To be published in the proceedings of ICC'95 workshop on Intelligent Computer Communication

The Messenger Paradigm and its Implications on Distributed Systems[*]

Giovanna Di Marzo, Murhimanya Muhugusa, Christian Tschudin
Jürgen Harms
Centre Universitaire d'Informatique, University of Geneva
24, rue Général Dufour, CH-1211 Genève 4
Phone: +41 22 705 76 43, Fax: +41 22 705 77 80
e-mail: `dimarzo@cui.unige.ch`

Abstract

Most distributed systems are built on top of a message exchange infrastructure. Processes coordinate their execution by exchanging messages which are interpreted according to a pre-established set of protocols. We present in this paper a novel way of communicating which does not require the preconfiguration of protocol entities. The host initiating the communication has the ability to instruct the other host "how" the data exchange has to take place. We call this "communication by messengers". The host that initiates the data exchange sends to the recipient a program called "messenger" which contains all or parts of the protocol logic. After it has been received by the recipient, the messenger is executed and can, if necessary, deliver its payload. Communication by messengers will change the way distributed systems are built. This paper discusses the impact of the messenger paradigm in various fields such as communication protocols, distributed operating systems and intelligent agents.
Keywords: communication by messengers, distributed operating systems, intelligent agents, protocol design and implementation.

1 Introduction

The classical architecture of distributed systems relies on computer communication protocols. The main components of a distributed system are protocol entities that reside in different hosts and which exchange messages following a well defined communication protocol.

This architecture has some drawbacks:

- The different parts of a distributed system (or application) must be pre-installed and well configured before the application can be run. Surely this poses a problem for software installation, testing and debugging. On one side, some distributed applications require an important deployment for testing and debugging; on the other side, users are not interested to use a software if it has not been thoroughly tested and is not largely deployed;
- Distributed operating systems use the exchange of special purpose messages for implementing non-local services. This imposes that a minimal set of protocols be explicitly wired into the kernel. These specialized protocols restrict the kernel's genericity and adaptability;
- Distributed applications usually have two parts: one that is responsible for communication management i.e, implementing a protocol for moving data between the different hosts; the other part uses the communication facilities provided by the first one in order to implement the actual distributed algorithm.
- Extending the above argument, we can argue that for the application programmer, developing a distributed application requires a different methodology from that of a centralized application. When developing a distributed application, the programmer must explicitly handle efficiently the data exchange in the system and its processing whereas for a centralized application, only data processing is considered. We can thus argue that developing distributed applications that way is error prone; this is exacerbated by the difficulty to implement "correct" computer communication protocols. The ideal situation for the programmer would be to implement distributed applications the same way he/she implements centralized applications;

We present in this paper a new paradigm for performing computer communication, which we call "communication

[*]This work is supported by Swiss National Fund for Scientific Research (FNSRS) grant `20-40631.94`

by messengers”. When two hosts communicate using this paradigm, they exchange programs called “messengers”. The messenger contains the appropriate code to instruct the recipient “what” it has to do next. A messenger contains both the data and the rules necessary to perform a given protocol.

This way of performing protocols does not require protocol entities be preconfigured in the communicating hosts. It is the source host which completely determines which protocol is to be used for the data exchange; the destination host is not required to “know” the protocol being used. When used as the basis for distributed applications, the communication by messengers paradigm brings more flexibility. It becomes possible for an application to explore the network and spread itself instead of having to rely on pre-configuration and installation of application specific software.

Another view of a messenger is that of a thread created in a remote host on behalf of the source host. From a programmer’s viewpoint, sending a messenger to a remote host is similar to the creation of a local thread. This offers a uniform way for programming both centralized and distributed applications. Moreover, the communication by messengers paradigm can be used as a simple way to extend the remote procedure call paradigm.

Section 2 presents in more details the messenger paradigm, the following section 3 discusses its impact on the implementation of computer protocols. Section 4 shows how messengers can be used as the basis for distributed operating systems and section 5 discusses the impact of this paradigm on intelligent agents. Finally, we conclude this paper in section 6.

2 The Messenger Paradigm

The messenger paradigm is born from a new way of thinking computer communications. Instead of the classical sender/receiver model based on protocol entities that exchange and *interpret* protocol specific messages, Tschudin proposed in [13] a communication model based on the exchange of protocol unspecific programs, called the *messengers*. Hosts receiving messengers will then *execute* the messengers code instead of interpreting them as messages.

