An Introduction to Mobile Agent Programming and the Ara System

Holger Peine
Dept. of Computer Science, University of Kaiserslautern
P.O. Box 3049, D-67653 Kaiserslautern, Germany
Email: peine@informatik.uni-kl.de

ZRI-Report 1/97

An Introduction to Mobile Agent Programming and the Ara System

Holger Peine
University of Kaiserslautern, Germany.

Traditional computer communication is based on sending messages, that is, data to be processed by an active entity at the receiving side, such as a client or server process. While this model has achieved pervasive use throughout today's networked world, certain of its drawbacks have moved into the focus of interest with the massive increase in network load caused by the mass diffusion of the Internet and with the advent of wireless networks for mobile computers. Communication by data exchange requires both parties to remain on-line during the complete transaction, usually consisting of several acts of data exchange; further, often large amounts of data have to be moved across the network to the remote processing entity, only to be discarded for the largest part after processing.

Mobile agents are an approach to solve these and other problems of networked computing in the face of limited bandwidth and connectivity. Instead of repeatedly sending data to be processed by a remote processor, the processor is sent in one act to the remote data for on-site execution. Data and assorted services can be accessed locally then, and no ongoing connection is needed during the execution. Such a mobile processor, consisting of executable code and some form of execution state, has been termed a *mobile agent.*

Mobile agents have raised considerable interest as a new concept for networked computing with potentially very far-reaching implications, and numerous software platforms for various forms of mobile code have recently appeared and are still appearing [CGH95, CMR+96, GRA96, HMD+96, LAN96, LDD95, JRS95, RAS+97, SBH96], with different foci and exploring diverging solutions. Still, the basic idea is quite simple: *Give programs the ability to move.* It seems natural, therefore, to *add* mobility to the large and well-developed world of programming, rather than attempt to build a new realm of "mobile programming". Mobility should be integrated as comfortably and unintrusively as possible with existing programming concepts — algorithms, languages, programs, and operating systems. This is the basic idea of the Ara[1] system [PES97]: A platform for agents able to move freely and easily at their own choice and without interfering with their execution, utilizing various existing programming languages and even existing programs, independent of the operating systems of the participating machines. Complementing this, the system provides facilities for the specific requirements of mobile agents in real applications, security concerns being the most prominent here.

This report will present Ara and its specific approach to itinerant agents (or *mobile* agents, as they are termed in Ara), explain the concepts and features of the system and demonstrate how they are used to program mobile agents solving real-world problems. Ara is an example of middleware, situated between specific applications (e.g. an airline booking system) and the underlying operating system. It provides the system-level facilities to execute and move programs ("agents"), let them interact and access their host system, all in a portable and secure manner. The actual application ("the real work") is of no interest to the system, but

---

[1] "Agents for Remote Action".
Zoologically, an Ara (also called a macaw) is a large, multi-colored genus of parrot as pictured above, a family of birds renowned for their intelligence and longevity.

programmed in the behavior of the agents, expressed using various programming languages on top of a common run-time core. The system should be seen as a broker between visiting agents and the host system, providing the agents with access to local services and the host with control over the agents. And like any good broker, its policy is not to interfere and let both parties do their business as they see fit, yet oversee that they play to the rules of honest business.

A remark is in order here concerning the maturity and completeness of the Ara system at the time of this writing. The system is in active development, meaning that while the basic concepts are resolved and the system is sufficiently worked out for useful applications (see section 2, "Mobile Agent Applications with Ara"), many of the more advanced features are not yet implemented. This will be indicated in the presentation where appropriate.

The rest of this report is structured as follows: The initial section introduces the basic concepts of Ara, such as languages, agents, mobility and the like. The second section then demonstrates how common problems of networked computing can be solved using those concepts. This is followed by a section explaining the individual features and facilities of Ara as they are presented to the programmer. These are put to use in a section working out a complete programming example for searching the World Wide Web (WWW). A subsequent section discussing selected aspects of the Ara system architecture deepens the understanding of the system's capabilities and also shows how to extend it with further programming languages. The report concludes with a discussion of Ara regarding different approaches to mobile agent systems and future developments. Three annexes provide a glossary of Ara terms, a glimpse of evolving features, and an overview of the Ara software distribution.

