A Distributed Micro–Kernel for Communications Messengers

Christian F. Tschudin, Giovanna Di Marzo, Murhimanya Muhugusa and Jürgen Harms
University of Geneva, e-mail: <tschudin@cui.unige.ch>

June 15, 1995

Abstract

Mobile software agents require a distributed execution environment in which they can command the use of many resources like memory, CPU time and bandwidth. While current research seems to concentrate on roaming agents at the application level and tries to define suitable high–level environments, we think that such an execution environment must be rooted at a very low–level. We argue that these environments have very in common with distributed micro–kernels, and that therefore one should position them even below current operating systems. In this paper we report on the implementation of such a micro–kernel built for communications messengers (messengers are autonomous threads of control that can travel through the network: they are the low–level elements from which higher–level agents can be built). We present our viewpoint and describe the current state and some implementation considerations of our Messenger Operating System prototype MOS.

Keywords: communications messengers, operating systems, mobile software agents, distributed computing.

1 Introduction

At the same time as the client/server programming model becomes more and more accepted in the industrial domain, a new metaphor is lively discussed in research on computer science: mobile, roaming or itinerant software agents promise elegant and efficient ways for implementing a new generation of distributed applications. The possibility to send code (i.e., an agent) to a remote side that will execute it, could be understood as just a smart way of doing remote procedure calls: instead of doing several queries across the network, this set of queries is packed into an agent, send once, executed remotely and the results are retrieved later or brought back by the returning agent. Reduced latency and less network traffic are just two advantages of such a solution.

However, one realizes that agents have a far bigger potential than just a RPC replacement. New types of applications become imaginable e.g., intelligent forms that collect information among a set of persons and forward themselves to the next destination, or electronic markets where agents can be sent to watch stock market prices and, if they drop below a certain level, become active buyers. The intriguing aspect of agents is that the application is not built into the system but can be defined at run time by the end user launching the agents.

Agents require en execution support i.e., hosts that are ready to receive and execute agent code. Several proposals for such environment are currently discussed which focus on the capabilities of agents and the instruction set as well as the environment required for their programming. But not many ideas have been proposed how the computing resources should be managed in such a volatile environment. Agents with bugs can stray forever in a agent network if there are not rigorous damping mechanisms.

Building such a network and managing its resources resembles in many ways the problems of setting up distributed operating systems. In this paper we propose to look more closely at the lower layers of current distributed application architectures and to imagine a network of platforms for very simple agents on top of which applications, but also distributed operating systems, are implemented. Our goal is to deal with resource allocation at the lowest possible level instead of having to implement it at operating system, network and again at application level.

In section 2 we introduce *communication messengers* as very simple and prototypic agents and discuss the concept of messenger–based operating systems. In the same section we try to isolate some basic assumptions of the messenger–oriented view on distributed applications and look at related work. Section 3 reports on our prototype messenger operating system and some of the resource management techniques we rely on. A brief summary closes this article.

2 Operating Systems for Roaming Software Agents

When designing an execution environment for mobile software agents one should stress one criteria: In order to host a maximum of application types built with agents, the environment should be generic and minimal. In a first place we present messengers as a guarantor for genericity regarding protocols and communication services. It turns out that messengers are at the same time prototypical mobile agents and therefore represent the ideal level for positioning resource management problems.

2.1 Communications Messengers

Messengers were introduced in the domain of computer communications [9] as a generic replacement of protocol–specific messages also called protocol data units (PDU). Hosts exchange instructions packets (communications messengers) instead of PDUs that carry with them the full or parts of the protocol's logic. The instruction set in place is such that messengers can command the submission of other messengers, locally as well as remotely, and that only an unreliable datagram service is needed between hosts. Higher–level protocols like reliable transfer, routing etc. can be implemented solely with messengers, thus, messengers form a protocol–unspecific support for computer communications in general.