2.1 The Messenger Platform

All hosts involved in a messenger based communication share a common representation of the messengers and provide a local execution environment, *the messenger platform*. A messenger is executed sequentially, but several messengers may execute in parallel inside the platform. Platforms are connected through unreliable channels through which messengers are sent as simple messages or data packets.

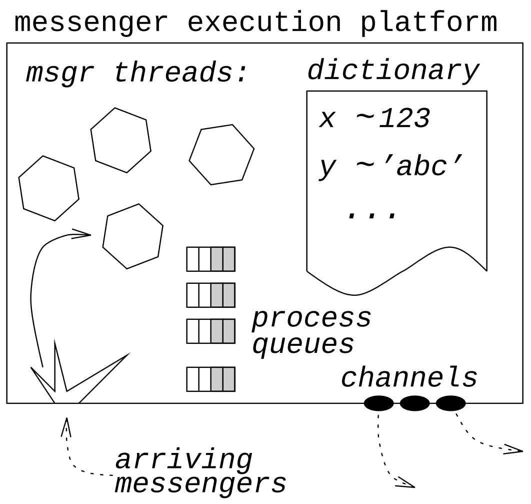

Figure 1: Messenger platform

Figure 1 shows the basic elements of a messenger platform:

- A set of *channels* enables a messenger platform to exchange messengers with other platforms. The channels provide an unreliable datagram service which enables messengers either to reach entirely and correctly the neighboring platform or to be discarded or lost;

- Integrally arriving messengers are turned into *independent messenger processes* or *threads of execution*, also called *messengers* for simplicity. The platform does not interfere with their execution except for programming errors or unavailable resources. Also do other messengers have no means to stop or kill another messenger process without its consent;

- The executing threads are able to share common data inside a given platform by the means of a *dictionary* which contains pairs of keys and data values;

- *Process queues* are the low-level mechanism offered by a platform in order to allow messengers to implement basic concurrency control functionality such as synchronization of threads, mutual exclusion, etc. Process queues are FIFO queues of messenger processes. All threads of the queue are blocked except the thread at the head of the queue.

Platforms share (1) a common messenger programming language, used to express messenger behavior; and (2) a common external representation of the messenger, used for the physical exchange messengers.

2.2 Primitives of a Messenger Programming Language

Messengers are exchanged between two platforms as simple messages, using a common external representation. Arriving messengers are turned into threads of execution by the corresponding platforms. A messenger is then a sequence of instructions expressed in a *messenger programming language* common to all platforms. Beside general purpose instructions (arithmetic operations, logical operations, etc), a messenger programming language must offer a set of specialized primitives. Here we present an example of such a set of primitives:

- Process queues, channels as well as global variables in the dictionary of a given platform are accessed by a key. Keys are either generated by the platform (e.g., name of a channel) or can be constructed/chosen by the messenger processes using the `key()` primitive.
- Global variables in the dictionary are accessed through their key using the `get(k:key)` and `set(k:key, v:value)` primitives;
- A messenger is sent through a channel to another platform by the `submit(k:key, m:msgr_descr)` primitive. It is also possible to create a new local messenger process by submitting a messenger to a special "loop-back" channel – the new process is completely independent of its launching process;
- A messenger is able to replace its behavior by another behavior with the `chain(m:msgr_descr)` primitive;
- Process queues play a central role in the synchronization of messenger processes. Primitives for handling process queues are: `enter(q:key)` which enables a messenger process to enter a queue, such a process will then be blocked until it reaches the head of the queue; `leave` which enables the messenger process at the head of the queue to remove itself from the queue; `stop(q:key)` which stops the queue so that when the messenger process at the head of the queue leaves the queue, the next messenger process coming at the head of the queue will remain blocked; and `start(q:key)` which resumes a queue previously stopped by the `stop` primitive. A referenced queue that does not yet exist is automatically created by the platform.

By the use of the `submit` and `chain` primitives, messengers are able to act as mobile entities that can propel themselves across the network to search for a given information or to perform a given task in a remote place. Alternatively, a messenger can continue its task in the local platform while the messengers it sent out will execute remotely before returning with the result of some sub-task.

