amoeba operating system by Andrew Tanenbaum and his work group, developed 20 years ago. VAMNET is not only a native operating system. It is a hybrid solution for expanding widely used operating systems like UNIX with a distributed execution environment.

Distributed operating system and programming, functional ML programming language, virtual machine environment, heterogeneous computer networks.

1 Introduction

1.1 Overview and Motivation

Distributed programming and execution environments play an increasing role ever since computers can be coupled in networks. But today, there are only two main solutions for distributed communication and programming: specialized hardcore distributed operating systems, like Amoeba [7] or Plan9 [2], or specialized programs and environments enabling distributed programming like PVM [4] and MPI as a hybrid extension to the underlying operating system.

A special operating system will probably give the best performance and the cleanest approach, but lacks standard software and enough hardware device drivers known from todays desktop operating systems. Especially the lack of device drivers in the i86-PC world is a hard constraint. While this article was written, several new hardware devices were designed and pushed to the market!

The second solution with the hybrid approach avoids these constraints. But most common solutions are not very well integrated in the underlying operating system concept, they can only provide special solutions. Furthermore, they mostly use the IP protocol family for interprocess communication, not very well suited for distributed programming. The main disadvantages are the user network configuration needed to communicate between computers and a lack of performance capabilities for distributed communication. Additionally, most hybrid solutions have a complicated and heavy weighted programming interface. Combining distributed and native operating system interfaces and methods is not fully supported.

Not enough, the native and the conventional hybrid approach have another serious drawback: for N system architectures (and operating systems in the case of a hybrid solution), N binary programs must exist. Each time a program is modified, it must be recompiled N times! The imperative C programming language commonly used in all programming environments results in best performance, but it is not today's solution for solving high level and abstract problems. C binary programs are system-dependent, and due to error-proned memory pointer arithmetics and buffer overflow problems, C programs are not reliable. Like most other imperative programming languages C provides no automatic memory allocation and deallocation, one barrier for rapid prototyping. The low abstraction level of C and missing modularization features can lead to an unstructured and inefficient program implementation in large-scale problems.

To avoid all these constraints, a new hybrid approach was chosen combining the following concepts and components:

1. Basic concepts from the distributed native operating system **Amoeba** [7], developed by Prof. Andrew Tanenbaum et al. from the Vrije Universiteit, Amsterdam. Especially the for distributed purposes only developed high performance **Fast Local Intranet Communication Protocol (FLIP)** [1] and the **Remote Procedure Call (RPC)** interface (layered on the top of FLIP), just consisting of three basic functions, together with the ingenious simple-to-use capability object concept is a good starting point to implement distributed programming concepts, not only inside the native Amoeba operating system. The basic concepts are layered around the client-server approach.

2. These Amoeba concepts were implemented using the **functional programming language OCaML** [5] from the INRIA institute, France. It is a ML dialect with an objective class extension (an extension concept comparable with C++ on the top of C) and an easy to handle module system. ML hides usually all memory access and allocation. It is a strongly typed language. This feature avoids common inconsistencies in algortihm and program implementation and leads to a more

structured and cleaner programming style. OCaML provides elementary functions for list, string, array, and other more complex data type manipulations like data tuples. There are a large number of modules implemented in ML providing Amoeba functionality.

3. ML programs are compiled into architecture and system-independent **bytecode**. This bytecode is executed using a stack based **virtual machine** (VM), a relatively small native binary program, written in C to give maximal performance and a clean and simple approach to extend the virtual machine with already existing customized C library code and routines. The virtual machine is responsible for the loading and execution of bytecode programs, for automatic memory allocation and deallocation using a garbage collection method. The OCaML garbage collector combines Stop&Copy and Mark&Sweep methods [10]. Together with (2), the **Virtual Amoeba Machine (VAM)** environment was built, combining functional and distributed programming features. Since source code can be compiled and executed on the fly, rapid prototyping is possible.