At a conceptual level, messengers are autonomous flows of control that can travel across the network. Their functioning is based on the presence of execution platforms that receive messengers and execute them unconditionally. Principally, such a platform must provide concurrent *threads* of execution, *shared memory* and some means for thread *synchronization* as well as the *channels* through which messenger threads can send new messengers to neighboring hosts (figure 1). Note that two messenger threads can interact only when they agree in a rencontre and that messenger threads are not addressable (and can therefore not be stopped or killed by other messengers).

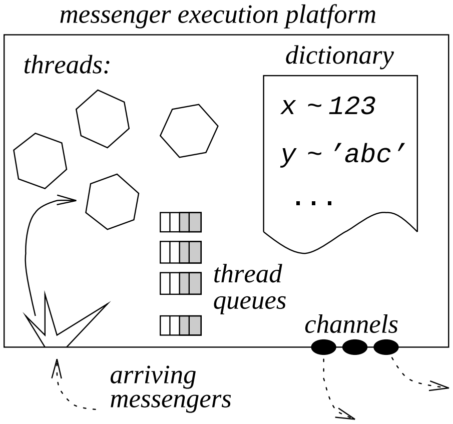

Figure 1: A schematic view of a messenger execution platform.

2.2 Agents

When compared with "agents" (no consensus exists on a definition of this term), messengers are rather low–level. Agents require more than just a network of bare execution environments. Usually, agents know a lot about the environment in which they execute e.g., in the sense that they depend on specific configurations of data in the shared memory area. Especially when object–oriented programming is applied to agents one must install class hierarchies in all execution platforms. Moreover should agents be capable of complex (object–) interactions between them while messengers are rather "autistic" and cannot be addressed directly.

However, messengers can be used to implement (carry) the behavior of agents. The simple concepts of the messenger execution platform presented above are sufficient to realize different styles of agent interactions, thus to implement a high–level agent by one or more low–level messenger. In [7] we proposed an architecture where messengers form a kind of "control plane" that is responsible for all aspects of mobility and resource consumption and a native code plane where the remaining agent's functionality is implemented (figure 2). Because co–resident messengers can enter into contact by shared memory, it is even possible to support direct method invocations for agent–level communications ("meeting"). These techniques are discussed in section 3.2 after having presented the principle of messenger–based operating systems.

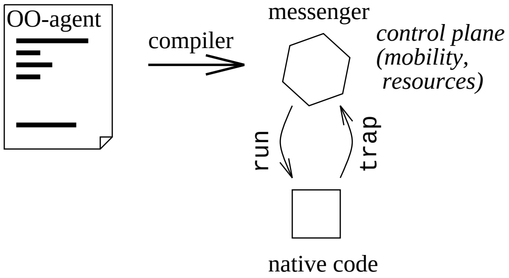

Figure 2: A (messenger) control plane for distributed applications.

2.3 Messenger–based operating systems

Current micro–kernels like MACH or CHORUS confine a minimum of OS functionality in the kernel and let standard OS functionality like file system, process management etc. be handled by independent processes outside the kernel. These micro–kernels also offer additional services like location transparency for interprocess communication (IPC) and new services like group communications. This allows processes to send messages without having to care about the physical location of the destination process(es) and forms a basis for building distributed operating systems.

We think that current micro–kernels are not minimal enough. The fact that a micro–kernel offers cross–node services (like transparent IPC) means that it has an internal kernel–to–kernel protocol for providing this functionality that must be respected by all nodes. Changing or replacing this protocol usually means that the software has to be changed on all nodes. This is especially true when some limitations are hardwired into the protocol (CHORUS had to change its capability format because the range of 10 bits reserved for addressing the destination node turned out to be too small – now 14 bits are built into the protocol, limiting CHORUS' configurability to systems with 8096 nodes). The goal of messenger–based operating systems was to remove from the micro–kernel the protocol–dependency and to exploit the messenger's property to be a generic protocol execution support [8]: At the end we would have a *protocol–free* micro–kernel.