The possibility for a messenger process to replace its behavior with the `chain` primitive can be compared with the change of behavior of actors in the actor model of Agha [1]. The difference resides in the fact that the behavior change is a fundamental part of the actor model: actors are seen as abstract machines dedicated to process one incoming communication. Actors always change explicitly or not their behavior before processing the next communication. On the other side, messengers are mobile entities that perform given tasks, which are not necessarily reactions to incoming communications.

2.3 Two Messenger Platform Implementations

We built two prototype implementations of messenger platforms that allowed us to make first experiences with tangible messengers and to study in more details the basic elements of such platforms. Three elements have to be distinguished:

Messenger data format: In order to exchange messengers, two platforms must agree about the allowed "channels" as well as the data format used to express the messenger. This channel must offer a datagram service, thus relies on a datagram delimitation mechanism. For practical reasons there will be more information inside a messenger than just the instruction sequence: version number, possibly a checksum and, for convenience, an arbitrary data field etc.

Messenger language and execution model: After having agreed upon the placement of the instruction sequence(s) inside a messenger datagram, two platforms also must "speak the same language". The messenger languages will probably be interpreted ones, but messengers may also contain or consist of native CPU code (machine language code).

Conventions about local data resources: Finally, we stress the importance of *conventions* required across all messenger platforms understanding a given messenger language: because each platform will offer varying services (number and type of channels, file system and other data bases, code libraries etc.), it is important to have conventions about the naming of such resources as well as the mechanism used to discover and access them. Such an infrastructure resembles very much *type systems* e.g., the one found in the distributed object execution environment CORBA [2].

In the following, we briefly describe our two messenger platforms.

2.3.1 The MØ messenger Platform

The MØ (M-Zero) project started in december 1993; a first version of the MØ platform that runs on various UNIX systems and that contains a full interpreter of the MØ language was distributed in June 1994 [12] and can be fetched by FTP at `cui.unige.ch` under `pub/m0`. The MØ language is ASCII-based and closely resembles POSTSCRIPT although it does not contain any graphics related operators. Operators usually have single–character names, thus making messenger programs very compact. The platform's common variable space is represented as a dictionary, enabling messenger processes to persistently deposit and fetch data values. The implemented messenger channels are based on raw ETHERNET and UDP.

Different major types of protocols have been implemented in the MØ environment: network exploration using ETHERNET broadcasts ("wave protocol"), alternating bit and Stenning's sliding window protocols, self–fragmentation and defragmentation as well as a "remote console". Some of these protocols were implemented in variants e.g., using code downloading techniques.

Although useful for exploring messenger programming styles and techniques, the MØ platform is currently incomplete insofar as it has very limited links to the world of applications and other operating system services. Thus, more work on the third element, the "conventions", is needed, as well as the integration of standard services (windowing environment, file system) into MØ. Still under study are strategies for managing inside MØ the various resources offered by the platform (CPU time, memory, transmission bandwidth etc.)

2.3.2 A SCHEME-based Messenger Platform

A case study was made on a *functional* messenger language, namely one that would be based on SCHEME. It resulted in a SCHEME interpreter named MSGR-S [15] that supports parallel interpreter processes. The new functions `setglob` and `getglob` enable the access to the common variable space, the function `exist?` can be used to check if a name already is in use. Most notable is the way new messengers are generated and submitted. While in MØ the programmer can refer to a special `convert-to-messenger` operator in order to build a messenger datagram from an arbitrary (code) string, this step is hidden by the MSGR-S interpreter: only a *closure* can be submitted i.e., a lambda expression that has no parameters. Hence, the `submit` operator takes such a closure, (re-)constructs an "equivalent" ASCII string and sends it inside a datagram through the requested channel. This string is simply a SCHEME lambda expression that, after reception at the destination platform, is evaluated in order to obtain the function to start.

We think that basing messengers on a LISP-like language is for some programmers an essential asset, especially when messengers are used to implement intelligent agents in the context of AI. Although elegant, this *implicit* back-transformation of an internal closure to an equivalent ASCII-string has its price: on one side in terms of computing time and the problem of limited datagram length, on the other side in terms of expressiveness, possibly excluding the implementation of some classes of protocols within this environment.

Both environments, MØ and MSGR-S, have shown to us the importance and necessity of a new programming style. The first time we started to code messengers we found it difficult to "overcome" the thinking in terms of static execution flows. Later on, however, once one is more familiar with the (possibly unreliably) spreading execution flows of messengers, it is fun to express communication tasks with messengers. We already see specialized "interaction techniques" poping up which will form the basic elements of a new communications programming style.

