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.

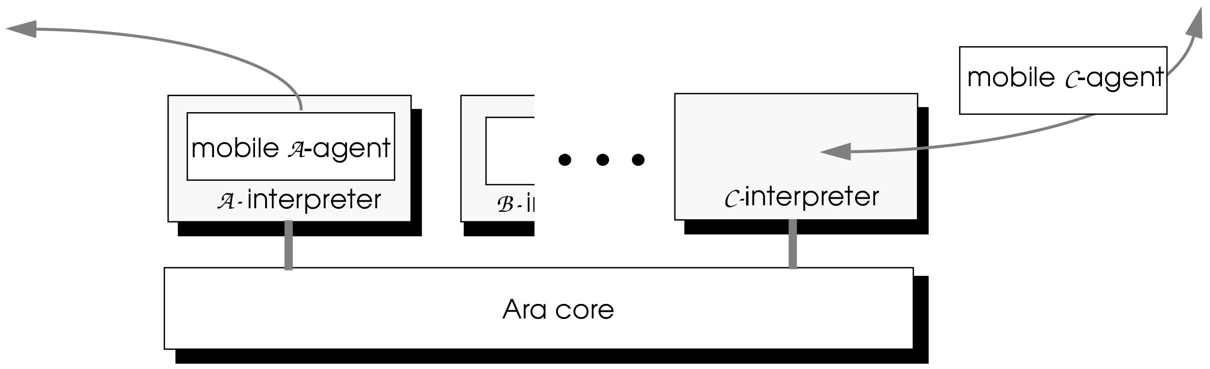
1.1 Agents, Languages, and the Ara System

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 $\mathcal{C}$.



mobile $\mathcal{A}$-agent
$\mathcal{A}$- interpreter
$\mathcal{B}$-i
• • •
$\mathcal{C}$-interpreter
mobile $\mathcal{C}$-agent
Ara core

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.

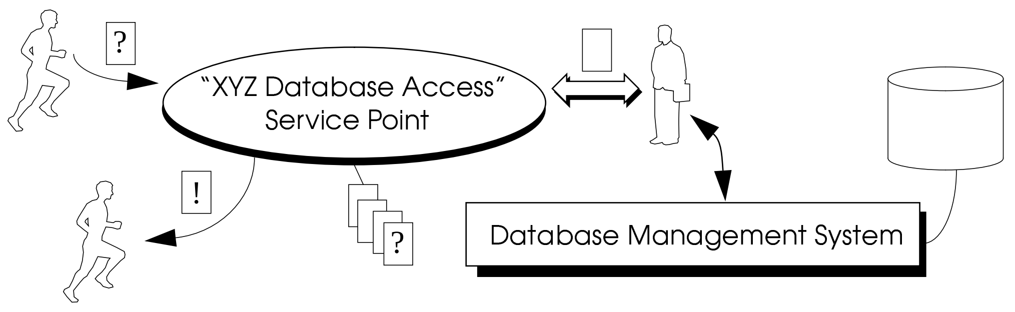

?
"XYZ Database Access" Service Point
!
?
Database Management System

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.

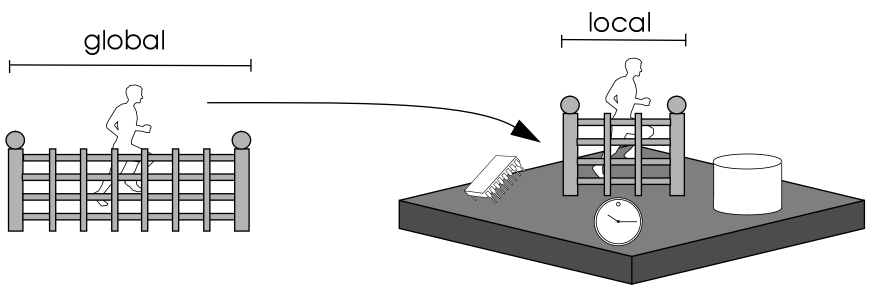


Fig. 3: A Place Restricting an Agent by a Local Allowance

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.

