Cahier du Centre Universitaire d'Informatique No 90

Messenger-Based Operating Systems

Christian F. Tschudin <tschudin@cui.unige.ch>
Giovanna Di Marzo, Muhugusa Murhimanya, Jürgen Harms
University of Geneva, Switzerland

July 1994, revised September 14, 1994

Abstract

This report proposes to employ messengers, initially developed in the context of communication protocols, as a fundamental component of the architecture of distributed operating systems. Current microkernels offer non-local services which require the collaboration of neighboring machines or special servers and are implemented using special protocols. These protocols are hard-wired into the microkernel and can not easily be changed, which is problematic for interworking and scaling. We propose a modified software architecture for operating systems where all non-local services are implemented outside the basic software coat that hides the hardware. The key element of such an architecture are messengers i.e., worm-like programs exchanged between neighboring machines: they represent an intermediate layer between the basic computing platform and the operating system to support. Operating systems are still run in native mode but under the control of messengers. An important element of this architecture is a currency mechanism used to control the resource allocation of messengers and the implemented operating systems.

Keywords: Distributed operating systems, communication messengers, operating system structure, communication paradigms, microkernel, worm programs.

1 Introduction

The primary goal of our research[1] is to explore the possibility to found operating systems on the *messenger paradigm*. The assumption is that communication messengers provide a generic technique and a unifying framework not only for computer communications but also for the implementation of distributed operating systems. In this report we sketch a software architecture based on messengers that extends current microkernel approaches. The specification and validation aspect of messengers, also part of the research project, are not discussed here.

1.1 From Operating Systems Research to Computer Communications and Back

Initially, operating systems were introduced as a *management* and *abstraction* layer between the computing hardware and user processes. The abstraction provided uniform access to the available resources, while the management aspect finally lead to multi-programming. In a first phase, this layer became bigger and bigger due to new and more complex services (e.g., resource management). The advent of data networks also falls into this phase (new services had to be added), but it triggered the reverse trend of breaking up the monolithic operating systems, leading to the current microkernel approach. Only few functions are now collected

[1]Research project funded by the *Swiss National Science Foundation* [1].

in the kernel. The others, mainly *management and policy* aspects, are shifted to external servers. Microkernels can be seen as a kind of “glue” necessary to tie together the servers – they provide a common substrate on top of which server and user processes execute. The question is whether this hardware abstraction level is minimal, or if it is possible to remove additional management aspects from microkernels and make the coat around the hardware even thinner.

To provide a response to this question one must examine the set of high-level services offered by a microkernel, especially the services which span several CPUs and provide “location transparency” (server addresses and capabilities) or which depend on specific protocols and server processes (memory mapper). These underlying concepts differ from microkernel to microkernel. Moreover are similar services implemented quite differently. Interworking under such circumstances is, without gatewaying, virtually impossible. But even within the same kernel architecture is scaling or merging to a global degree cumbersome (e.g., address spaces implicitly coded in fixed-field capabilities). In order to remove these arbitrary limitations, one must make the hardware abstraction layer truly generic with regard to network topologies and protocols used.

There is an analogue argument in the domain of computer networks, where different protocol suites have hard-wired the piling of protocols to use. The resulting static protocol stacks are currently being made flexible by introducing special negotiation protocols that permit to assemble at run-time the wanted protocol stack (e.g., [2]) or by offering huge protocol graphs [3]. However, the problem remains that the communication architecture has at least one built-in negotiation protocol (with implicit addressing and routing assumptions) such that this specific protocol is difficult to replace.

As an answer to these concerns we propose communication messengers [4], that is, small worm-like programs written in a specific messenger language and exchanged between hosts. Received messengers are automatically turned into independent processes and unconditionally executed. They provide a complete replacement of message-based communication, do not require preinstalled protocol stacks and represent an alternate implementation technique for protocols in general.