Hence, what is needed is a messenger platform (which is devised to be protocol–free) that can serve as a micro–kernel. The ensemble of interconnected messenger platforms constitute a (potentially world–wide) distributed micro–kernel. It *is* a distributed kernel because although there are no real cross–node services offered by such a micro–kernel we have *the* "unreliable remote execution" service that is mediated by messengers. In consequence, every other distributed service like transparent message exchange must be built on top of this single and generic interaction method. In which (technical) way the messenger platform model has to be extended in order to become a micro–kernel for other operating systems is discussed in section 3. Beforehand we stress some other points of view in which we depart from common believes found not only in the domain of micro–kernels but also in distributed object systems and roaming agent execution environments.

Against transparency: The absence of cross–node services is not the consequence of our micro–kernel design but its starting point and working hypothesis. Thus, we think that transparency, although very convenient, is an insufficient base for distributed programming (see also [11]). A distributed application *must* be able to "sense" its physical distribution if it wants to handle typical problems (node crashes, link failures etc) of distribution itself. Moreover is the roaming software agents development a trend that goes in the quite opposite direction of (location) transparency. Our conclusion is therefore: provide local services only.

Build security *with* messengers: One of the most prominent cross–node services (that event roaming agent environments provide) is security. The goal is to tag agents (messengers) with security–relevant information in order to validate their origin and to deduce their access rights. Agents that do not pass this validation are simply rejected by the platforms in the hope that the validated ones will not break the platform's functioning or obtain access to sensitive data. In fact, this represents a very special "filter" for agents which requires an important certification infrastructure in parallel to the agent network. Our believe is that one should not be afraid of the virus and worm like appearance of messengers but that one should explore means to let messengers handle security issues themselves in the same way as unsecure message exchange is used to implement security protocols. First steps in this direction are presented farther below.

2.4 Related work

The research work we reference here is either in the domain of mobile application–level agents or low–level distributed micro–kernels. These two domains are not so disparate as they seem. We are convinced that the application–layer work will more and more concentrate on problems that are since long known to the low–level operating system issues (resource management, deadlocks etc).

TeleScript, Java and Python: These languages are a new generation of "virtual machine interpreters" similar to the old UCSD Pascal P-code. The definition of the TELESCRIPT language is, probably for commercial reasons, still not made public. As far as was legible through overview articles [13, 2], it seems to be close to FORTH and also includes a higher–level language that can be compiled into low–level TELESCRIPT. Although HotJAVA was announced as an environment for downloading interactive code into a WWW browser [6], it is clear that the JAVA language (which offers concurrent threads) is a very good candidate for a more symmetric environment where code (i.e., agents) also travels in the client to network direction. Like TELESCRIPT it comes with a code authentication mechanism.

`Tcl`: The Tool Command Language `Tcl` [3] is often cited as an attractive candidate for mobile agents. In fact, its ability to execute `Tcl` script remotely seems sufficient for agents, but the execution environment has severe drawbacks. On one side it is not rich enough (concurrent execution of scripts, synchronization etc), on the other side the environment is too powerful (uncontrolled direct access to the underlying operating system) and currently provides (except for CPU time restrictions) no resource control mechanisms. There are proposals for `Tcl` extensions and restrictions, and future will tell if `Tcl` can be retrofitted for a satisfying agent environment.

Micro–Kernels: Micro–kernels have a long tradition in operating systems research and there is a consensus that they are a key technology for distributed operating systems. Vicarious for classical references to MACH, CHORUS or AMOEBA etc., we point to a recent paper [1] that also has a more complete reference list. An interesting point of this paper is that their $V++$ micro–kernel fits into less than 150 kbytes and that this quantity of binary code can easily be placed in a PROM. We expect that our micro–kernel, including the messenger language interpreter, will fit into a similarly small area. Thus, computers can be shipped directly with a micro–kernel built into (instead of the BIOS of PCs as we have it today). Genericity is important in this case. Once such a machine with a messenger micro–kernel is connected to the network, it immediately becomes available for roaming messengers and extends the "substrate" in which messengers can live. From the domain of operating systems we also mention Shoch's early experiments with distributed computing based on worm programs [4].

