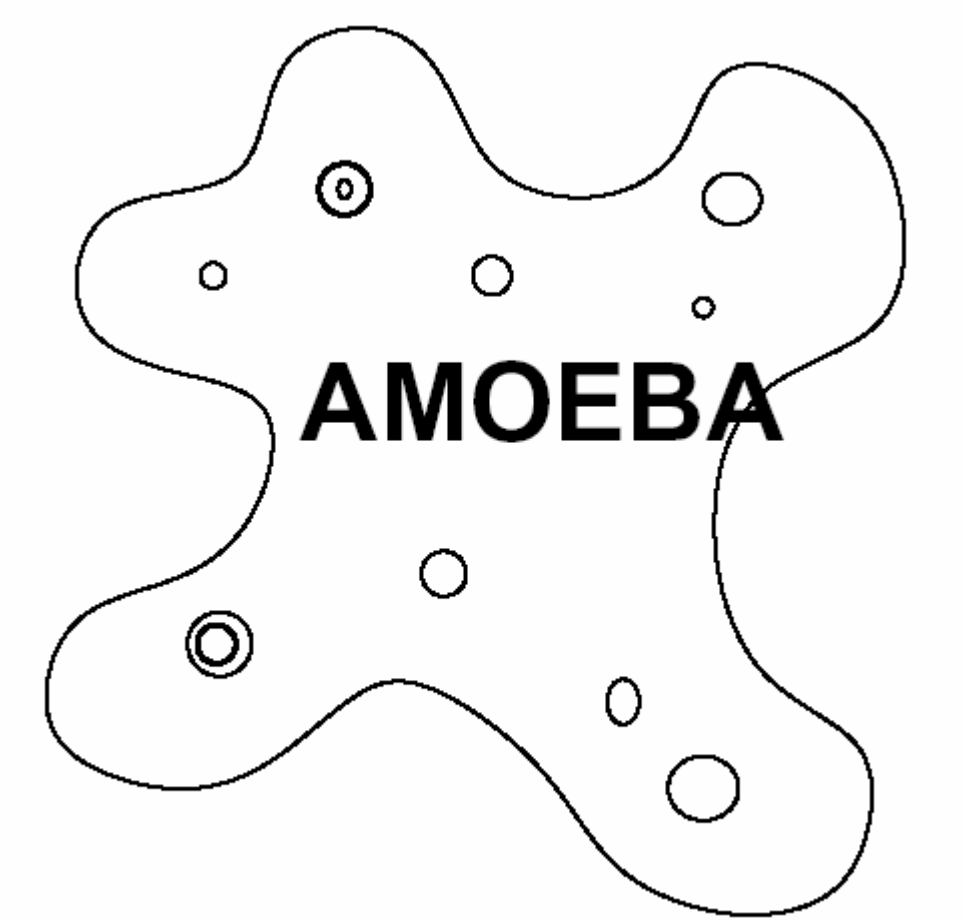
Amoeba Distributed Operating System



Matt Ramsay
Tim Kiegel
Heath Memmer

CS470
Case Study Paper
4/19/02

Amoeba

Introduction

The Amoeba operating system began as a research project at Vrije Universiteit (Free University) in Amsterdam. The research effort was aimed at understanding how to connect multiple computers together in a seamless way. The basic goal of Amoeba was to provide users with the capabilities of a single powerful timesharing system while actually running their jobs on a collection of machines, potentially distributed over several computing sites. This paper will examine the characteristics of the Amoeba operating system, give a diagram of its structure, give the pros and cons of its implementation and present our personal reactions to its features.

History of Amoeba

The initial research done in Amsterdam led to the design and implementation of the Amoeba Distributed Operating System, which was then used as a prototype and a vehicle for further research both at Vrije Universiteit and at the Center for Mathematics and Computer Science, also in Amsterdam. The goal of this further research was to build a distributed system that is invisible to users. [5]

The concept of an invisible distributed system can be illustrated by contrasting it with a network operating system, in which each machine has its own identity. In a network operating system, each user logs into a specific work station or machine, his home machine. When the user executes a program it is run on his local machine unless he gives an explicit command to execute it on another machine. In this situation, the user is definitely aware that multiple machines exist and has to deal with them explicitly in order to utilize each one.