1. An Overview of Ara

From a very high-level point of view, Ara consists of agents moving between or staying at places, where they use certain services to do their job. The fundamental concepts of agents, motion, places, and service access in Ara will be explained in this section. As agents are expressed in some programming language, the role of such languages within the Ara system will also be of interest.

1.1 Agents, Languages, and the Ara System

There has been, and no doubt will continue to be, a broad discussion about "what constitutes an agent", as opposed to a general "program" [FON93, FRG96]. This discussion has been both controversial and confusing, and apparently has not yet reached a consensus beyond a few buzzwords, "autonomous" being the most prominent among them. Ara's policy, stressing application independence, is not to take sides in that discussion. Instead, the Ara notion of an agent is simply derived from the more clearly defined notion of a *mobile* agent[1]. In Ara, a mobile agent is a program with the ability to move during execution. That is, besides mobility there is nothing new to a mobile agent. Of course, this could be called an intentionally misleading statement, since the very mobility of program code across different machines has truly far-reaching consequences for the design of the execution system for such code, most notably portability and security concerns. Both problems of portability and security are fundamental to mobile agent systems, portability being an issue

---

[1] Although, even agents which cannot really move like e.g. stationary servers are subsumed under the term "agent" in Ara. This is simply because it is convenient to have a common term for all active entities in the system.

because mobile agents should be able to move in heterogeneous networks, i.e. between machines with different operating systems and hardware architectures, to be really useful, and security being at stake because the agent's host effectively hands over control to a foreign program of basically unknown effect[1]. Most existing mobile agent systems, while differing considerably in practice, use the same basic solution for portability and security: They do not run the agents on the real machine of processor, memory and operating system, but on some virtual one, usually an *interpreter* and a *run-time system*, which both hides the details of the host system architecture and confines the actions of the agents to that restricted environment. This concept of a dedicated execution environment, providing a secure and portable set of services to access the host system and possibly other agents, enables agents to move in heterogeneous networks and permits a fine-grained control of the executing agent without depending on the hardware platform.

This is also the approach adopted in Ara: Mobile agents are programmed in an interpreted language and executed within an interpreter for this language, using a special run-time system for agents, called the *core* in Ara terms. However, the relation between core and interpreter is characteristic for Ara: Isolate the language-specific issues (e.g. how to capture the Tcl-specific state of an agent programmed in the Tcl programming language) in the interpreter, while concentrating all language-independent functionality (e.g. how to capture the general state of an agent and use that for moving the agent) in the core. This separation of concerns makes it possible to employ several interpreters for different programming languages together on top of the common, generic core. The core deals with general agents only, making its services, e.g. agent interaction, uniformly available to all agents regardless of their respective interpreter languages. Although this matter will be treated more thoroughly in section 5.1, "Processes and Internal Architecture", it should be remarked here that the complete Ara system of agents, interpreters and core runs as a single application process on top of an unmodified host operating system. Fig. 1 shows this relation of agents, core, and interpreters for languages called $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$.