4. The native **VX-Amoeba Kerne**l. It is a standalone Amoeba microkernel providing basic Amoeba concepts for a raw Amoeba process environment: enhanced (compared to the original Amoeba) thread and process scheduling and control, device drivers, memory management, FLIP, RPC. Currently only the i86-PC system architecture is supported. It's therefore best suited for hardware reduced embedded systems, like the PC104 technology widely used in industrial controlling applications. Unlike UNIX systems the VX-Kernel requires no root filesystem located on disks.

5. The underground communication layer consisting of an enhanced version of FLIP and RPC is implemented both inside the VX-Kernel and on the top of existing operating systems in hosted mode, for example UNIX like systems like Linux and FreeBSD. The latter case is realized with the so-called **AMUNIX** glue layer, written entirely in C. It consists of: an Amoeba thread implementation **AMUTHR**, fully compatible with the native Amoeba-Thread system located inside the VX-Kernel, the client part interface for RPC, several Amoeba basic and utility functions, like the capability management and server stubs. The FLIP protocol stack was restructured and divided into a (small) system-dependent and in a main system-independent part. Therefore, most source code can be shared between the VX-Kernel and the AMUNIX implementation. Under AMUNIX, FLIP is executing as a generic user space process.

   The VAM virtual machine program is implemented on the top of the AMUNIX- layer, mainly using RPC and AMUTHR.

6. Many **Amoeba system servers** were implemented in ML using VAM. They can operate using both the raw VX-Kernel and the AMUNIX system without recompilation enabling rapid and reliable prototyping. Examples for servers are the new Amoeba Filesystem AFS and the directory and name server DNS. These servers either use local image files or access physical partitions to store file data. Furthermore there is a boot server responsible for starting and controlling a completely or partialy distributed environment.

The advantages of this hybrid system and its components are:

- The FLIP protocol operates connectionless and determines network routes automatically. The FLIP protocol stack acts always as a router between networks[1]. There is no necessity for user configuration. FLIP and RPC communication takes place between processes independent of their loaction. FLIP is not limited to the Ethernet technology. Any kind of data transfer media can be used for networking (USB, point-to-point devices ...).

- For special (embedded) hardware systems, like PC104 industrial computer boards, the raw VX-Kernel environment is best suited. They can be used to build up distributed measuring and control services with direct support of special hardware devices connected to a network. The VX-Kernel provides an unique and simple device driver interface, both inside and outside the kernel (user space processes), using the common RPC communication, too.

- Common desktop operating systems can be still used, extended with Amoeba concepts. The Virtual Amoeba Machine provides full support for Amoeba and UNIX programming interfaces. If VAM operates on the raw VX-Kernel, the UNIX part is currently restricted.

- VAM hides architecture and system dependencies. Only the virtual machine must be compiled for each target architecture and host operating system.

- There are different ways of executing VAM bytecode programs: directly from a UNIX shell on the local host, or using the VAX util for remote program start on a native VX-Amoeba host, and finally (under development) on any host (native Amoeba or UNIX, both local and remote), using Amoebas process descriptor technology and the virtual machine itself acting as a process server (see implementation section for details).

- The functional programming language can simplify and speed up software development (see section 2.4 for a detailed explanation).

All these components are loosely coupled and can be combined depending on customer demands. With this hybrid operating system environment it is possible to build up distributed measuring and control systems and parallel numeric clusters in an uncomplicated and simple way with good scalability, or it just provides a convenient way for research and education in distributed programming. Parallel clusters can be built using both generic desktop computers and specialized computer hardware (bare bone computers with only CPU, memory and network) [11].

A comparable approach of such a hybrid system was already introduced with Bell Labs Inferno and Plan9 operating system environment. Plan9 has some similarities with Amoeba, though it is more UNIX-orientated using a generic file concept rather than an unique object concept. Inferno uses a virtual machine approach with automatic memory management, too, and can operate stand-alone or on the top of existing host operating systems [12]. The new programming language Limbo developed for Inferno inherited some main features from ML, like strong typing and limit and bound checking, and a module system, but the functional concept of programming is entirely missing. And the Inferno approach is more monolithic than the VAMNET approach. VAMNET components are loosely coupled and can be combined like a modular construction kit, in contrast to Inferno, which needs a full-sized execution environment on each computer node.