Applying this observation from computer communications to operating systems means that one should avoid schemes where specific protocols are mandatory for the correct functioning of the supporting base system i.e., the abstraction layer. The goal is to set up a common but protocol-*unspecific* execution environment: messengers are the key element to achieve this.

For an discussion of the messenger paradigm we refer to [4] where messengers are introduced from a communications point of view. A first implementation of a messenger environment, called `MØ`, is freely available and was distributed in the `comp.sources.unix` newsgroup [5]. It is also described in [6].

In this report we link messengers to operating system concepts and look at the actual questions to be examined during our research project. The report ends with an overview of related work and a concluding section.

2 Towards Messenger-Based Operating Systems

Our longterm goal is the creation of a fully generic “substrate” consisting of interconnected bare messenger platforms. In a first place this substrate is empty and waits to be populated by messengers which mediate between the user and the network of platforms. Messengers are nearly unrestricted in what they can do with the found computing resources. This section looks at how one can operate operating systems *under the control* of messengers.

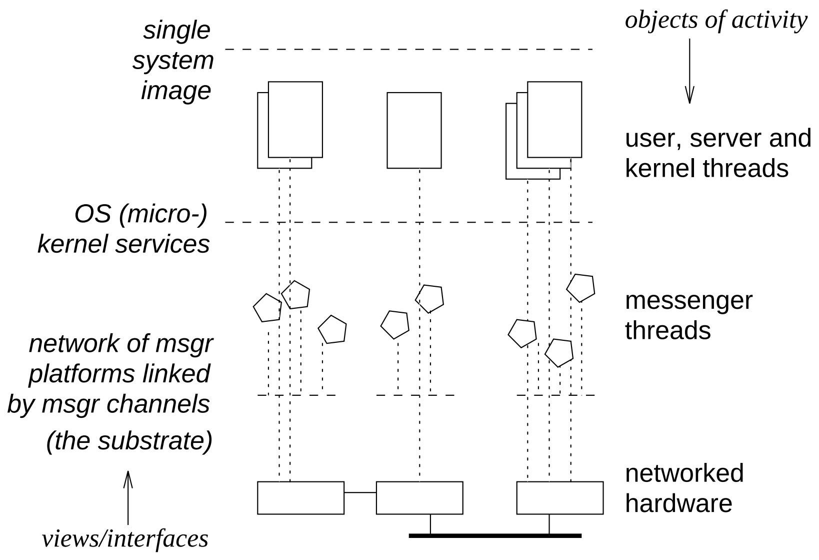
single system image
objects of activity
user, server and kernel threads
OS (micro-) kernel services
messenger threads
network of msgr platforms linked by msgr channels
(the substrate)
networked hardware
views/interfaces

Figure 1: The relations between of objects of activity and the service interfaces.

2.1 Controlling the execution of native CPU code through messengers

In terms of the classical distinction between user and kernel mode, messengers occupy the role of an intermediate agent. They ‘sit’ at the transition point between these two modes. It is the messengers which must ask the platform to execute a given sequence of native CPU instructions. Upon termination, the messenger resumes its execution. Messengers are, from the viewpoint of a user process, invisible: what for the user thread is a trap, segment violation or an ordinary termination, becomes for the messenger the end of the execution request. It depends on the controlling messenger what comes next. It is possible to implement operating system functionality in the (interpreted) messenger language. However, it is also possible that the messenger lets execute another piece of native CPU code which provides the necessary OS functionality: in this case the implemented OS is even not aware that messengers were used to control its execution.

From the point of view of the messenger there is no principal difference between code belonging to the user mode and code executed for the kernel (server) of the operating system to operate. The corresponding execution mode, which we call native CPU mode, corresponds to the top layer of figure 1. There is a separate messenger interpretation mode that supervises the native mode (middle layer). And finally there is a third mode always present in a messenger environment: it corresponds to the internal operation of the messenger platform (the “coat”).