In contrast, in a transparent distributed system, users log into the system as a whole, not into any specific machine in the network. When the user runs a program, the system decides which machine in the system should execute it. This decision is invisible to the user. All the user sees is that his program has been executed, just as if he had run it on his local machine. Also, there is a system-wide file system. Files that share a directory may be located on different machines anywhere in the network. There is no concept of file transfer, uploading or downloading from servers, or mounting remote file systems. A file's position in the directory hierarchy has no relation to its physical location.

Since the initial implementation of Amoeba, it has been used sparingly throughout the world, mostly in as a topic for continued research in the distributed systems area. Although Amoeba is not widely used, it is still very interesting to study because it points out the characteristics of a distributed computing system.

Main Objectives of Amoeba

The basic design goals of Amoeba are:

- Distribution – Connecting together many machines
- Parallelism – Allowing individual jobs to use multiple CPUs easily
- Transparency – Having the collection of computers act like a single system
- Performance – Achieving all of the above in an efficient manner

As outlined earlier, Amoeba is a distributed system in which multiple machines can be connected together. Even more important, these machines do not have to be all of the same kind. Most of the time, these multiple machines are spread throughout the same building, connected through Amoeba's high performance FLIP

(Fast Local Internet Protocol) network protocol over a LAN. However, Amoeba machines can be located anywhere across the world and the system still works in the same way. [5]

In addition, Amoeba is also a parallel system. This means that a single job or program can use multiple processors or machines to gain processing speed. This feature makes Amoeba capable of efficiently utilizing older and slower hardware as part of its machine pool. An Amoeba system allows a company to purchase several discarded machines for very little cost and have them perform with equal efficiency to a much more expensive new machine.

The third key goal of the Amoeba operating system was transparency. In an Amoeba system, the users do not need to know the number or location of the machines in the system. Similarly, the user also does not know where the files are located throughout the system. In this type of system, there is no concept of a “home machine”. Once a user logs into the system, remote login and remote mount operations are not required to access files and CPU’s of remote machines in the system. The whole system appears to be a single conventional timesharing system.

Efficient performance is obviously a goal of Amoeba, as it is of any operating system. Amoeba has an optimized communication system to manage messages and data transfers between machines in the Amoeba system. This communication system serves as the basis for implementing high performance subsystems and applications on an Amoeba system.

Overview of System Structure

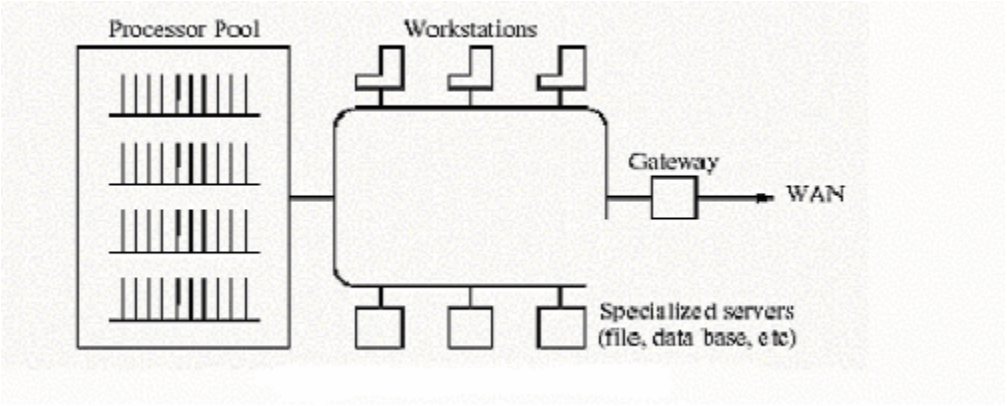


Figure 1: System Structure [3]

Computing is said to be distributed when the computing and the data it manipulates is spread across many machines[4]. This is the basic fundamental which Amoeba is built upon. The operating system breaks up the processes which need to be run and distributes them to multiple machines across a network. This creates parallel computing and allows for more information to be manipulated in a shorter time than it would take for a single processor system.

Amoeba achieves this distribution by breaking the OS into four main parts. The first is the workstations, which can range from simple dumb terminals to a part of the processor pool. Second is the main processor pool, which consists of multiple machines connected by a LAN. The third is the specialized servers, which examples are file servers, directory servers, and base servers. The fourth key component is the WAN gateways, which allow Amoeba to run across a WAN and perform distance distributed computing.[4]

Every machine running Amoeba runs the same microkernel. This is because the kernel is very simple and takes care of things like managing processes and threads,

providing low-level memory management, thread communication and I/O. The only difference is that some machines may run a server process, where other machines, like a workstation may run just the kernel and a shell.