Plan9 and Inferno use a user orientated security approach, different from Amoebas more universal object based and user independent capability approach. Additionally, Plan9 and Inferno use a different kind of communication network protocol, called 9P/Styx. This protocol approach is focused to file objects similar to NFS, and is IP based with the necessity of knowledge about network topology and network configuration [3].

1.2 The Complete System

The following pictures 1 and 2 show all the components contained in the hybrid VAMNET system and introduced in the previous section, both the native Amoeba system and the addon approach, executing, for example, on the top of the Linux operating system.

On the native Amoeba side, the protocol stack FLIP, the RPC layer and process and thread management is located inside the VX-Kernel, for example executing on a PC104 embedded system. The kernel can be booted from an EEPROM-like medium, for example an IDE compatible CompactFlash card. On the desktop operating system side with a UNIX host operating system, the FLIP protocol stack is implemented as an external process. Each AMUNIX and VAM process has its own thread management (AMUTHR). AMUNIX processes must communicate with the FLIP protocol stack using UNIX sockets. The VAM application programs can be executed both on the native AMOEBA and the AMUNIX environment without recompilation.

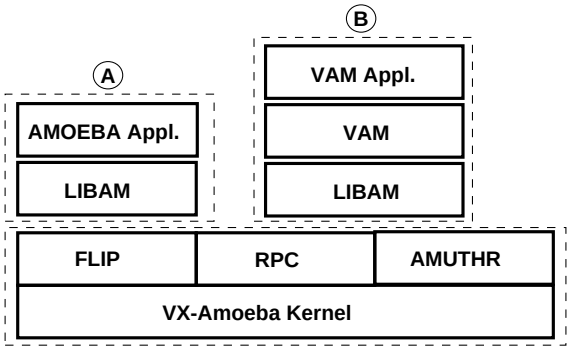

Figure 1: The native Amoeba operating system side with two application programs, one native (A) and one bytecode (B).

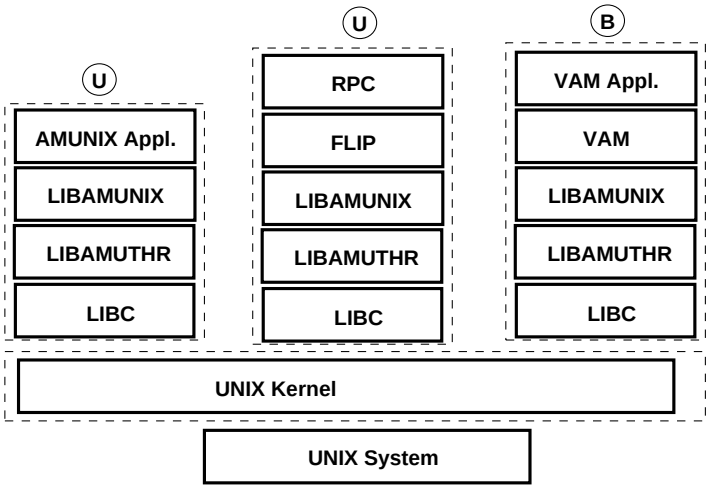

Figure 2: The UNIX operating system side with addon layers AMUNIX + VAM software with two application programs, one native (U) and one bytecode (B), and the FLIP server process in the middle.