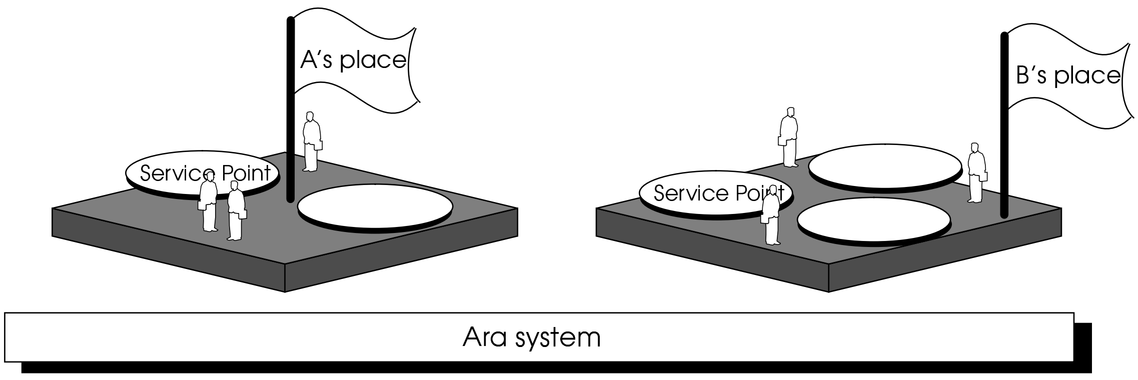


Fig. 4: Two Places with Service Points and Agents on one System

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.

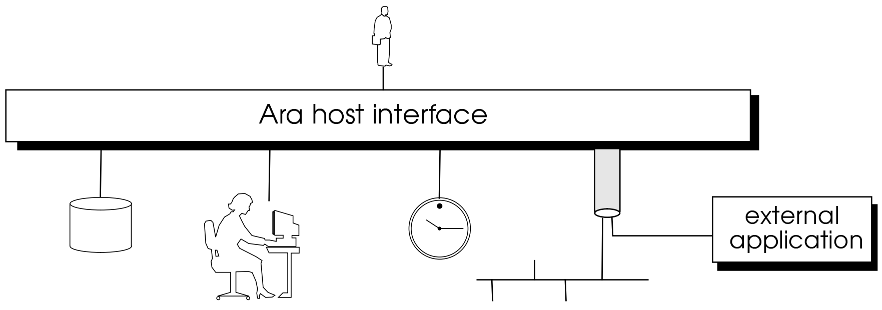


Fig. 5: Agent Access to the Host System

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 presentation* is the central example for the remote production case, where a server produces presentation information, e.g. a sophisticated graphical dialogue, needed by a remote client. In both cases, rather than transporting the data to its remote destination, an agent can be sent to produce respectively consume the data directly at the location of the data, exploiting the ability of mobile agents for computing on site.

Another important area of application is found in *mobile computing* [FOZ94]. In this domain, the ability of mobile agents to change their host machine during execution can be exploited to let a computation dynamically adapt its location to changing conditions, e.g. a mobile unit entering or leaving an area or a host shutting down.

We will now look at these application areas more closely, illustrating how Ara agents could be of help there.

1. Reading from the standard input stream, e.g. a terminal when in interactive mode, does not exhibit this problem.

2.1 Information Research

An agent wandering through the net, looking around, picking up items of interest to its principal, perhaps occasionally performing a transaction — this is probably the most intuitive and appealing picture that comes to mind when thinking of mobile agents. Indeed agents are well-equipped for such tasks. An Ara application in need of information located at some remote site would send a search agent there, to meet a server agent managing that information. If the place of the applicable service point is not known beforehand, the search agent would consult the local directory service[1] first. The search agent then goes to the destination place and meets a server agent at the concerned service point. It proceeds to submit requests to the server, asking for the desired information and receiving replies. The search agent may perhaps consult more than one service point, relating and combining the replies. If the retrieved data indicates that it would be worthwhile to extend or move the search to other machines, the search agent makes a note of this and later goes to look there, or it creates a clone of itself to treat that other site in parallel — in fact, arbitrarily sophisticated search strategies can be encoded in the agents. Ara agents benefit from their ability to smoothly migrate and clone when performing such tasks distributed across the network.