Amoeba workstations may be configured in many ways. The simplest workstation is a dumb terminal. This can be either a text-based terminal or an X-windows terminal. The workstation can also be a full machine which can run the Amoeba kernel and an X-server[4]. This is called a smart terminal because the terminal does some processing of its own. In this mode, the workstation machine may also be a server, in the process pool or both. This is not recommended, but in smaller configurations such as testing setups, this may be used.

The processor pool consists of a multitude of machines which are connected through Ethernet. This pool may be heterogeneous, consisting of the following architectures: Motorola MC680x0, Intel 386(486, Pentium) and some members of the SPARC family[2]. These machines run the Amoeba kernel which may be booted from the hard drive, a boot server, or floppy. These machines may also run servers. This is not suggested, but may be used in special cases.

The specialized servers handle services which are needed by the system. Common services include file, directory, memory and process distribution[4]. These servers are dedicated to being only servers and are not used for processing data. Each server machine may run multiple servers from it. This may be useful if you have a small file system or run few processes.

The WAN gateways are used to separate the Amoeba system from other networks and their protocols, as well as allow distant distributed computing. The

gateway server is responsible for the communication scheme to allocate processors from a distant Amoeba pool for its own use. This is done by translating the Amoeba RPC packets to TCP/IP (for Internet) and transmit them to the remote server[4].

Processor Scheduling

Processor scheduling is a very important part of any operating system. To explain how Amoeba handles process scheduling, it is imperative to explain how Amoeba defines a process. A process is a running program, which has an address space, a set of registers and a stack. Amoeba also uses threads, which share the same address space as the main process[1]. Threads are scheduled by the kernel and are used to create a multi-programmed environment.

Amoeba's process scheduling is much like other operating systems. The kernel implements a round robin scheduling algorithm with a given time-slice which is usually 10 milliseconds. This may change if a process is blocking.[1]

Since Amoeba is a distributed operating system, processes are distributed over multiple computers. At any given time, a computer may have any number of processes running on it, with each process possibly having multiple threads. Amoeba does not, however, have threads of a common process running on multiple computers[1].

A base server takes a user's command and starts the execution process. The base server next allocates processors from the pool and begins to distribute processes to them. When the command is finished, the processors are placed back into the processor pool to be reallocated.[1]

In the case of a heavy load where no processor is idle, the base server distributes the processes based on each processor's current load. The one with the smallest load will get the next process will then be timeshared.

Thread Communication

Since processes are distributed across multiple machines, some kind of communication is needed. This is implemented using remote procedure calls, or RPCs for short. RPCs are a message-based communication scheme across a network which invokes a procedure on a remote machine

The RPC communication is actually between threads of a process, not the process itself. This can be from a user thread to a user thread, a user thread to a kernel thread, and a kernel thread to a kernel thread[1].

The RPC creation, sending, receiving and decoding are all processed by the kernel. There are library functions defined which give interfaces to the user program and allows for a seamless view of thread communication to the programmer.

Some examples of thread communication may be needed to get a good feel for how the system works. A kernel-to-kernel thread may consist of a message like "execute program x with these parameters on your processor". This is one of the main conduits for which the distribution of processes is handled. Another may be a user-to-user message like "give me return data from what you just processed". This could happen from a user program like a compiler, in which compiled data must be passed from one thread to another (e.g. parser and assembler information).

File Management

The basic underlying concept in Amoeba is the object. Fundamentally abstract data types, objects are managed by server processes. Each object is named and protected by capabilities. A capability is a 128-bit permissions field which defines what the object is "capable" of. The format of one is shown in Figure 2.