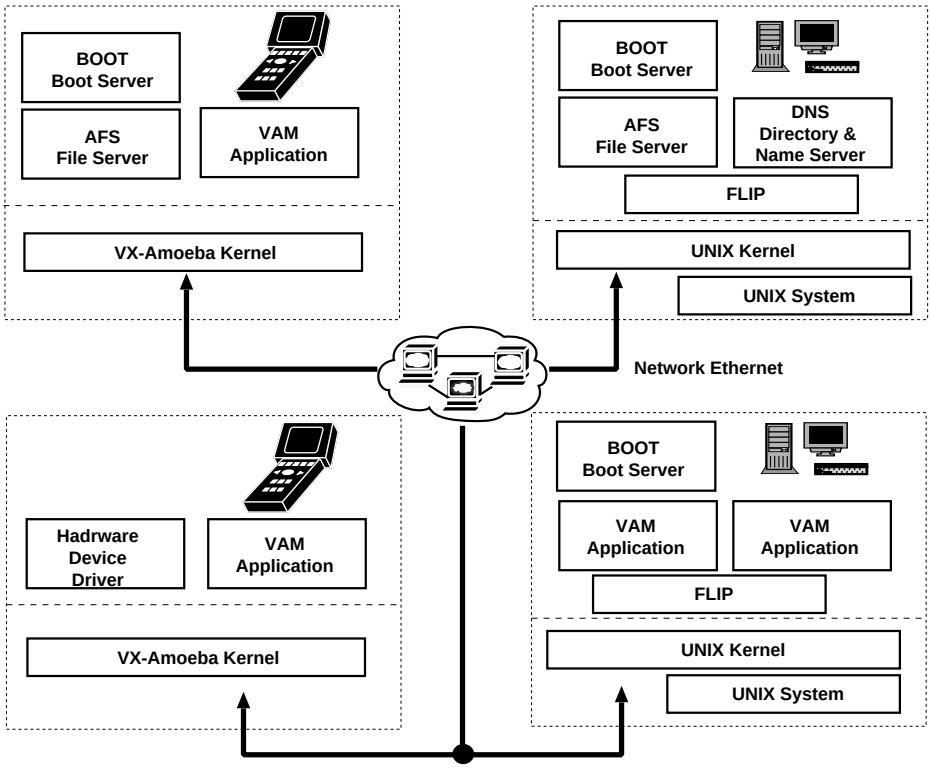

Figure 3: Several native Amoeba and UNIX nodes connected by a network with different application and system programs.

Figure 3 shows native Amoeba and UNIX nodes connected with an Ethernet network with distributed filesystems and some VAM application processes. All computer nodes melt to one large virtual machine system. The programmer and user should not be aware of the location of an Amoeba program. The location dependencies are resolved exclusively by FLIP.

.

2 Implementation Details

2.1 Objects and Communication

The main communication method is the remote procedure call (RPC) using the message-based client-server-approach. Amoeba needs only three primitives to provide this method: the '*transaction*' function for clients, and the '*getrequest*' and '*putreply*' functions for servers. A server can be any process simply by creating a so-called server port, and calling the *getrequest* function. A public version of this port (created by a one-way-encryption method) is published and must be known by a client who calls the *transac-*

tion function to send a message to a server, stored in a generic data buffer together with a communication header. Because Amoebas RPC operates synchronously, a reply is sent back to the client. Requests and replies are symmetric functions. Both operate with a generic data buffer up to 1GByte size and a communication header with additional information.

Around RPC, the object capability concept plays an important role. A server manages objects, for example files, processes or terminals, and each object is associated with a capability. This capability contains the server port (public version), an object number, a rights field specifying allowed operations with this object, and a security field for protecting the rights field against modification. The RPC communcation layer is stacked on the top of the FLIP protocol stack. The FLIP protocol stack is implemented directly inside the VX-kernel. Processes running on the top of this kernel simply call kernel functions in order to get access to RPC/FLIP. FLIP is partially weaved with the process management (some process signals can be transmitted using FLIP).

In the UNIX environment, this method is not desirable because of the strong operating system dependency this method envolves. Here, another way was chosen: the FLIP protocol stack is entirely implemented in user process space. Clients wanting to send messages or listen for requests connect to the FLIP box process using generic UNIX sockets. Each process thread doing RPC transfer has its own socket link. On the other side, the FLIP box is connected to the network layer of the host operating system using either raw sockets (Linux) or the packet filter interface (FreeBSD).

2.2 AMUNIX

The AMUNIX layer consists of:

- the AMUNIX glue library with the AMUTHR module implementing native Amoeba threads for UNIX user processes with identical programming interface and behaviour, additionally some basic Amoeba functions and server stubs (predefined RPC requests for various servers and commands), and the client part of the RPC interface (adapted to the AMUNIX FLIP socket concept),
- the FLIP protocol stack daemon program,
- a development environment,
- some C-based programs derived from the original Amoeba operating system, like *aps*, the Amoeba process displayer, and some utilities for standard commands. They are not needed for the distributed environment explained here.