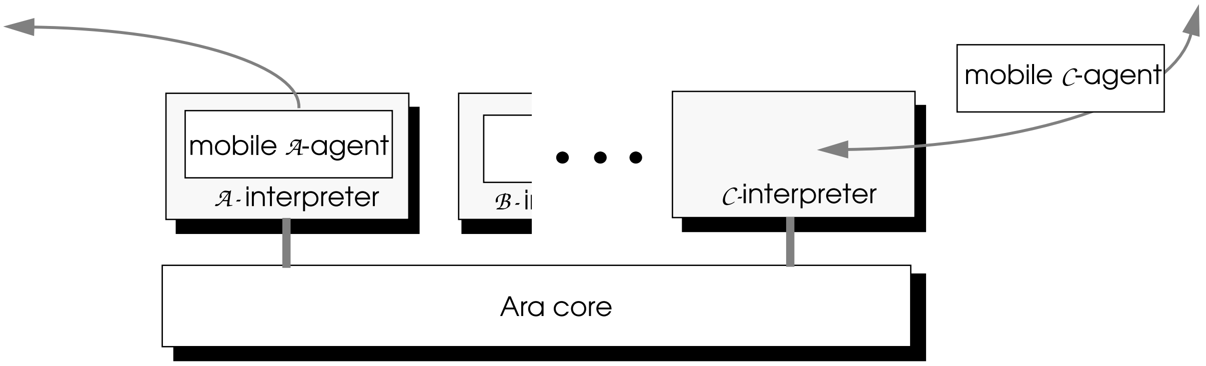

Fig. 1: Agents, Interpreters and the Core

While Ara stresses the independence of its concepts for mobile agents from specific programming languages, in practice the choice of languages is not irrelevant. At the time of this writing, interpreters for the *Tcl* [OUS94] and *C/C++* [STR90] programming languages have been adapted to the Ara core, and the *Java*

---

[1] There is also the reverse problem of the agent´s security against undue actions of the host, e.g. spying on the agent´s content. There is, however, no general solution for this problem except several individual aspects; see section 6.4, "Security and Payment" for a discussion of this.

[GJS96] language will be added next. Tcl and C/C++ differ considerably in complexity, run-time efficiency and development expense. Tcl, a popular scripting language, offers low development expense and a compact, embeddable, and freely available interpreter, which made it the language of choice of several mobile code systems [GRA96, LDD95, JRS95, OUS95]. C/C++, on the other hand, is superior regarding the available abstractions, the run-time efficiency and the interoperability with existing software. The reader may wonder that C[1] is used as an interpreted language here, since C as a typical compiled language is not directly suitable for interpretation. However, it was considered distinctly important to utilize the most widely-used programming language for mobile agents, saving the effort of adopting a new language and enabling the reuse of an enormous base of software. Accordingly, a dedicated interpreter for C was developed, specially adapted to mobile code. This interpreter works by precompilation to the portable MACE bytecode, which is subsequently interpreted with remarkable performance by the MACE interpreter [STO95].

The rationale of interpretation was support for portability and security, especially the portability of a live agent's execution state (see section 5.4). While there seems to be no viable alternative to interpretation when the full mobile agent functionality is desired, *compiled agents* were integrated into the Ara system as a native speed alternative for cases where certain security and portability requirements can be sacrificed. Such compiled agents cannot usually move to other machines, as they generally exist in a machine-dependent form only, and security cannot totally be warranted. In many respects, however, they behave like their interpreted siblings, at a substantially increased speed. Compiled agents are usually employed for services resident in a local system.

Ara thus offers the option to choose a programming language, preferably an existing one proven for the application at hand, instead of requiring all applications coded in one prescribed language, possibly even specially created. Prerequisite for this is that the desired language interpreter has been adapted to Ara; however, this adaption is well-defined and straightforward on the part of Ara (see section 5.6, "Adaption of Further Programming Languages to Ara").

The programming examples in this report will use Tcl most of the time, since it is more concise than C. C examples will be given from time to time to aid the distinction between language and concept.

1.2 The Life of an Agent

What makes an Ara agent program different from conventional programs is its characteristic use of functions provided by the core for agent actions, control and interaction. An active agent program in Ara is a *process*, i.e. a self-contained activity with its own state and progress. This seems both a natural and powerful[2] way for an agent, being supposed to act autonomously after all. Creation and deletion of such new agents, respectively agent processes, are among the most basic functions offered by the core. Each agent is assigned a unique, immutable name on creation. Newly created agents will execute their program in parallel with other agents active on the same system; they can clone themselves, suspend, resume and terminate their operation, sleep for some time, or be likewise acted upon by other agents, provided the necessary access rights. These agent control functions may be used to form a team of agents working on a common task.