2.2 Messenger Strategies for Building Operating Systems

The most radical utilization of the network of messenger platforms would be a single huge messenger that contains the complete logic of an operating system (the “father and mother of all tapeworms”, cited according to [7]). By injecting this “germ” into the substrate it unfolds and distributes to a complete distributed environment. Messengers are used in this context as a configuration and exploration tool: because they are at the same time a communication tool, they can be sent out to discover the network topology, measure transmission delays and the computing power, the available resources of each platform etc. Based on these information (or other statically fixed data), the set of machines to incorporate into the system is determined and tied together by installing a common address space, routing procedures, possibly resident server messenger processes etc.

More realistic is a scenario where the initial messenger does not contain the complete operating system, but knows where the necessary resources can be found. Such resources may be the native code of some OS functionality which is stored under some well-known key in a platform. The role of the messenger thus is to instantiate the required processes. This scenario is, of course, less general because it relies on preinstalled resources, but it comes close to the current model of OS kernels which come with a whole bunch of high-level (non-local) and ready-to-use services.

Injecting two initial OS messengers into a set of platforms may lead to many, perhaps different instances of operating systems. To which degree should the platforms intervene in the allocation of memory and CPU resources? It is clear that it is insufficient to manage such resources with a local platform view only (fairness among different concurrent messenger-based distributed operating systems). However, starting to include in the platforms a global allocation policy is in conflict with our basic assumption that there should be no non-local services implemented at this level. In order to overcome this difficulty, we propose to introduce a *currency mechanism*.

2.2.1 A currency mechanism as a unified tool for controlling resources

Introducing a currency means that all platform services have a price. Each platform has its own local currency, and prices may vary with time. There is only a local allocation policy that modifies the prices according to the available resources (offers) and demands (allocation requests).

The big advantage of this approach is that there is a single unified instrument able to deal with memory, CPU time and transmission requests. The essential thing, of course, is that the money becomes detachable from the platform, thus can be carried away. The platform implements a local pricing policy, but it leaves completely unspecified the way *how* the money is distributed and used. Messengers can, for instance, pool their money, or accumulate it, or obtain resources when prices are low (low demand) and sell it when demand is high. Such a currency mechanism does of course not solve the (global) allocation problem, but it provides the instrument to implement a solution with messengers.

Messenger threads also become “older” by charging them on a time base for allocated resources. If their credit is consumed, they are silently removed.

2.2.2 Direction of future research

Native code control and memory map interface: Based on the existing MØ-platform, new operators will be added to permit the (messenger) controlled execution of native CPU code as well as the access to the CPU’s memory paging functionality. It will be demonstrated how parts of an existing operating system (currently we are looking at the freely available LINUX operating system) can be run under the control of messengers.

Currency mechanism: In order to supply each platform with its own local currency, we need to define a “money format” that makes forgery difficult (at least detectable). As the “prices” for platform services are variable (depending on demand and available local resources), we must define an algorithm for computing them. The questions to examine are:

- Because the currency is local, the platform must offer an exchange service: define an algorithm of computing money exchange rates.
- Show that these two local algorithms are apt for achieving global stability (no raging inflation because a platform protects itself by rising prices . . .).

Process (task) abstraction: An interesting point to examine is the question whether the simple messenger process abstraction is sufficient for representing the heavy-weight processes of ordinary operating systems. It may be necessary to introduce "virtual platforms" which group resources of several messenger processes in order to unleashing all these resources at once. Debugging questions also may influence this decision.

Thread scheduling: Is the currency mechanism powerful enough for implementing different scheduling priorities? It would be nice to stick with a simple (preemptive) round-robin messenger scheduling and to control the scheduling of native code execution by the amount of money spent by the messengers.

At the current stage of this research project we are mainly concerned with the basic functionality of supporting true operating systems on the basis of messenger platforms. Security issues are, at this level, not in the focus of the project: this coincides with e.g., the CHORUS approach where security is dealt with at the operating system level only (but not in the microkernel). First explorations indicate that the messenger paradigm does not exclude solutions in this area and that security functionality can be handled at the level of and with messengers.