With the AMUNIX environment it is possible to execute Amoeba programs (recompiled using the AMUNIX library) directly under the UNIX operating system.

2.3 The VX-Kernel

This native microkernel based on the kernel present in the original Amoeba system, finished in the year 1996 [8]. The kernel provides the following services:

- Thread, Process and Memory management (segment-based),
- Device drivers (network, storage, display),
- FLIP protocol stack, RPC and group communication layer,
- several system servers accessible from the outside using RPC.

In the past seven years, the kernel was restructured and improved. Several new features were added:

- Support of user space device drivers was introduced.
- A new priority-driven scheduler was implemented: still strictly non-preemptive scheduling policy inside the kernel, but with three levels of process and thread priorities: HIGH, NORM, LOW. The process priority has a higher weight than the thread priorities. Hardware event-driven threads and high level interrupt handlers always have the highest priority and are scheduled first. Processes have a time slice, and can be scheduled preemptively. User process threads can be scheduled optionally preemptively, too, if they are not actual executing in kernel mode.
- A local high performance interprocess communication module for local user space device drivers and kernel communication. The communication primitives are similar to the ones known from the RPC interface. But in contrast they support shared memory segments for ultra fast 'data transfer'.
- More bus systems (PCI) and hardware devices (network) are supported on the i86-PC architecture. 100 MBit/s ethernet is provided, and support for USB communication is under development.

Together with the Amoeba system library it is possible to directly execute native Amoeba binaries on the top of the VX-Kernel. No additional external servers are required. There is a cross-compiling environment **AMCROSS** to build native Amoeba libraries, kernels and programs.

2.4 VAM and ML

The Virtual Amoeba Machine consists of these parts:

- the virtual machine itself linked with either the AMUNIX layer or the AMOEBA system library - the VM is built around a library concept and can be easily extended and rebuilt,
- all OCaML standard modules, Amoeba modules implementing all Amoeba concepts (except low-level threads and RPC) entirely using ML: all functionality around capabilities, standard operations, buffer methods, encryption, several server stubs, process execution and many more. With these modules it is possible to build distributed systems. Additionally, an UNIX module exists to provide all UNIX system calls independent of the underlying operating system. VAM provides an abstraction of the underlying hardware resources.
- of course a ML compiler, written entirely in ML, too (therefore portable and system-independent), which can be used (and embedded) for on-the-fly compilation during runtime,
- an interactive toplevel system used mainly for ML script execution (the toplevel system contains the ML compiler),
- many system servers implemented in ML and utility programs needed for system administration, building an operable distributed operating system environment,
- a complete development environment,
- and last but not least a graphical user interface based on the X11 system (XLIB/VXLIB).

OCaML belongs to the class of functional languages, which means that functions are first-class values, and they can be treated like any other data type. ML is not purely functional because it includes assignments and side-effects. ML is a strongly typed language. This is an important feature for reliable programming and systems. ML programs are safe in some kind: no illegal state instructions or memory access faults can occur, and array, list and buffer bound checks are performed during runtime. Moreover, ML uses type inference, which means that the ML compiler determines data types during compilation and there is no necessity for an explicit type declaration of values. OCaML is a fast execution environment, commonly faster than native compiled C++ code, and even faster than JAVA-Code [13] (though these benchmarks compare different fruits). Because the OCaML compiler produces operating system and machine-independent bytecode, OCaML programs are highly portable (though there is a native compiler version, too).

Just a recompiliation of the virtual machine is necessary for a new operating system or machine environment, maybe with slight changes in source code. The virtual machine itself is entirely written in C to facilitate its adaption to new environments.

The success of the implementation of operating system concepts using only ML depends strongly on the kind and structure of the operating system. For example, UNIX-like operating systems consist of a monolithic structure. Nearly the complete system functionality is managed inside the kernel. For each system function there is one kernel system call. A ML implementation of such operating system concept just leads to wrapper functions going through the virtual machine. But in contrast, message based systems like traditionial microkernels, for example the VX-Amoeba kernel, require only a few message-passing functions. Operating system functionality is implemented here mostly with server processes outside the kernel. It is a simple job to implement servers, like a fileserver, entirely in ML. Only few message-passing functions will pass through the virtual machine to the underlying operating system kernel.