---

[1] Unless explicitly stated, in the following text “C” is meant to comprise C++ as well.

[2] If this should raise concerns about performance, see section 5.1, "Processes and Internal Architecture" for an explanation of the efficient implementation of processes in Ara.

Although many useful tasks can be performed by a single mobile agent using the functions offered by the local host, considerable flexibility and new kinds of applications are gained by *interaction* between agents. Agents can „talk” to each other then, exchange information, offer and request services from one another, and even negotiate and trade. This concept is also useful for structuring a host environment, as higher-level services can be offered through „system” agents, which is more modular and flexible than offering the services through a static function call interface.

It can be argued whether agent communication should be remote or restricted to agents at the same place. Considering that one of the main motivations for mobile agents was to avoid remote communication in the first place, Ara emphasizes for local agent interaction. This is not to say that agents should be barred from network access (which depends on the policy of the host system interface, see section 1.4). Rather, the system encourages local agent interaction[1]. There are various options for such an interaction scheme, including disk files, more or less structured shared memory areas (“tuple space”, ”blackboard“), direct message exchange, or special procedure calls, each entailing different ways of access and addressing. Ara chose a variant of message exchange between agents, allowing client/server style interaction. The core provides the concept of a *service point* for this. This is a meeting point with a well-known name where agents located at the same place can interact as clients and servers. An agent announces the service point, thereby assuming the role of a *server agent*, whereupon *client agents* may meet it. A client may also attempt to meet a service point which has not been announced yet; such an attempt will either block the client until the announcement, or optionally return a failure indication. After a successful meeting, a client can submit service requests to the service point, and the server can fetch those and reply to them as it sees fit. Each request is marked with the name of the client agent, and the server may use that in choosing a reply. For example, a service point might advertise access to some data base as shown in fig. 2, and clients may submit queries, which are fetched by servers to be inspected, perhaps preprocessed, billed, or translated before being passed to the database management system. The query results are then passed back to the client as the reply to the service request.

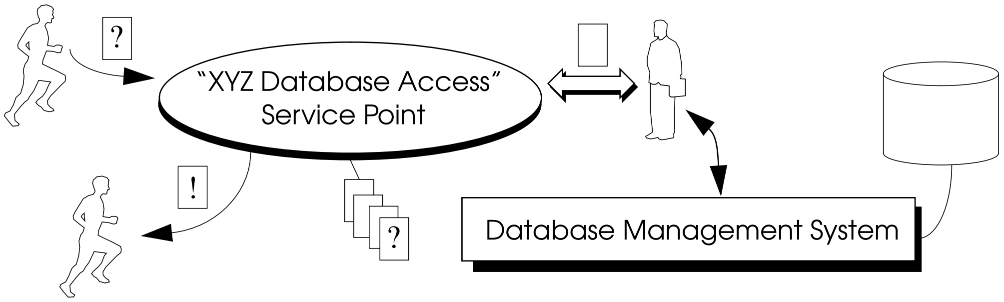

Fig. 2: Client and Server Agents at a Service Point

---

[1] In spite of this, a simple remote messaging facility between agents at known places will be added for pragmatic reasons. However, to avoid a tight remote coupling, this messaging facility will not involve itself in any guarantees against message losses.

Service requests and replies are implemented as messages with content and meaning entirely up to the participants. Many agents can meet at a service point. An agent can play the role of both client and server at different service points at the same time.

Agents on the move are programs running out of the sight and reach of their creator. It seems wise, therefore, to have a means of setting global limits to their actions, safe-guarding against unwanted effects like e.g. circling through the net in endless loops. The potential danger of agents getting out of control becomes even clearer when considering the bill certain hosts might charge visiting agents for the resources consumed during their stay, e.g. execution time. The hosts, on the other hand, have an even stronger interest in setting limits to visiting agents running on their system, in order to prevent overuse of host resources and to enforce resource agreements.