3 A Messenger Micro–Kernel

In this section we report on our implementation of an operating system for messengers, the messenger operating system MOS. First we describe the language that underlies our messengers, show how this interpreted language makes native code execution available and discuss our approach of managing resource allocation. Finally we describe how this execution environment is "populated" with (or "booted" by) messengers.

3.1 The messenger language `MØ`

The `MØ` (M-zero) language was designed and implemented as an experimental language for messengers in 1994 [10]. It's structure resembles very much POSTSCRIPT, from which it inherits its stack orientation and the dictionary concept for resolving references by name, but without any graphics related operators. Most standard operators have self-delimited one–letter keywords which results in a very compact coding (e.g., the complete alternating–bit–protocol (ABP) messenger has less than 75 bytes). Two details of the language shall be presented at this place: the new data type `key`, and the implementation of the shared memory space of an `MØ` execution environment.

3.1.1 Shared memory area and the key data type

In MØ, shared memory is the only way of exchanging data between messenger threads. It is represented in form of dictionaries: all messengers have access to them and can define there arbitrary data pairs (the first element of this pair will usually be an identifier). Messengers can furthermore lookup already defined entries and can also remove them. Currently there are two such global dictionaries defined: globaldict and servicedict. The difference lies in the ability to be browsed: while it is possible for a messenger to loop over all entries defined in servicedict, this is not possible in globaldict. This restriction forces messengers to know the exact identifier that was used to deposit some data in globaldict.

This alone cannot guarantee that nobody but the holder of the exact identifier can remove an entry, but it already provides considerable privacy when used in conjunction with randomly chosen identifiers (see below). If an entry should be readable but at the same time protected from being removed, there is a special “secret define” operator: instead of using the plain identifier for the data pair, the MØ platform computes a new identifier by applying a one–way hash function. The resulting value will become the visible first part of the data pair. Entries defined this way can only be removed by giving the original (secret) identifier.

“Keys” are a new data type introduced in MØ for various usages. Key values are arrays of 64 bits which – in most cases – are randomly chosen. Their main usage is for generating “unique” identifiers. For instance: during the processing of the ABP protocol, a flag has to be left at the remote side telling if the next messenger is in sequence or just a duplicate. In order to avoid name clashes due to several concurrent applications all using the same ABP messenger code, each application generates a random key and uses it remotely as an identifier for the flag. Keys have to be used as identifiers for the above–mentioned “secret-define” operator. But keys can also be used as true “secret keys” for the DES routines (data encryption standard) which are also built into the MØ language. Finally we mention the currency defined in MØwhere “cheques” are also referenced by keys: internal tables are used to prevent that a cheque can be cashed more than once.

3.1.2 Secure publishing of services

Given the public and unprotected nature of the shared dictionaries, how can service procedures by “published” without having antagonistic messengers wipe out all traces of them?

Based on the servicedict and the key data type, messengers can advertise their services at a “well–known” place and in a secure way. For this, they choose a random key k_1 and make a “secret define” in the servicedict of a character string identifying the offered service (the one–way function returns the visible key k_2, see figure 3). The service messenger then uses the same operator and key k_1 for adding in globaldict an entry with the “service access point” e.g., a read–only dictionary with execute–only procedures. Potential clients can browse servicedict and find all visible keys that match the required service. These keys are used in the unbrowsable globaldict for finding the access procedures. To sum up: The browsable servicedict allows clients to find the offerings, the use of the “secret–define” operator in globaldict prevents that the associated entries can be removed although the dictionary has read/write rights for everybody. Thus, by providing very simple primitives in MØ, messengers can protect their vital data and even make portion of their code accessible to other messengers in a controlled way.