3 Protocol Implementation with Messengers

The most fascinating property of the messenger paradigm is the ability to "run" protocols without requiring the remote communication partner to know it. The sender has, within the limits of the receiver's resources, the complete freedom in choosing (a) the protocol and (b) the way it is implemented. In this section we discuss the relation between messengers, classical PDU-exchange based protocols and the corresponding implementations.

3.1 Messengers for PDU-exchange Based Protocols

The central part of a protocol is the definition of the interaction sequences that are necessary for realizing the wanted communication service. These interactions i.e., the exchange of protocol data units, have been carefully analyzed in order to verify that the sender and receiver side will not enter a deadlock or can not make any progress. All this work should be preserved and one would like to know if and how known protocols, although expressed in terms of PDU exchange, can be realized with messengers.

The first time you try to setup messengers that adhere to a given protocol, you realize that the usual way of expressing the protocol's logic e.g., a procedural description of the protocol entities or their finite state machine equivalents, is not very useful because you can not assume the presence of such entities. More appropriate is a change of viewpoint: instead of looking at the protocol entities, we focus on the PDUs. More precisely, one has to imagine the "journey" a

PDU makes and one has to attach to it the actions that are triggered by its reception.

As an example take the alternating bit protocol and consider the journey of a user data unit presented to the ABP service access point: first, the current flag value of the sender has to be fetched, a PDU is assembled which then is sent out. On reception we have to check the receiver's flag value and possibly have to return an acknowledge. The acknowledge PDU just unblocks the sender thread that waited for a timeout or an ack message. This description is the starting point of the ABP implementation with messengers: one messenger is used to forward the payload, another messenger must acknowledge the arrival of the former one.

The general philosophy for implementing a protocol expressed in terms of PDU exchanges between static protocol entities is therefore to define for each PDU type a corresponding messenger type. Based on this structure one has to find all data and control paths through the protocol entities and assign the related instructions to the messenger types. This decomposition of the protocol's logic is related to a software structuring technique called "upcalls": this technique is not new, but it is not always intuitive to produce [3]. Thus, at long term it would be interesting to have automated tools that take a PDU-based protocol as input and produce a set of "equivalent" messengers.

To close this section on protocol implementation we emphasize that one can speak about a "protocol being performed" although there are no protocol entities installed. Nevertheless *do* protocol entities exist, when messengers were constructed along the mentioned recipe: their functionality is provided by transient messengers processes and not by the usual statically configured execution flows. In other words: only *abstract* protocol entities are present and the messengers become a "strange" way of coding of the PDUs.

3.2 Messenger Modes of Operation

It can be a waste of bandwidth to send with each messenger the complete logic necessary to treat its payload. This *full messenger mode* is nevertheless indicated in the case we choose a very specialized way of performing a protocol, but usually, more efficient methods exist.

A first improvement consists in sending the code once and to install it on the remote system. The platforms global variable space can be used for this, allowing subsequent messengers to carry only the lookup instructions necessary to find and execute the *downloaded* instructions.

Another mode of performing protocols with messengers is to *emulate* the classical PDU model. A first messenger is sent to the remote platform and "installs" itself i.e., waits for follow-up messengers. These follow-up messengers mainly carry the classical PDU as defined by the protocol to implement. Once arrived, these messengers deliver their payload to the first messenger process that will treat the received PDU like an ordinary protocol entity.

The previous model can also be called PDU *tunneling*, because messenger are used to create a PDU channel across two messenger platforms. This model does not require other messengers than the PDU carriers. In fact, one can imagine that two standard protocol stacks are linked into a messenger platform such that sending a PDU at the stack bottom results in the creation of a local messenger. It's sole task is to forward the PDU to the remote platform, where it is delivered to the remote stack's reception point. The stacks will not be aware of the tunneling of their PDU. For the messengers, it looks like linking two services that are local to the involved platforms.

What has been shown in this section is that messengers can be used in many different ways, even in ways that are not inspired by the "communication by instruction" paradigm. In fact, we see messengers as one pole of a *spectrum*, the other being the exchange of predefined message types (PDUs) and in between many variants or hybrid forms. Putting all communications functions on instructions is probably not economic - using only predefined messages is too rigid: which technique is more suitable depends on the problem setting.