For this purpose of resource access limitation, Ara agents are equipped with resource accounts, called *allowances*. Depending on the type of resource, an allowance may be defined either quantitatively, e.g. an allowance to allocate some amount of memory, or qualitatively, as is the case with an allowance to read a certain file[1]. When the agent accesses a resource, the agent's allowance is checked and possibly updated accordingly by the system. The system ensures that an agent can never overstep its allowance. In the simplest case, an agent is given an initial allowance for its own perusal at the time of agent creation; however, a group of agents may also share a common allowance, each consuming from it according to their own policy. Agents may inquire about their current allowance at any time and may transfer amounts of it among each other. Such transfers can be used as an organizational measure in a group of cooperating agents, but also for trading resources between buyers and sellers. Most of the time, however, allowances are used by the creator of an agent to bound its global range of action, and by the receiving system to limit the agent's local resource accesses. In the latter case, the receiving place temporarily imposes a *local allowance* on an arriving agent, further restricting the agent's *global allowance* set by its creator, as sketched in fig. 3. Additionally, the agent itself may specify a desired local allowance when moving to a place (default for this is the full global allowance), and the target place decides what allowance to actually concede to the entering agent, or to deny access altogether. Anyway, on leaving the place, any local allowance is released and the global one takes effect again.

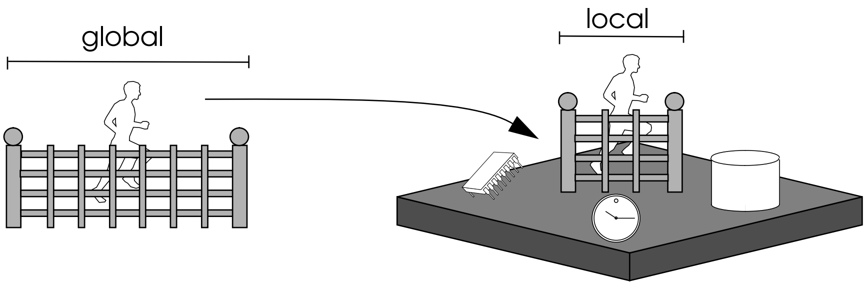

Fig. 3: A Place Restricting an Agent by a Local Allowance

---

[1] At the time of this writing, allowances can be specified for CPU time and memory consumption; other resources like disk space, files, network connections or places visited will follow.

Mobile agents lead a more dangerous life than ordinary, stationary programs. When moving through a large network such as the Internet, over computers with widely differing uses and policies, connected by links of all qualities and loads, they have to be prepared to all kinds of accidents happening. Agents may get blocked on the way, they may fall a prey to unexpected software faults, or their current host machine may simply crash, burying the agent with it. Some of these dangers can be countered by specific measures, but there is no general protection against an agent dying unexpectedly on its itinerary.

Rather than trying to anticipate all potential pitfalls, Ara offers a simple means of *recovery* from such accidents: An agent can create a *checkpoint*, i.e. a complete record of its current “internal”[1] state, at any time in its execution. Checkpoints are stored on some persistent media (usually a disk), and they can be used at a later time to restore the checkpointed agent. On restoration, the agent resumes from the exact state before the checkpoint. The obvious application for this scheme is for an agent to leave a checkpoint behind as a “back-up copy” before undertaking a risky operation, e.g. moving to unknown machines. In the event of an accident happening to the agent, it can be restored from the checkpoint and take appropriate recovery measures. The system will soon provide a facility to implicitly checkpoint all locally existing agents in the event of an emergency shutdown. Of course, the checkpointing mechanism may also be used for agents simply wishing to idle for a longer time at low resource requirements, or for working with “canned” agents. Checkpointing and restoring should be used with care, e.g. restoring an agent which is indeed still alive on a remote system would produce two copies of the same agent, requiring explicit treatment.

Having covered basic agent handling, agent interaction, limitation, and recovery, Ara’s concept of the one essential aspect of mobile agents, mobility, has only casually been treated so far. It is time now to take a closer look at that.