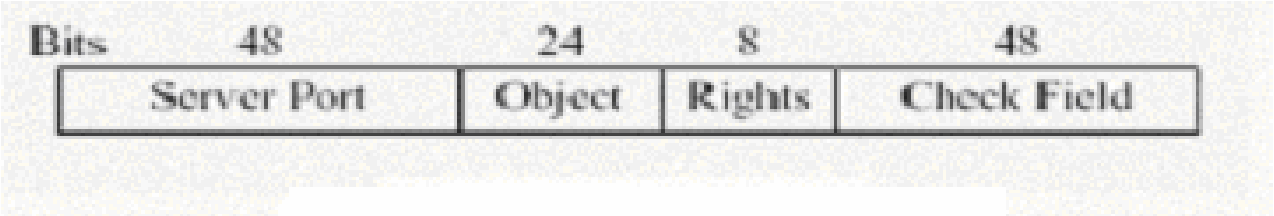


Figure 2: A capability [4]

The *Server Port* identifies the machine on which the server of the object resides. The *Object* field identifies which specific object needed. This field is identical to the i-node number in UNIX. The *Rights* field specifies the operations that the holder of the object can perform. And the *Check Field* serves as a validating the capability; essentially protecting it from being forged by user processes which manipulate the capability directly. [4]

The file structure in Amoeba is quite different than normal operating system file systems. A diagram of the Amoeba file structure is pictured in Figure 3.

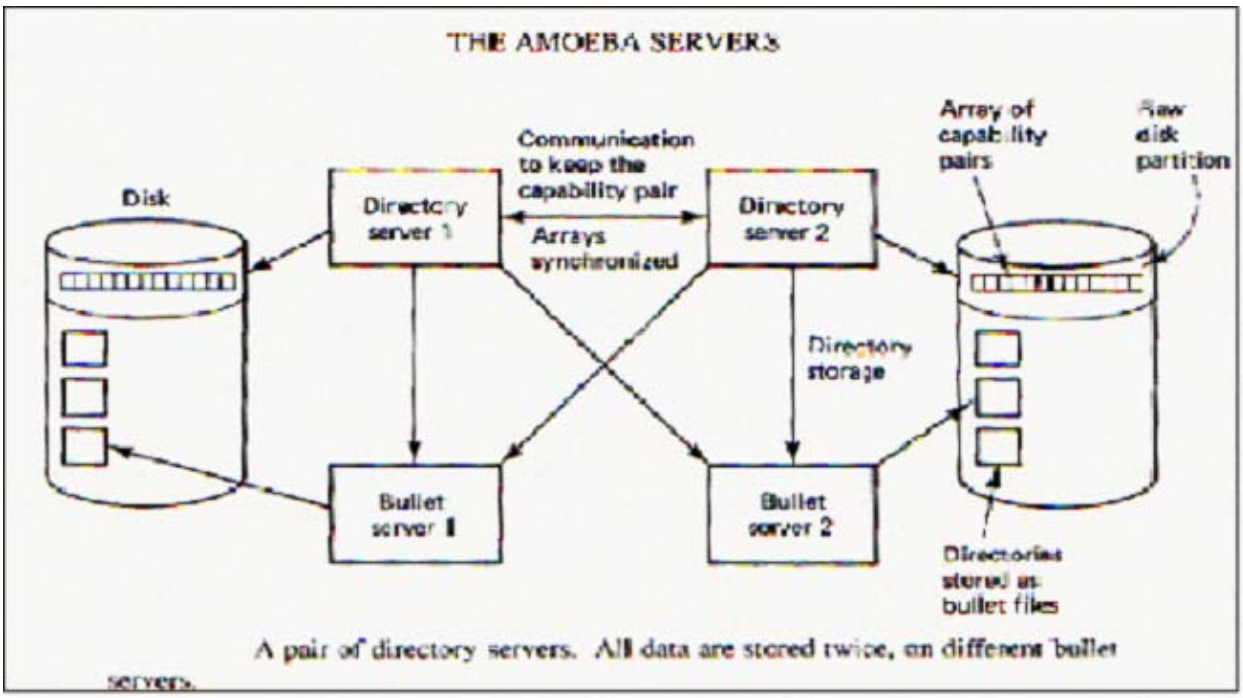


Figure 3: File Structure [4]

The file server in Amoeba is known as the Bullet Server, called so for being very fast. It has a very large buffer cache and stores files contiguously, in core and on disk to give very high performance. Files in the Bullet store are unique because they are immutable, meaning you can not change them once they are created. This only allows for basically two operations, read and create. Bullet files are also created atomically so they can be created in cache but they do not officially exist until they are written to disk, or committed [4]. Since bullet files are immutable, the size is known at creation time, hence the contiguous file storage. This way, files can be read into memory in a single disk operation. Bullet stores all the entries in a table at the beginning of the disk, similar to the UNIX i-node table, which holds the disk address, size, and a random number for capability checking [4] (See Figure 4 for visual).