4 Messengers and Distributed Operating Systems

An operating system is a piece of software which adds an "abstraction" and a "management" layer on top of a bare machine. Current operating systems offer to the user, among others, the abstractions of process and virtual address space. The management aspect has lead to the concept of multiprogramming and recently to that of distributed systems which can be seen as an extension of multiprogramming. With the advent of distributed systems were born the distributed operating systems whose management task spans multiple machines connected by a network.

Distributed operating systems try to create the impression of a virtual machine (a single system image) out from a set of hosts interconnected by a network. They rely on an efficient communication mechanism between the different hosts to achieve their goal. Communication by messengers being a new way of performing computer communications, it is natural to see whether it can be used as a basis for distributed operating systems and what can be its implications on such systems. That is what is discussed in this section.

4.1 Modern Operating Systems

Most operating systems [11] are designed as a monolithic piece of software structured in modules or layers, each module being responsible for a well defined service. Examples for the services offered are process management and memory management. With the advent of computer networks, more functionality was to be added in operating systems which grew bigger and became more difficult to debug and maintain. Moreover, management decisions and policies were directly "hard wired" in the heart of the system making it difficult to evolve.

To overcome the drawbacks of this monolithic architecture, modern operating systems e.g., Amoeba [10] and Chorus [9] use a micro-kernel architecture. Only a minimal set of services is confined in the kernel, other services are provided by special processes external to the kernel called servers. With this approach, management decisions and policies are shifted from the kernel to the servers. Adding functionality to the system is done by adding server processes without recompiling the whole system. The most important impact of this approach is perhaps the possibility of coexistence of multiple servers for the same kind of resources, each server implementing its own policy for managing its resources (e.g., two independent file store servers).

Distributed operating systems must provide local but also remote services. Some of these non-local services are still implemented in the micro-kernel. The different hosts of the system cooperate to offer non-local services. This is done by letting them exchange information using a message passing mechanism. This means that the different hosts must agree on a protocol or a set of protocols to interpret the different messages. These different protocols are still "hard-wired" in the kernel. As a consequence of this, changing a protocol (for example the message format) requires recompiling all the system. Moreover, each operating system uses its own set of protocols for data exchange between different hosts, thus interoperability between different operating systems is difficult to achieve.

Let us consider a process A running in host X that wishes to send a message to process B running in the remote host Y. Because each exchanged message must be handled by some entity, (in this case process B), the sending process must "identify" in an non-ambiguous way the destination process. This implies that there must exist between all the hosts a uniform way of identifying individual processes. Clearly, this is an example of a non-local service that must be offered by the micro-kernel to achieve interprocess communication.

Non-local services implemented in the micro-kernel considerably restrict the micro-kernel genericity. The key solution to truly generic micro-kernel is to remove "hard-wired" protocols from the micro-kernel. The messenger paradigm allows computer communications to be realized without preconfiguration of any kind of entity i.e., without any form of agreement on a protocol to be used by the communicating parties except the messenger platform. Using messengers it becomes possible to build micro-kernels for distributed operating systems without protocols wired in, and eventually to have truly generic micro-kernels.

4.2 Messenger Based Distributed Operating Systems

A messenger based distributed operating system should use the micro-kernel approach. The messenger paradigm allows to remove from the kernel all non-local services. The architecture of such an operating system consists of three layers. At the first layer resides the messenger platform together with the hardware: they offer to the concurrent messenger processes a virtual machine which understand the messenger language. All the hosts must implement a messenger platform. The second layer is populated with messengers which control the use of the platform's resources. The third layer, finally, corresponds to the classic process abstraction: at this place we will find the server and user processes. Figure 2 shows the architecture of a messenger based distributed operating system.