Additionally, there is an object orientated approach which extends the ML core. The OCaML programming environment has some more advantages over traditional imperative programming languages like C. First of all, OCaML provides a powerfull but simple to use module system which provides strong and clean modularization and abstraction, and therefore enables scaling software to large sizes. And even if a microkernel approach is used: operating systems (and services) always tend to enlarge from an initial thousand line block up to millions of source code lines during development. It seems this is a natural law.

With a pure functional- and recursive-based approach together with lists, it is difficult to implement system programs like servers with a lot of repetitive server loops and data tables. Apart from providing first-class funtions, lists and recursion, OCaML supports imperative constructs such as loops, references and arrays. Finally, it supports polymorphic type inference, a prerequisite for abstraction and modularization [9].

2.5 Distributed Process Execution

The basic Amoeba concept of process handling is built around the so-called process descriptor (PD) [7]. Initially the PD is present in the binary file. It contains information about the target architecture, the owner capability (empty as long as not running), and a segment descriptor (SD). This descriptor contains information about all memory segments currently present in the process. In the case of the initial binary file, these are the text, data and stack segment. During runtime, more memory segments can be added. Additionally, the PD contains a thread descriptor giving information about the currently present process threads. A new process is started simply by

reading and passing the process descriptor to the process server, for example inside the VX-Kernel. The kernel process server reads all memory segments from the fileserver and starts the process using the PD informations. Because the stack segment contains the process environment (string and capability environment variables), it is created temporarily and read from file server, too. The stack initialization must be done by the client program which starts the process initially, for example a boot server.

This process concept differs from UNIX like operating systems. Therefore, the PD concept can not be used directly in the UNIX environment. But an AMUNIX binary can be directly started on the local UNIX host (native code, compiled only for this architecture), for example from a shell, and VAM bytecode programs executed by the VM, too. AMUNIX/VAM programs need no special runtime environment.

To enable starting of native VX-Amoeba binaries or VAM bytecode programs on a remote VX-Kernel, the **VAX** tool was implemented. This program does all steps necessary for starting PD based programs on a native Amoeba host. The binary is either loaded from an already running AFS/DNS filesystem (the original way), or with some hooks from the local UNIX filesystem. In the latter case, the VAX tool provides a temporary file server which maps UNIX to Amoeba file behaviours. Optionally, a virtual terminal server is invoked to display program output on the UNIX console, from which VAX was started.

But to this point a main feature is missing: starting bytecode programs on any host (Amoeba or UNIX) from any host in the same way. The solution of this problem is quite simple: a process server must be added to the virtual machine and on each host at least one virtual machine must be started in the so-called process server mode. In this case, the VM waits for a process execution request following the PD guidelines. Each bytecode program is wrapped with a process descriptor containing only two segments: the bytecode segment (unmodified output from the ML compiler), and a stack segment, filled with the process environment in an architecture-independent way. This PD is sent to the process server inside the VM, and the VM loads the bytecode from the fileserver in the same way as the server inside the native VX-Kernel. In contrast to the kernel process server, the VM can only execute one program at the time. Therefore, enough VMs must be started in process server mode, for example by a boot server executing on this host.

2.6 Portable Graphical User Interface

The graphical user interface for VAM uses the X11 protocol and is implemented entirely with ML, too. No native X11 system library is invoked. There are two client libraries: the low level client X11 core library implementing the X11 protocol and client-server communication [6], and a high-level widget library **VXLIB** built around classes. The fundamental methods

Machine	Components
A	AMD-Duron 650 MHz CPU, 64MB RAM, 3COM905 100MBit/s Ethernet,**native VX-Amoeba**
B	Celeron 700 MHz CPU, 64MB RAM, 3COM905 100MBit/s Ethernet, **native VX-Amoeba**
C	Celeron 700 MHz CPU, 128MB RAM, 3COM905 100MBit/s Ethernet, **FreeBSD 4.6 + AMUNIX**
D	Celeron 700 MHz CPU, 128MB RAM, 3COM905 100MBit/s Ethernet, **FreeBSD 4.6 + AMUNIX + VAM**

Table 1: Machine configurations used for performance tests.

are widget containers stored in boxes with either horizontal or vertical alignment, similar to TEXs typesetting boxes. All common widgets like buttons or text labels are supplied. The VXLIB supports two graphic output devices: X11 and postscript. Therefore all windows can be directly printed into postscript format, usefull for documentation and scientific GUIs, for example data plots or tables.