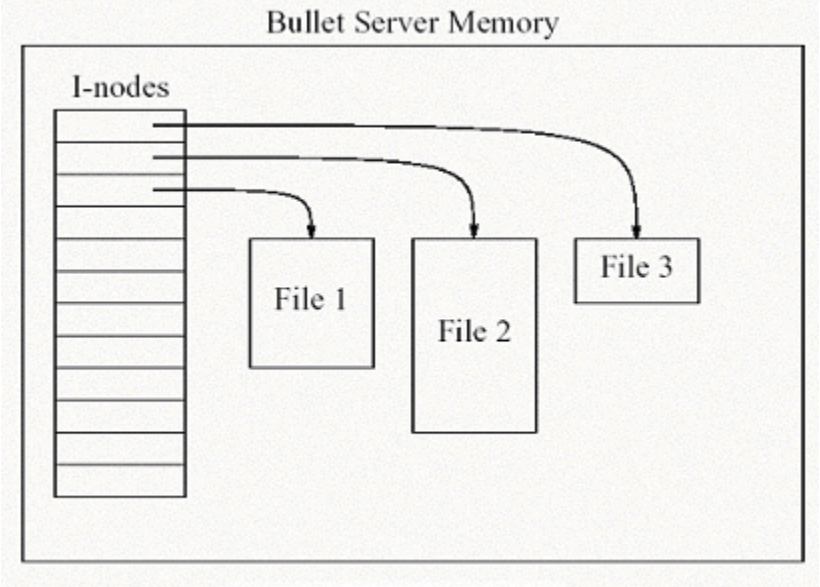


Figure 4: The Bullet Server [4]

Files in Amoeba, however, are not named by the file server. The general naming service is implemented by the directory server which employs an arbitrary directed graph of directories [4]. The directory server maps the ASCII names to ports with a port for each directory leading to the directory graph. The directory server supports several commands including: lookup, readall, makedir, recover, remove… Unlike the file server, directories are not immutable. A directory object consists of an ASCII string and a capability set. The entry can either have a single capability or multiple capabilities, which allows the entry to map onto a set of replicated files.

Name	Cap set	*Owner*	*Group*	*Other*
README	file 1	111	100	100
profile	directory	111	110	100
x.c	file 2	110	100	000

Figure 5: Directory Structure [3]

The directory graph has no concept of an absolute root since there is no tree structure, thereby allowing each user to have his own '/' directory [4]. An added degree of fault tolerance is added to the directory server by having a duplicate of itself. Each copy has its own set of data, and they communicate with each other to keep the data consistent. This allows for very high reliability because if one server goes down, this is detected by the other one, and it carries on functioning. When the failed server comes back onboard, it is brought up to date with all the changes by the surviving one and continue cooperation. This also allows for easy upgrading, seeing as how you can perform upgrades to one server without halting execution of the other one. Then when the first one is restarted with the new version, the second one can undergo the upgrades.

I/O Scheduling

Input and output (memory, disk) is handled by the microkernel threads. To read a block from the disk, a process does remote procedure calls to a disk I/O thread in the kernel. The calling process does not know that the server is actually a kernel thread because user thread interfaces are identical to kernel thread interfaces. [6] As far as file I/O goes, having multiple threads within a process accentuates the distributed and parallel and computing model very well. For example, when a file server receives a request, one of the threads will accept it and if that thread then blocks for some reason, the other threads can continue. However, with all this independent control, Amoeba does require the use of semaphores to provide inter-thread synchronization, via a common block cache.

Group communication is handled by a reliable facility which guarantees that all receivers get group messages in the same exact order. [6] An interesting fact about network I/O in Amoeba is that it supports nothing but the Ethernet network medium. It also uses terminals the same way UNIX works, however, when a user logs in, he does not log into the machine, but to the system as a whole.