In many cases, the search agent is actually looking for data more specific than the server interface allows to specify, e.g. the agent might ask for a text file, but is actually interested only in a certain part of it, or even not interested at all unless the content satisfies a certain condition. A search agent is well prepared to handle such information filtering, e.g. by parsing a text or relating the information to previous findings. It will later bring or send back to its principal only the actually desired information, eliminating the transfer of unnecessary data. This is why mobile search agents are especially effective for searching data with complex and irregular structure, such as natural language texts or graphical images: They can bring their own data analysis and filtering methods with them, tailored to and optimized for the specific need, with a gain that increases with the structural complexity and the size of the data. In contrast, a remote search would be confined to the comparatively primitive standard filtering methods (if any) offered by servers, such as scanning texts for key words, and would usually entail the transfer of large amounts of data to the client site to perform the processing there. When it comes to concrete programming, Ara's capability of handling various programming languages proves helpful here, employing, say, a Tcl package for text parsing, while reusing an existing C module for image processing.

At the time of this writing, a realistically-sized mobile agent application for researching Usenet news [KAL86, HOA87] using the Ara system is in development. Usenet is a network of autonomous server nodes, each storing news articles received from its neighbors and passing them on to the other neighbors. A user is attached as a client to some local server and may read the server's stock of news articles, or write an article of his or her own, which is then included in the server's stock and propagated along. Articles which have been in stock at a server for a certain time are automatically discarded to prevent overflowing the server's storage. Usenet nodes are autonomous in that each maintains its own stock of articles, propagating and, most importantly, discarding them according to its own policy, mirroring the preferences of the people in charge of the server. Usenet information is thus distributed across the whole network, with each node possessing a specifically local and constantly changing picture of it. This poses a problem when a user needs to access a

1. This will be included in a future release of Ara (see annex B, "What May Be Expected from Ara in the Future?").

set of articles only partially in stock at the local server, e.g. when desiring to read all other articles preceding or referring to a certain interesting article of a current discussion found locally. Some of those other articles may not (yet) have reached the user’s node, and yet others may have been automatically discarded already.

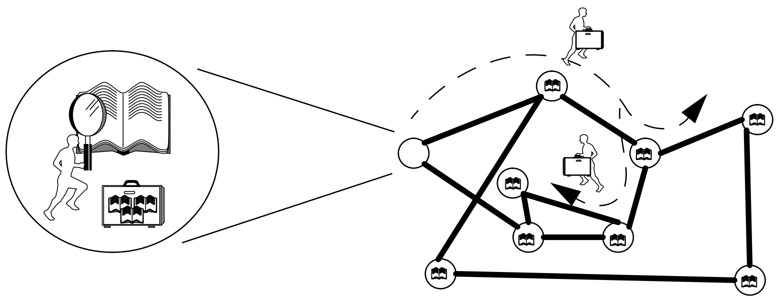


Fig. 6: Agents Searching Usenet for Interesting Articles

Fig. 6 illustrates how Ara agents are used to solve this information research problem. Usenet nodes are equipped with Ara systems providing stationary server agents for access to the local news server. A user at some node may specify a query for news articles, typically those related to a certain article by topic or explicit reference, which is used to launch a search agent trying to collect the articles not locally present. The agent directs its search according to the content of its previous findings, exploiting the format of Usenet articles, which bear information about their itinerary and references in a header. The agent will track that itinerary, visiting servers[1] on the path in the hope of finding further relevant articles which a server has either not yet forwarded or not yet discarded. Cross references linking articles will also be exploited by dynamically creating sub-agents to track those recursively. All search agents will be equipped with a sufficient allowance for their task and will constantly monitor it to prevent getting lost in the network. At the end of their search, all agents return to their principal, bringing home the collected articles[2]; during the search, the principal does not need to remain on-line. Without mobile agents, searches of this kind would have to be performed on-line, and large amounts of articles would have to be transferred to the home site and processed there. Moreover, in practice most Usenet servers are not configured for remote access at all.

As a matter of fact, the presented method of searching Usenet is an instance of *semantic routing* [CGH95], a concept for distributed information access where a mobile object (e.g. a message or an agent) directs its path through one or more intermediate relays according to the semantics (i.e. the content or goal) of the object, approaching the final destination on each step. This is a valuable concept when dealing with information the location knowledge of which is distributed itself, providing a flexible and robust method to cope with the constantly changing structure of the global information space. Mobile agents are well-adapted to semantic routing thanks to their ability to decide locally about their further plans and itinerary.

1. Technically precise, visiting the Ara system on the server node.
2. Actually, it is not necessary to transport the articles themselves, since Usenet articles are tagged with globally unique identifications; it is sufficient to collect these identifications and retrieve the corresponding articles at the end of the search.