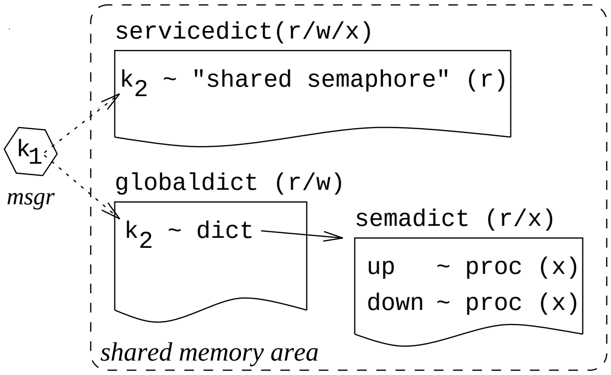

Figure 3: Secure public service definition in a messenger platform.

3.2 Controlling native code execution

Native code execution is essential for building a micro–kernel that should be a competitive alternative to run standard operating systems. The main idea of messenger–based operating systems is that native code execution shall be governed by (interpreted) messengers and that messengers intervene only when communications or resource control tasks have to be handled. The access to the hardware had to be made accessible through the MØ language.

3.2.1 The `run` instruction

Most prominent is the `run` instructions which "jumps" into native code and returns at well–defined moments (figure 2). These are: end–of–timeslice, supervisory (system) call and traps due to illegal instructions. The parameters for the `run` instruction (register values, content of address space etc) are implicitly taken from the state of the calling messenger: `set-processing-unit` selects the native instruction set that the messenger will execute (e.g., an MØ platform can offer various emulations and modes of native code execution), `set-registers` is used to initialize the selected CPU's registers and `set-page-map` declares the (possibly virtual) address space in which native code execution should take place. All these values can be retrieved and are available to the messenger in form of standard MØ data types (integers, strings etc). Note that address spaces can be shared between messenger threads, permitting thus the implementation of multi–threaded native applications.

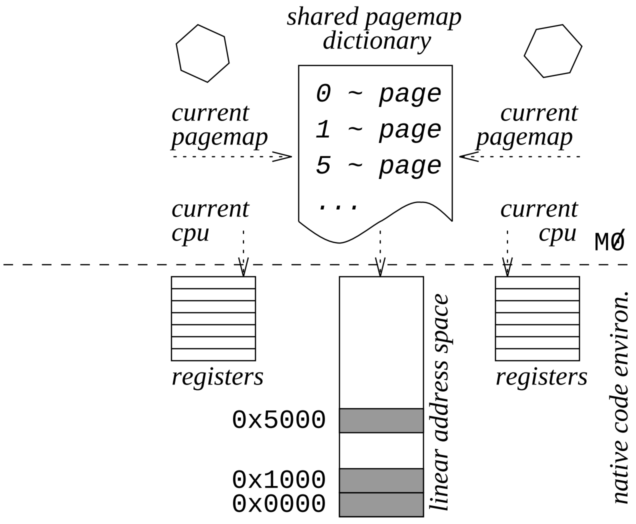

Figure 4: Sharing the virtual address space for native execution.

At the return from a `run`, the messenger must decide which action to take. In case of an end–of–timeslice it could simply do a `run` again, or in case of a "system call" try to satisfy it before jumping back to native code execution. In both cases, messengers find themselves in a position that in standard operating systems like UNIX is occupied by the kernel (which usually takes these decisions). However, with MØ it is not necessary that all these decision procedures be written in the interpreted language. More probably is that a second messenger controls the native execution of the "kernel code" of a classical operating system. Thus, all the first messenger has to do is to pass the "system call" on to the other messenger which would give it "to its native code". Messengers intercept interactions between native threads and have the power to redirect these interactions locally as well as across several MØ platforms.

3.3 Resource control

For the Messenger Operating System we followed a market model at the lowest level possible. In principle, all resources are, if available, accessible to all messengers without restrictions (there is no security system built into the MOS). However, the usage of resources is controlled by a charging mechanism. Currently we apply this model for the CPU and memory resources, bandwidth will be the next candidate to look at.