1.3 Agent Mobility: Going from Place to Place

Moving a program between computers can mean different things. In the simplest case, the program code is transported to its destination site prior to program start, and then run to completion there. It might seem that calling this a mobile agent system would be stretching the notion too far (note that this could basically be achieved by carrying a diskette with the program to the destination computer), but if the receiving system is aware of the foreign nature of the program and runs it in an appropriate environment, this qualifies as a mobile code system at the least. The Java language environment [GJS96] is an example for such a mobile code environment. From a point of view of mobility, the characteristic property of such mobile code systems is the restriction that a running program cannot move any further.

However, mobile agents frequently need the ability not only to be moved once from their origin to a destination site, but to move at their own decision, visiting several sites in a row, to fulfill a requirement which cannot be satisfied by the initial destination site (alone). This might be the case e.g. when collecting information from several sources, or when the final destination is not known beforehand, or when a site has to be left because it is shutting down, which is a common situation in mobile computing. For such purposes it is no longer sufficient to simply transport the program code, (re)starting the agent in its initial state. Rather, the

---

[1] This internal state should be thought of as the complete state of the agent minus its relations to external objects and resources such as service points, or files. Such objects cannot be warranted to persist until the agent is restored; consequently, they cannot be included in the checkpoint.

agent needs to carry additional information about its experience so far — technically speaking, about its execution state. Moving a running program, i.e. a live process of code and state, is usually termed *migration* in systems programming[1].

Ara agents can migrate at any point in their execution, simply by using a special core call, named `ara_go` in Ara's Tcl interface[2]:

```
ara_agent {ara_go moira; puts "Hello, world!"}
```

This creates a new agent, giving it a Tcl program (enclosed in braces) to execute. The agent will first move to a place named `moira` (a machine name, in this case) and then print the greeting message there. The `ara_go` instruction is all the programmer needs to know about migration — the system ensures that the agent is moved in whole to the place it indicated and resumed exactly where it left off, i.e. directly after the `ara_go` instruction. The complexity of extracting the complete agent from the local system, getting it to another machine and reinstalling it there, possibly even on a different architecture, is thus hidden in a single instruction, allowing the programmer to concentrate on the application rather than on the technical details of communication. Furthermore, the act of migration does not affect the agent's flow of execution nor its set of data, allowing the programmer to make the agent migrate whenever needed, without preparation or reinstallation measures, as illustrated in the following example:

```
set previousPlace [ara_here]
foreach place {moira kismet fatum} {
    ara_go $place
    puts "Hello at $place, I'm coming from $previousPlace!"
    set previousPlace $place
}
```

This agent visits the places `moira`, `kismet`, and `fatum` in turn, each time printing the place it came from[3]. The migrations are embedded in its flow of execution (a `foreach` loop in this case) without interfering with it, and all its data (`place` and `previousPlace` in this case) remains available untouched. This concept of non-interfering migration is termed *orthogonal migration,* in contrast to variants which do affect further execution. Section 6.3, "Mobility" discusses this issue further.

Note that while the internal state of a moving agent is transferred unchanged, this is not possible for its "external state", i.e. its relations to other system objects and resources like service points, files, or windows. Such objects are not mobile and thus have to be left behind when moving to another machine. It would be theoretically possible to add a software layer to such stationary resources making them appear as mobile, effectively creating a distributed operating system. However, the complex protocols and tight coupling involved with this approach do not seem well adapted to the low-bandwidth and heterogeneous networks targeted by mobile agents.

---

[1] Note, however, that "process migration", when used in operating systems context, nearly always refers to processes in tightly-coupled homogeneous systems (e.g. a workstation cluster on a local network) without security and portability problems. The two concepts are clearly related, but their focus is very different.

[2] The same effect could be achieved in C by calling a C function `Ara_Go()` etc.

[3] `$place` is the value of the variable `place` in Tcl syntax; `[ara_here]` is the result of the `ara_here` command (which returns the name of local place).