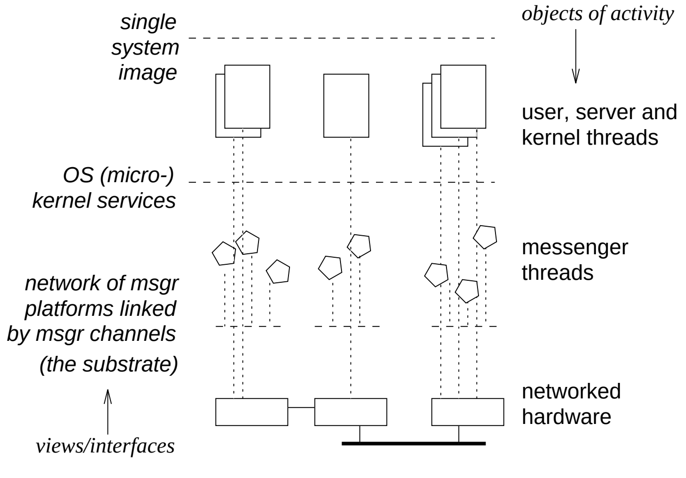
single system image
objects of activity
user, server and kernel threads
OS (micro-) kernel services
messenger threads
network of msgr platforms linked by msgr channels (the substrate)
networked hardware
views/interfaces

Figure 2: The relations between objects of activity and the service interfaces

At bootstrap time, a first messenger is injected in the network of platforms. It will create other messengers to populate the network and will collaborate to realize the operating system. One can even imagine an extreme case where different messengers are injected in the system at start-up. Each messenger will gain control of some hosts; this will partition the system in groups of hosts using different operating systems. However, as all these operating systems are messenger based, they will be able to interact. More precisely this means that a group of hosts will implement "its own" memory management, "its own" process management which can be completely different from what is used by other groups of hosts.

The controlling messenger will now ask the platform to execute some piece of native code belonging to a third-level (user) process. Thus, the execution of each (user) process is controlled by a messenger which puts itself into "native mode", a mode where it executes native CPU-code instead of messenger instructions. Whenever the process being controlled issues a system call or needs to interact for some reason with the system, the controlling messenger is brought back to the normal mode. The interpreted code of the messenger will regain control and can decide what to do. All interactions of the process with the system and all unusual conditions such as interrupts and exceptions which occur while the process is being "executed" will thus be "trapped" by the messenger. The messenger will eventually ask its platform to execute another piece of native CPU-code to handle the process' service request or the unusual condition that arose. The key idea here is that most of the time the messengers will be executing their process in "native mode".

Controlling the execution of a process by different messengers can lead to different results. Indeed, each messenger will decide according to its own code what to do with the process being controlled. We can view the messengers as forming a powerful, flexible and programmable interface between processes and the system. We need such an interface because systems become more and more complex and must evolve to adapt themselves to new conditions. Our messenger interface provides the technical base for truly configurable operating systems. We can make here a parallel with the area of communication protocol architecture. One topic of interest in this area is the configuration of flexible protocol stacks which also requires an interpreted language and some kind of "platform" (see e.g., COMSCRIPT [8].

As messengers can move freely from platform to platform without the need of protocols between hosts, a micro-kernel for a distributed operating system can be *protocol-free*. This is possible because there is no longer the abstraction of "data exchange" in a messenger based operating system. In fact, the "data exchange" abstraction has been shifted from the lowest level of the system to the messenger level. The key advantage of basing micro-kernel on the messenger principle is the possibility to build "protocol-free" micro-kernels which enable the implementation of more flexible and more robust distributed operating systems.

We are currently implementing a first prototype of a messenger based operating system where a currency mechanism is investigated to be used as a unified approach for controlling all system resources [14].

5 Messengers for Intelligent Agents

Intelligent Agents, being at the heart of Artificial Intelligence (AI), are now widely used in several others domains like human computer interactions, robotics, knowledge representation, etc. Agents are autonomous mobile entities able to interact with other agents and to react to their environment. Intelligent agents are then agents whose behaviors can be compared with those of humans [17]. The specific criteria applied above to agents can also be applied to messengers. However, messengers are not necessarily agents, but they may be used to implement them. Potentially, the messenger paradigm offers alternatives and new opportunities in the way of thinking high-level distributed applications using intelligent agents. This section aims to discuss different models of society of agents from a messenger point of view.

5.1 Actor Systems

One of the earliest models for distributed systems is the Actor model, where an actor is a computational agent which has a mail address and a behavior. Actors communicate by message-passing, also called communication or task, and carry out their actions concurrently. The behavior of an actor consists in processing a communication, which can result in the creation of new actors, new tasks and a replacement behavior enabling the current actor to process the next communication [1].

An actor, in a given configuration, can work with external actors, but an actor always waits for an incoming communication and interprets this message according to its current behavior. Actually, messengers are very close to actors, messengers are able to reproduce the behavior of actors, since they are able to create new messengers (`submit`), they can communicate through data values in the dictionary in order to mimic message-passing and they can change their own behavior (`chain`).