3.3.1 Charging of messengers and their revenue

Each messenger has its own account from which charges are deduced on a per–usage basis (figure 5). Messengers can withdraw money from their account in form of cheques that can be passed to other messengers of the same platform. Messengers can also share an account, enabling this way a division of labor (one messenger concentrates on the principal work to do, another messenger asserts the required funding). The prices for resource consumption varies according to the offer

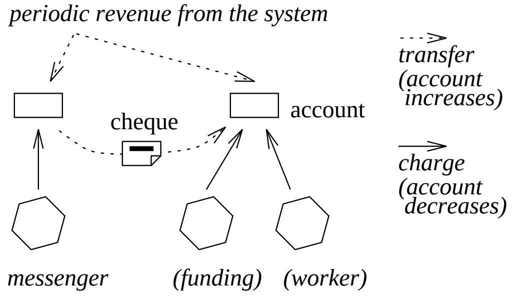
periodic revenue from the system
transfer (account increases)
charge (account decreases)
account
cheque
messenger
(funding)
(worker)

Figure 5: Charging messengers for their resource consumption.

(hardware limits) and demand (messengers).

Messengers obtain an initial amount of money when they start. Moreover, they receive at regular intervals a revenue that depends on the degree of competition among the messenger threads inside the platform. Less money will be given back to the messengers in times of heavy load than what would be necessary for the competing messengers to continue their "lifestyle". Thus, the systems gives incentives for releasing (or selling) memory, reduce the CPU share etc, but the platform does not implement or enforce a resource allocation policy for the messengers. Messengers with no money left on their account are silently removed.

3.3.2 Lottery scheduling

The CPU resource management is based on lottery scheduling [12] instead of round–robin scheduling: At the end of each timeslice a lottery is held for determining the next messenger to execute. Messengers can increase or decrease their chance to obtain the timeslice by buying or releasing "tickets". This system has the nice property that buying more tickets automatically adjusts the relative CPU shares of the other messengers (selling actual timeslots in advance is not very fair and is an invitation for speculation). Once a messenger has won the lottery, it obtains the CPU for the full timeslice and is charged according to the number of tickets it had put into the lottery. To figures are used to adjust the prices for tickets: There is the default number of tickets for each messenger and a system target load (e.g., four times the default number of tickets). If the total number of issued tickets exceeds the system target load, prices are progressively increased so that messengers with more tickets than the default number have very high charges and, at long term, will run out of money.

3.3.3 Memory sponsoring

Memory resources are also charged on a per–usage basis. This principle applies to all local variables and state information of messengers. However, another mechanism had to be put in place for the shared memory area because data items at this place belong in some sense "to everybody" (it is possible that the messengers which added a memory–intensive entry into the `globaldict` disappeared). For this we decided that global memory has to be "sponsored" (figure 6). If there is no sponsoring, a global item will either disappear (is automatically removed from `globaldict`) or, if it is still referenced, its content is set to zero. Sponsoring is implemented by letting messengers attach an account to a shared data item. In contrast to messengers which possess only one account, global data items may have several of them. In this way it is possible that new sponsors can show up in a non–exclusive way (otherwise one could "attack" shared data items by putting them on an empty account which effectively removes them).

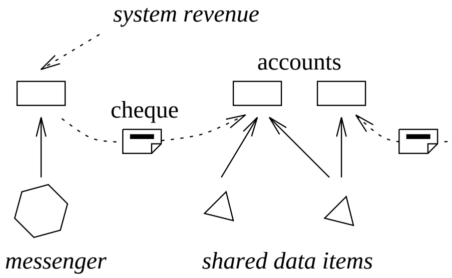

Figure 6: The "sponsoring" of shared data.

3.4 The boot messenger

Confronted with this concept of a bare messenger world, surprisingly often the question arises on where the first messenger comes from that should populate the platform(s). At long term we imagine that there could be a network of messenger platforms where it is virtually impossible to have no active messengers left (remember the pains of removing the internet worm [5]). Thus, we suppose that a freshly booted platform with network connectivity is very quickly discovered and populated by the messenger community.