Ara agents move between *places*, which is both an obvious association to physical location and a concept of the Ara architecture. Places are virtual locations within an Ara system, which in turn is running on a certain machine; places could thus be said to refine physical location. More importantly however, an Ara place establishes a domain of logically related services under a common security policy governing all agents at that place. Service points, for instance, are always tied to a place and can be accessed only by agents currently staying at that place. In fact an agent is always staying at some place, except when in the process of moving between two of them. Places have names which make them uniquely identifiable and which can be specified as the destination of a migration. In practice, a place might be run by an individual, an organization, or a company, presenting its services. Fig. 4 illustrates the relation of systems, places and service points.

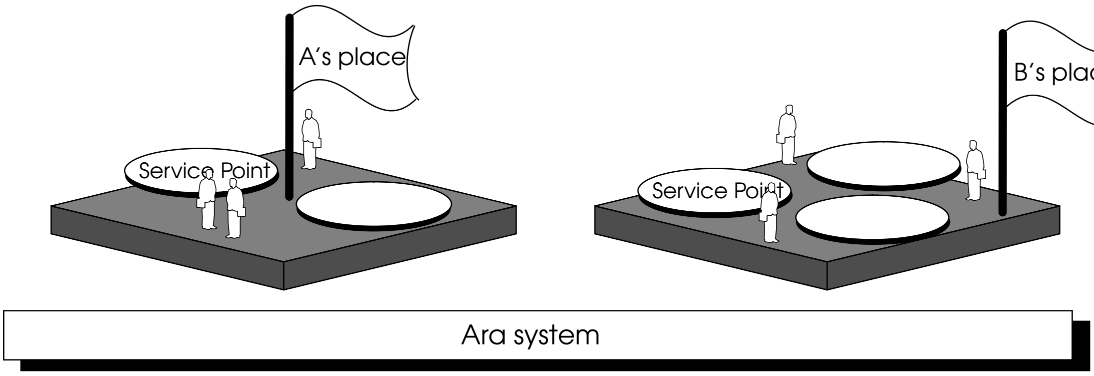

Fig. 4: Two Places with Service Points and Agents on one System

Besides structuring, places also exercise control over the agents they admit. First of all, a place has the authority to decide whether a specific arriving agent is really admitted, using e.g. the agent's name for this. An agent which has been denied access to its desired destination place is made go back to its source place, there to discover the failure. Thus, places play a central role in the security policy of an Ara system; in fact, each place can implement its own security policy to a large extent, requiring e.g. that arriving agents be authenticated (that is, as soon as authentication has been added to Ara) or that they have not passed through certain mistrusted places before.

Even if a place decides to admit an agent, it may impose a restrictive local allowance on it as described in section 1.2, limiting its potential actions while staying at the place. In effect, each place may create its own security domain and decide autonomously which agents to admit under what conditions. When the agent resumes after completion of the migration and admission procedure, it may check its effective allowance, discovering to what extent the place has honored its desires. This enables the agent to decide on its own what to do if it finds the conceded local allowance insufficient, rather than using some fixed negotiation mechanism built into the migration procedure.

In the current Ara implementation, places are not yet really distinct objects in the system architecture, and there is no application programming interface for places yet. Instead, there is one implicit default place provided per system. Accordingly, place names currently reduce more or less to machine names[1]. The default

---

[1] Actually, a place name currently designates a specific Ara system on a specific machine. See section 3.6, "Mobility" for how to exploit this.

place has a fixed behavior; it admits all arriving agents and does not impose local allowances on them, i.e. the agent is accepted subject to its global allowance. The next Ara release will restrict this generous policy, as well as provide the means to create new places with application-specific behavior.

1.4 Accessing the Host System

So far, the facilities of Ara have been described at the level of (mobile) agents. Apart from that, agents of course need to perform physical input and output as well, accessing the user interface, the file system, external applications and the network interface — in short, agents need access to the host system. Certainly the operating system of the host machine provides just that, but bearing in mind the increased security concerns inherent in access by mobile agents, such access has to be controllable in a more fine-grained way than is common with local programs. Additionally, the desired portability of agent programs calls for a somewhat higher level of access than usual.