The X11 and the VXLIB parts are both system-independent. There are two communication standards supported: UNIX sockets and Amoebas virtual circuit technology. Virtual circuits are global (distributed) circular buffers. Virtual circuit modules are part of the basic VAM system. The first kind is used on UNIX-like systems, the second for an embedded and system-independent implementation of the X11 server called VXS (currently under development) - of course programmed in ML. The VXS program is connected to a low level graphics engine - here programmed with C for performance reasons and with system dependent parts (graphics driver). VXS will operate both on the VX-Kernel and with AMUNIX (here with a dummy graphics engine with a generic native X11 interface) for test and evaluation purposes.

3 Performance - the Real Life

3.1 RPC Network Performance

RPC messaging takes place between processes, either locally or remotely. The measured data transfer rates and the latency relate to realistic endpoint communication. The test configuration used is shown in table 1.

Transfer	**Data rate**	**Latency**
A → B	11,2 MByte/s	130 μs
B → A	10,6 MByte/s	136 μs
A → C	8,7 MByte/s	270 μs
C → A	9,1 MByte/s	260 μs
A → D	8,7 MByte/s	300 μs
D → A	8,7 MByte/s	300 μs
A → A	136 MByte/s	12 μs
C → C	26 MByte/s	275 μs
D → D	22,4 MByte/s	400 μs

Table 2: RPC performance tests. Measured data are mean values (accuracy ±10%). The latency is the time for one transaction without data load (only communication header).

The RPC performance was tested using different machine and operating system configurations, for both the local and the remote communication case. Data frames (size 1 MBytes) were sent to a server (in only one direction with data load) using the client transaction function (though a reply without data load is always present). Table 2 shows the results of RPC data rates and zero data load RPC transaction latency for each transfer direction.

The shown measurements are example measurements with an accuracy of about ± 10%. The transfer performance of a RPC message transfer between two native Amoeba machines (A) and (B) reaches its maximal value. Not only compared with the following AMUNIX and VAM system addons, also compared with the maximal physical transfer rate possible of 100MBit/s ethernet: 11,9 MBytes/s. This result reflects the optimal adaption of the FLIP protocol stack to the underlying ethernet device drivers.

Communication between the AMUNIX glue layer (C) with a native VX-Kernel (A) leads only to a slight decrease in performance and latency. The transfer rate decreases about 20%, and the latency increases about twice. With additonal VAM (D), there is no significant difference. These results show the suitability of the ML programming language and the virtual machine concept for client-server implementations.

Using the RPC system for local communication gives best results with the native VX-Kernel. The data transfer rates decreases dramatically using the AMUNIX layer. This is not a surprising result because all data must be transferred from the client to the FLIP process, and from FLIP to the server process. In contrast to the native FLIP implementation, data must be copied several times (with several kernel system traps). But the additional VAM layer and the virtual machine concept add no significant decrease in performance.

	Bullet	AFS
Memory usage	8.8 MByte	10 MByte
Source code lines	7400	9100
Transferrrate	28.6 MByte/s	9.9 MByte/s

Table 3: AFS (ML) and Bullet (C) file server performance. The memory usage includes the (large) file cache.

3.2 Server Performance

Reference [13] shows benchmark results for various programming languages using different (mostly short) test code programs. The native version of OCaML is mostly located under the top ten winners and its speed comparable with C++ programs, usually 3-5 times slower than C programs. But the bytecode version seems to be 10-30 times slower than C code. A deeper look into the used benchmark source code leads us to expect that these benchmarks are not very realistic for performance considerations in real distributed (server based) applications: the C code consists for example of some 'for loops', and the ML code, too. No functional features, list support, automatic memory management and other higher-level functionality is concerned. The following application orientated performance comparision between C and ML gives a more suitable impression of VAM application performance.