On the other hand, in the *universe of actors*, messages are themselves actors, hence it is possible to perform an exchange of programs between two actors.

The fundamental difference between actors and messengers seems to lie in the mailbox concept and the fact that actors are specially dedicated to process one incoming communication without control and knowledge over the location of the mailbox. Once the communication has been processed, the current actor vanishes and is replaced by another actor (by the replacement behavior). On the contrary, messengers are not waiting for communications to handle, they are simple mobile threads of execution, whose first scope is to actuate a code execution locally or remotely. Moreover, messengers are entities able to have a larger scope: messengers can have an eye not only in a restricted area as a platform but also to extend their influence to a whole net of platforms.

5.2 Intelligent Agents

Agent-based software engineering makes it easier to create interoperable software. Following this approach, the software components, which are called *software agents*, communicate by the means of an *agent communication language*. There are basically two approaches for agent communication languages: (1) the *procedural* approach where the communication between software agents is modeled as the exchange of entire programs that are directly executable; (2) the *declarative* approach where the communication is modeled as the exchange of data together with the commands useful for interpreting the data [6].

In our opinion, the messenger paradigm sits at a lower level with respect to these two approaches. Indeed, messengers clearly are able to exchange programs: a messenger `a` can deposit a communication messenger (the code of a messenger) `c` in the dictionary, messenger `b` is then able to retrieve the code `c` from the dictionary and to execute it. Messengers are also able to exchange data and their interpreting command. Two messengers not belonging to the same platform communicate in the same manner, with the difference that a messenger has been `submit`ted across the network.

The messenger paradigm can also be seen with a completely inversed view. In the two mentioned procedural and declarative approaches, software agents seem to wait for incoming communications. If a software agent needs some information to perform a given task, it has to send a request to another agent and wait for the response. In contrast to this "ask and wait" function performed by the software agents, a messenger doesn't wait for anything, it is able to search for the available information, either itself or by the means of other messengers. Indeed, a messenger can move from one platform to another, can search for the information, then take it and finally come back to its original platform. With the messenger paradigm there is no more travel of information from one location to another; instead, there is a travel of code pursuing and retrieving the information where it resides.

5.3 Multi-agent Architectures

Different architectures realize software interoperation: (1) a direct communication between software agents, where agents handle themselves their own collaboration; (2) an indirect communication where agents communicate with intermediary and local system programs, the facilitators, which in turn communicate with one another [6].

The messenger paradigm provides a homogeneous framework for realizing the direct as well as the indirect communication. As we said before, messengers are completely responsible for their coordination inside a platform. This is also true for a set of messengers executing in different platforms. However, messengers can be dedicated to the coordination of the work and the interoperation of the other messengers. These coordination messengers can have a local effect i.e., they are responsible for the interoperation of messengers in a given platform, or an across-the-network scope by realizing coordination and communication of messengers lying in different platforms. Although the indirect communication using facilitators can be exactly reproduced in a messenger-based architecture, the messenger paradigm offers a new way of thinking interoperation: a coordination messenger is not necessarily local to a platform, as it is the case for the facilitators. Instead, a coordination messenger can extend its ramifications spider-like and thus extend its control over a whole set of platforms.

5.4 Applications and Related Work

Intelligent agents are used to build, among others, intelligent interfaces for email, news filtering or meeting scheduling; software robots interacting with an external (software) environment; or electronic marketplace applications relying on mobile agents. This section presents approaches realizing the above mentioned applications based on intelligent agents and which are very close to the messenger paradigm.

Autonomous Agents for Intelligent Interfaces

Autonomous agents are used as *personal assistants* used to *collaborate with the user* in order to ease the interaction between a given environment and the user. Maes proposes an alternative approach to the *knowledge-based* approach where the agents are provided with a background knowledge concerning the user and the application [7]. Her approach relies on machine-learning techniques and consists in giving the interface agents the minimum knowledge: the agents will then learn their appropriate behavior from other agents and from the user.

Softbots

Softbots, or *software robots*, are robots acting in the world of software or virtual world (by contrast to robots acting in the real world). Robots (software or not) take actions in response to incoming communications. Softbots are agents interacting with a software environment by issuing commands and interpreting the environment's feedback. They interact with the environment either by the means of *effectors* i.e., commands used to change the external environment's state and *sensors* or commands providing the softbot with information about the environment [5], [4].