In keeping with Ara's basic idea of adding mobility to existing concepts, the Ara host interface looks basically similar to the programming interface of common operating systems, restricted and simplified as necessary. Unfortunately, at the time of this writing, the host interface is still in various stages of development and not ready for distribution, parts being implemented, and others being designed. Therefore, this section will give only an overview of the forthcoming host interface as sketched in fig. 5.

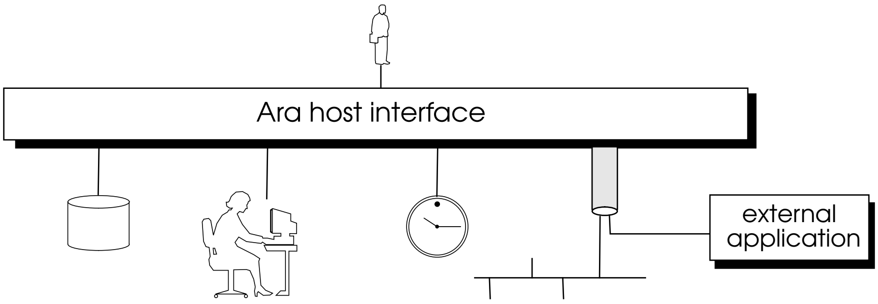

Fig. 5: Agent Access to the Host System

Ara agents will have access to a hierarchical *file system* with flat files, not much different from the interfaces of common file systems. The file system will build on that of the underlying operating system, presenting an enhanced view of it. The main enhancement, besides hiding platform dependencies, will be access control. Specifically, an agent's file accesses must be authorized through its allowance, and the accessed volume may be debited to the allowance again. A file may be visible only from a certain place with a restrictive admission policy, effectively constituting an access control list, selectively allowing or denying access to specific agents or classes of agents, e.g. those of a certain origin. Since files are frequently expected to be shared among agents, concurrent-read/exclusive-write locks on files will be provided for the synchronization of concurrent accesses. There are applications, e.g. electronic trade, where individual allowances for specific files are issued or transferred between agents, possibly restricted by volume limits or times of validity. Cryptographic certification would make such allowances, often called capabilities then, transferable and tradable across agents and systems.

A general concept of *communication channels* will be provided, both for network access and communication with local entities (see the external application interface discussed subsequently). Analogously to the file system, these channels will stay very close to conventional concepts with respect to reading and writing, but will be enhanced by access control features. Indeed it is expected that the channel interface will be integrated with the file interface, to utilize the access control facilities available there.

Interfaces from client agents to *external applications* on the local machine will usually be implemented by proxy agents — stationary, trusted agents dedicated to represent the external application to the agent system. A proxy agent usually supervises the security of all accesses by the client agent and performs potential translations between the two partners. If the client agent is trusted with respect to the external application, and if the application offers a means for external access (e.g. by TCP), the agent may also be allowed to communicate directly through a communication channel, without an intermediate proxy.

The Ara core provides timing functions in the form of a real time clock and synchronous delays. Asynchronous timed invocations will follow.

At present, while the host interface is not available yet, agents perform I/O using the standard facilities coming with their respective language run-time system as a temporary substitute, e.g. Tcl agents use Tcl I/O commands, and C agents use the functions of the ANSI-C library. While of course being inadequate from the point of view of security, this substitute is sufficient for all tasks which can be handled using the access security mechanisms imposed by the underlying operating system on the process containing the Ara system. Note, however, that blocking I/O operations should be used with care, since currently they may block the whole operating system process[1]. Both problems will be resolved in the forthcoming Ara host interface implementation.

2. Mobile Agent Applications with Ara

While mobile agents are a fairly general concept for networked computing, there are classes of applications which benefit characteristically from them. Most prominently, these are applications where a significant amount of data has to be produced (consumed) which is, however, needed (available) far away from its producer (consumer). The prominent example of the remote consumption case is *information research*: A client searches for information which is available scattered throughout the stock of one or more remote servers. *Information pr*