A file server, like the original Amoeba Bullet file server (entirely written in C and highly optimized for read transfer operations) and the new AFS file server (written entirely in ML and with equal behaviour) are examples of complex application programs with list, table and a lot of memory management functionality needed during runtime operation, for example inside the cache management.

The test configuration consists of a small embedded PC with a 300 MHz Geode low power microprocessor, 64 MByte RAM memory and a 256 MByte CompactFlash Harddisk replacement, operating with the native VX-Amoeba kernel, using the AFS server (ML, fully interface compatible with original Amoeba Bullet file server) and the Bullet Amoeba file server (C, enhanced version), both executed on the top of the kernel. Previously, one large file (1MByte size) was created in both filesystems. A test application, also running on this machine, reads this file multiple times. The time was measured and the mean transfer rate was calculated. Table 3 shows the results. The ML bytecode version is only 3 times slower than the highly optimized C version of this filesystem implementation, though runtime bound checking of arrays and buffers inside the virtual machine was enabled.

The number of source code lines of both implementations is comparable, though the AFS fileserver contains a more complex and efficient cache

management.

4 Conclusions

It could be shown that the presented hybrid operating system approach is suitable for a variety of applications requiring reliable distributed control and programming features, like distributed controlling services with embedded computers or parallel numeric clusters. VAMNET supports rapid prototyping because 1.) source code can be compiled and executed on the fly, and 2.) bytecode programs can be executed on both, the native VX-Amoeba and AMUNIX environment, regardless of the underlying hardware and host operating system as a result of the virtual machine concept. The loss of performance (commonly a factor 3-5 compared with C programs) is mostly negligible compared with the advantages of functional programming and the virtual machine approach: type safety and inference, modularity, runtime bound checking, host operating system independency.

References

[1] M.F. Kaashoek, R. van Renesse, H. van Stveren, A. S. Tanenbaum
FLIP: an Internetwork Protocol for Supporting Distributed Systems
ACM Transactions on Computer Systems, pp- 73-106, Feb. 1993

[2] R. Pike, D. Presotto, K. Thompson, H. Trickey
Plan9 from Bell Labs
UKUUG Proceedings of the summer 1990 Conference, London, England, (Jul. 1990)

[3] R. Pike, D. Ritchie
The Styx Architecture for Distributed Systems
Bell Labs Technical Journal, Vol. 4, No. 2, pp. 146-152, (Apr./Jun. 1999)

[4] V.S. Sunderam
PVM: A Framework for Parallel Distributed Computing
J. Concurrency, Practice and Experience, 1990, pp. 315-340

[5] *The OCAML language*
http://caml.inria.fr/

[6] Fabrice Le Fessant
xlib for Ocaml
projet Para/SOR, INRIA Rocquencourt, 1998

[7] S. J. Mullender, G. van Rossum, A.S. Tanenbaum, R. v. Renesse, H. van Staveren
Amoeba - A distributed Operating System for the 1990s
IEEE Computer, 14:365–368, May 1990

[8] A. S. Tanenbaum, M. F. Kaashoek
The Amoeba Microkernel
Distributed Open Systems, IEEE Computer Society Press 1994, pp. 11-30

[9] Jason Hickey
Introduction to the Objective Caml Programming Language
2001

[10] Emmanuel Chailloux, Pascal Manoury, Bruno Pagano
Developing Applications with OCaML
O'Reilly France, 2000, prelimenary english translation

[11] Stefan Bosse
The VAMNET Book - the virtual Amoeba Machine Environment, AMUNIX and the VX-Amoeba System
BSSLAB, http://www.bsslab.de, 2005

[12] Sean Dorward et al.
The Inferno Operating System
Bell Labs Technical Journal, Vol. 2, No. 1, 1997

[13] *The Computer Language Shootout Benchmarks*
http://shootout.alioth.debian.org, 2005