3 Related work

Surprisingly, the messenger paradigm glimmers in many domains, but the exchange of programs (messengers) has never been proposed as an universal tool for solving control *and* communication tasks.

Worms: The worm concept of Shoch and Hupp has many similarities with the approach described in this report: "worm programs" can move themselves across a network of machines, searching for idle stations where they can execute themselves.
Messengers are lower level than worms, thus could be used to implement them: worms rely on a specific machine/machine boot protocol (no protocol genericity), have a explicit goal (find an idle machine and squat it) and they consist of native machine code which prevents the introduction of a currency mechanism to limit a self-replication meltdown. For other related worm work see the bibliography in [7].

Knowbots and computational Mail: Knowbots (knowledge roboters, Cerf/Kahn [9]) can roam through data base services to collect useful (library) information. Electronic mail based on instructions can be used to implement "intelligent documents" (Borenstein [10]).

RPC **in a heterogeneous environment:** Falcone proposes a NCL (network command language [11]) as a tool for implementing remote procedure calls between heterogeneous operating systems. It is a true "networking-through-programming" approach.

Upcalls: Upcalls [12] i.e., thread-per-message processing instead of thread-per-layer software architecture, are not directly related with messengers, but there are parallels. As a software structuring technique it is well known and in use. Messengers adhere to the same software structure. See for instance the x-kernel approach [13].

4 Conclusions

This report describes the rationale behind a new research project on distributed computing: it puts particular emphasis on the aspects of distributed operating systems. The report proposes to embed current microkernels into a control plane populated with messenger threads. Messengers are programs expressed in a specific language: they can be sent to a neighboring machine where they are turned into independent processes. The messenger language is interpreted, but there are special operators to control the execution of native machine code. The aim of this concept is to obtain a service interface (hardware abstraction offered by the messenger platform) which offers purely local services. All services going beyond a single machine are not part of the "coat" covering the hardware and must be implemented either with messenger technology or by ordinary native code executed *under* the control of messengers. By charging messengers for each service, a mechanism is available for implementing global resource allocation policies *outside* the messenger platforms. This mechanism is essential in order to prevent the potential uncontrolled behavior of messengers. It may be this missing damping factor that prevented the worm concept to be used as a generic technique for distributed computing.

References

[1] Jürgen Harms and Christian F. Tschudin. Communication messengers as a basis for distributed algorithms (theory and implementaion). *Swiss National Science Foundation, project 20-40631.94*, February 1994.

[2] M. Vogt, T. Plagemann, B. Plattner, and T. Walter. A run-time environment for DA CAPO. In *Proceedings of INET'93, Internet Society*, 1993.

[3] S. W. O'Malley and L. L. Peterson. A highly layered architecture for high-speed networks. In Marjory J. Johnson, editor, *Protocols for High-Speed Networks II*, pages 141–156. Elsevier, 1991.

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

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

[6] Christian F. Tschudin. *An Introduction to the* `MØ` *Messenger Language*. Technical Report 86, Centre Universitaire d'Informatique, May 1994. Cahier du CUI.

[7] 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.

[8] M. F. Kaashoek, R. van Renesse, H. van Staveren, and A. S. Tanenbaum. FLIP: an internetwork protocol for supporting distributed systems. *ACM Transactions on Computer Systems*, February 1993.

[9] Vinton G. Cerf. Networks. *Scientific American*, 265(3):42–51, September 1991.

[10] Nathaniel S. Borenstein. Computational mail as network infrastructure for computer-supported cooperative work. In *ACM Conference on Computer-Supported Cooperative Work (CSCW'92)*, pages 67–74, Toronto, November 1992.

[11] Joseph R. Falcone. A programmable interface language for heterogeneous distributed systems. *ACM Transactions on Computer Systems*, 5:330–351, November 1987.

[12] David D. Clark. The structuring of systems using upcalls. In *Tenth ACM Symposium on Operating Systems Principles*, pages 171–180, December 1985.

[13] Norman C. Hutchinson and Larry L. Peterson. The *x*–kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.