Considering stand–alone platforms, we need a first messenger that, like `init` under UNIX, starts the desired activities. This first messenger will probably be read in from harddisk or any other non–volatile storage space. Currently we compile the boot messenger into the micro–kernel code. All it does at this time is to start two console messengers and connects them to the screen device: the (human) user can then interactively type in `MØ` code and explore this way the shared dictionaries or start up messengers whose code is defined by the interpreter's startup code. More complex interactions with the platform involving many commands are currently done via the network from a UNIX workstation using messengers to download the necessary code and to start it remotely.

3.5 Current state and future work

At the time of writing we have a first version of our messenger micro–kernel running. We used the public MØ interpreter that is multi-threading safe (debugging was done under a standard UNIX system) and has tight memory usage control for MØ data values (no memory leaks). The micro–kernel runs now on a i386-PC and has ethernet connectivity, it is developed under the LINUX operating system. We already mentioned the lottery scheduling (tickets) in place as well as the possibility to run native code (the i386 is configured for a linear address space with one identical code and data segment). Accounting is now done for CPU time and local memory, the sponsoring concept for shared data items is under implementation.

Only few experiments were conducted in order to calibrate the charging and money redistribution mechanism. Thus, beside trying to run a UNIX shell binary under messenger control and to go towards emulation of a standard operating system, we focus on operational questions of the resource market model and hope to find simple local pricing rules such that when several platforms are put together we can observe smooth global resource allocation.

4 Summary

Messengers were introduced in this paper for two reasons: first they are very simple mobile software agents, and second they guarantee the maximal genericity for communication services. We presented the conceptual elements needed in execution platforms and showed how these platforms can be extended in order to become micro–kernels on top of which ordinary operating systems can be run. The central point is that messengers become responsible for native code execution. We also reported on our prototype implementation of such a micro–kernel and some of the algorithms used for setting up a resource management system that relies solely on rules that are local to a single platform. More practical experiences are needed to verify whether all desired services and properties (including security and execution guarantees) can be realized with this simple messenger platform structure.

References

[1] David K. Cheriton and Kenneth J. Duda. A caching model of operating system kernel functionality. In *First Symposium on Operating System Design and Implementation (*OSDI*),* USENIX *Association*, pages 179–193, 1994.

[2] Tom R. Halfhill and Andy Reinhardt. Just like magic? BYTE, pages 22–23, February 1994.

[3] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[4] John F. Shoch and Jon A. Hupp. The “worm” programs – early experience with a distributed computation. *Communications of the ACM*, 25(3):172–180, March 1982.

[5] Eugene H. Spafford. The internet worm program: An analysis. *ACM SIGCOMM Computer Communication Review*, 19(1):17–59, January 1989.

[6] Sun Microsystems. The HOTJAVA browser technology demonstration. URL `http://java.sun.com/`, March 1995.

[7] Chr. F. Tschudin. OO-agents and messengers. Position paper for the ECOOP95 workshop W10 on Objects and Agents, August 1995.

[8] Chr. F. Tschudin, G. Di Marzo, Muhugusa Murhimanya, and Jürgen Harms. *Messenger–Based Operating Systems*, July 1994. Technical report 90 (Cahier du CUI).

[9] Christian F. Tschudin. *On the Structuring of Computer Communications*. PhD thesis, Université de Genève, 1993. Thèse No 2632.

[10] Christian F. Tschudin. *M0 - a messenger execution environment*. Usenet newsgroup `comp.sources.unix`, Vol 28, Issue 51–62, June 1994.

[11] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. Technical report, Sun Microsystems Laboratories Inc., November 1994.

[12] Carl. A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional–share resource management. In *First Symposium on Operating System Design and Implementation (*OSDI*),* USENIX *Association*, pages 1–11, 1994.

[13] Peter Wayner. Agents away. BYTE, pages 133–118, May 1994.