Memory Management

Amoeba's memory model, which is handled by the microkernel, is very small and efficient. A process's address space contains segments mapped onto user-specified virtual addresses. These segments include, but not limited to, a text/code segment, a data segment, and a stack segment for main/thread process. [6] When a process is executing, all of its segments are in memory. Amoeba does not utilize virtual memory or swapping so it can only run programs that do not exceed physical memory. This does however keeps the management scheme simple and provides high performance. [6]

Reasons to Use Amoeba

With the knowledge that distributed operating systems are not widely used in the computing world, a listing of the reasons to use Amoeba seemed to be appropriate. Most of the more obvious reasons to use an Amoeba system have been touched on previously in this paper. The need to control several machines from one user process can be accomplished by a distributed operating system, such as Amoeba. Another situation that could efficiently utilize a distributed system is any program that commonly requires large computations. A more practical reason to use the Amoeba operating system is to save money, both by utilizing old hardware and by lessening the need to

purchase expensive servers. An additional reason to use Amoeba is that the source code for the operating system is available for free, so you can make your own version of Amoeba to suit your needs.

Reasons Not to Use Amoeba

As was mentioned, Amoeba is not a widely used operating system. There are distinct reasons for this. Other clustered operating systems that operate in a similar fashion to Amoeba are more readily available and easier to use. Why would a company take a chance on a poorly documented, poorly maintained operating system like Amoeba, when they could use a much more dependable system maintained and supported by a well known software company. Another reason to avoid Amoeba is that there is no warranty. This is due to the fact that the source code is available for each user to edit. These reasons, along with others, make Amoeba a risky business venture.

Heath's Personal Reaction

I have very mixed feelings about Amoeba. I have a few reasons why I like it, but I have many more as to why I do not. I like Amoeba because of how difficult it is to understand. The system uses techniques not found in most other OSs. The RPCs, the way it handles processes and threads and how it can take a multitude of slow machines and create one large computing center is amazing.

Mostly why I don't like Amoeba is because of the lack of documentation and support. When looking for documents, there was really only one repository which contained redundant information. Also, the documents didn't really disclose all information like I had hoped. They left me still asking questions about how the system worked. This was extremely frustrating.

Amoeba has been tested on networks with what seems today to be archaic hardware. I have been unable to find any information about an amoeba system running on anything higher than a 16MHZ RISC processor, or higher than a 486 x86. This leads me to believe that Amoeba is not suited to today's processing requirements.

Matt's Personal Reaction

I think that the concept of a distributed operating system is extremely interesting. I actually would like to study this idea further. As far as Amoeba itself, I do not think that I would trust it enough to actually use it, considering its lack of documentation, testing and support. I am really interested in seeing an Amoeba system in operation, but due to their rarity I probably will not get that opportunity.

Tim's Personal Reaction

Hating to be unoriginal, I have to agree with Matt and Heath. I am finding it very difficult to enjoy learning Amoeba due to its lack of documentation. I have, however, always found distributed and parallel operating systems intriguing and wish that Amoeba were more up to date with today's processing needs. It just might be interesting enough to mess around with when I have the time. Unfortunately, with it being a research design, its capabilities are limited and out of date.

Conclusion

Overall, the Amoeba distributed operating system is an interesting and unique idea. It provides system administrators the power to distribute work over a variety of machines without inconveniencing the users. Despite this and other useful features, we predict that Amoeba will not ever become a widely used operating system. However,

the knowledge gained by the development of and research about Amoeba has helped in the design of other distributed and clustered operating systems.

Works Cited

#1. Tanenbaum, Andrew S. "The Amoeba Reference Manual, Programming Guide." Vrije Universiteit, Amsterdam, 1996.

#2. Tanenbaum, Andrew S. "The Amoeba Reference Manual, System Administration Guide." Vrije Universiteit, Amsterdam, 1996.

#3. Tanenbaum, Andrew S. "The Amoeba Reference Manual, User Guide." Vrije Universiteit, Amsterdam, 1996.

#4. Tanenbaum, Andrew S., M. Frans Kaashoek, Robbert van Renesse and Henri E. Bal. "The Amoeba Distributed Operating System – A Status Report." Vrije Universiteit, Amsterdam.

#5. Tanenbaum, Andrew S., Robbert van Renesse, Hans van Staveren and Gregory J. Sharp. "Experiences with the Amoeba Distributed Operating System." Vrije Universiteit, Amerstdam.

#6. Tanenbaum, Andrew S. and Gregory J. Sharp. "The Amoeba Distributed Operating System." Vrije Universiteit, Amsterdam.