Telescript

In the TELESCRIPT technology, agents occupy virtual locations, called places. They are mobile processes able to communicate (exchange messages) with one another and to move from one place to another. The moving of agents is realized by sending their program to another place where it is executed. In electronic marketplace applications, agents are providers and consumers of goods [16], [17].

6 Conclusion

The messenger paradigm is a communication model relying on instruction-passing instead of message-passing. Messengers are mobile threads of execution that can travel across a network. The basic requirements needed to achieve the messenger paradigm are: (1) all hosts involved in the communication must be provided with the messenger platform, (2) all platforms are able to interpret the same messenger language; and (3) unreliable messenger "channels" link the different platforms.

Distributed systems are built on top of a communication mechanism. The proposed communication by messenger paradigm is a low-level communication mechanism, which can be used at different levels of abstraction. We have shown that all kind of protocols can be implemented using this new paradigm: functionality of classical protocols, based on entities exchanging PDUs, can be realized without entities and without PDU exchange, using messengers. Distributed operating systems, which depend on communication protocols, can then be implemented using messengers. We have proposed a three layer architecture populated at the second layer with messengers; this layer constitutes a control interface between processes sitting at the third layer and the messenger platform. This architecture allows the building of more generic micro-kernels, by removing from them any hard-wired protocol. Finally, at the highest level, the messenger paradigm seems to be suited for the implementation of multi-agent applications.

Going further, the classical view of data traveling the network to be exchanged is no longer available according to the messenger paradigm where it is the code which travels the network to search for the data. Following these ideas, the well-known models of server/client and sender/receiver are no longer available, since a messenger based client will ask no request to a server, but goes itself to take the information and a messenger based server will not know which clients are taking informations. Implementing distributed applications with messengers implies a new way of thinking: distributed applications will no more be seen as composed of static entities exchanging data, but as a set of mobile threads which can propel themselves across the network to look for resources.

References

[1] G. Agha. *Actors: A model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] M. Betz. Omg's corba. *Dr. Dobb's Special Report*, (225):8–12, Winter 1994/1995.

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

[4] O. Etzioni, N. Lesh, and R. Segal. Building sofbots for unix. Technical report, University of Washington, 1993.

[5] O. Etzioni and D. Weld. A softbot-based interface to the internet. *CACM*, 37(7):72–76, July 1994.

[6] M. R. Genesereth and S. P. Ketchpel. Software agents. *CACM*, 37(7):48–53, July 1994.

[7] P. Maes. Agents that reduce work and information overload. *CACM*, 37(7):30–40, July 1994.

[8] M. Muhugusa, G. Di Marzo, C. Tschudin, E. Solana, and J. Harms. Comscript: An environment for the implementation of protocol stacks and their dynamic reconfiguration. In *International Symposium on Applied Corporate Computing ISACC9́4*. ITESM Monterrey and Texas A&M University, 1994.

[9] M. Rozier, V Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the chorus distributed operating systems. Technical Report CS/TR-90-25, Chorus Systèmes, 1990.

[10] A. S. Tanenbaum, , M. F. Kaashoek, R. van Renesse, and H. Bal. The amoeba distributed operating system-a status report. *Computer Communications*, 14:324–335, July/August 1991.

[11] A. S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 1992.

[12] C. Tschudin. An introduction to the m0 messenger language. Technical Report No 86 (Cahier du CUI), University of Geneva, 1994.

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

[14] C. F. Tschudin, G. Di Marzo, M. Muhugusa, and J. Harms. Messenger-based operating systems. Technical Report No 90 (Cahier du CUI), University of Geneva, 1994.

[15] R. Lino Valverde. Msgr-s: Un environnement d'exécution de messagers basé sur un interpréteur scheme parallèle. Diploma thesis, University of Geneva, 1994.

[16] J. E. White. Telescript technology: The foundation for the electronic marketplace. White paper, General Magic, Inc., 2465 Latham Stree, Mountain View, CA 94040, 1994.

[17] M. Wooldridge and N. R. Jennings. Agent theories, architectures, and languages: A survey. In Jennings Wooldridge, editor, *Intelligent Agents, ECAI-94, workshop on Agent theories, Architectures, and Languages*, LNAI 890, pages 1–39. Springer-Verlag, August 1994.