2.2 Information Presentation

Reversing the perspective, mobile agents can also be applied to remotely produce information rather than collect it. Not surprisingly, agents are beneficial wherever large volumes of data are to be produced, or when the produced data depend on interactive user input, since in both these cases a remote communication solution would make heavy use of a remote connection, which is dispensable with mobile agents. Both requirements of volume and interactivity are clearly combined in remote multimedia presentation: Video and audio must be displayed to a user in real-time, and user input has to be perceived and reacted upon instantly. Obviously mobile agents, travelling with the data and performing display and input processing at the user's site, are superior to any remote communication solution.

In an application scenario, a server publishing multimedia documents (e.g. enriched WWW pages) would enclose a custom-written agent to present the data on the client site in an arbitrarily specific fashion. Once a client retrieves and views the document, the presentation agent starts and handles access to the document, using the client machine's services and devices in any way it sees fit to present the information, possibly engaging in user interaction and taking advantage of remote resources behind the scenes. Examples of such presentations include interactive documents, multimedia documents in custom formats (e.g. CAD data), and network-aware documents. Once again, it will be helpful if the agent can be programmed in a language adapted to the specific task. An important practical advantage of using a customized mobile agent for presentation is the independence from presentation standards (such as HTML [PRI96]) which inevitably lag noticeably behind the technical possibilities. Regarding the enormous growth of the WWW [BER94] as a document publishing medium and the demand for multimedia documents, it is no surprise that using some form of mobile code for document presentation has recently attracted enormous interest, as witnessed e.g. by the Java language applied in the *HotJava* [SUN95] web browser and similar tools [ROU96, THB96]. Most of these systems do not provide migration; however, presentation agents usually do not move further than their target destination, being targeted more specifically towards one site than e.g. information research agents. Accordingly, mobile code systems without migration are hindered by the lack of this function much less noticeably in information presentation than in information research applications.

2.3 Mobile Computing

Quite apart from specific applications, mobile agents are a generally useful concept for mobile computing. As a matter of fact, this consideration has initiated the Ara project, and the use of mobile agents for mobile computers equipped with wireless links is investigated in the project. Actually, the "mobile" name coincidence here is not a pun, but hints at a real benefit for mobile computers, since some problems caused by hardware mobility can be compensated using the "software mobility" provided by mobile agents. However, the most prominent benefit of mobile agents for mobile computing is rooted in a generally better adaption to interaction over wireless links. Sending an agent avoids many of the problems of the predominant physical medium between mobile computers which the traditional model of remote message passing falls a prey to, as sketched in Fig. 7.

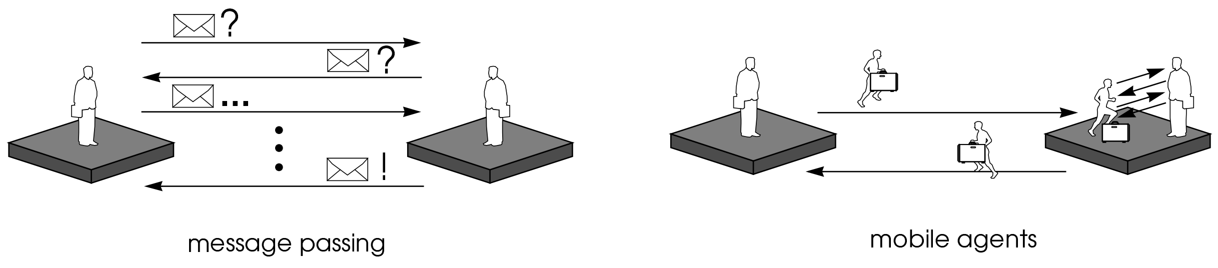


Fig. 7: Connection Usage of Remote Interaction Concepts

Passing messages over a network between computers effectively creates a remote interaction on a logical connection. This concept, frequently implemented by remote procedure calls, has achieved pervasive use throughout stationary computing networks, and has accordingly been applied to mobile interaction as well. However, the appropriateness of message passing relies on some implicit assumptions which become questionable with mobile computers. As a first point, the presence of an ongoing logical connection between the participants assumes a relatively reliable underlying communication network to keep it up for a sufficient time; even more, such a logical connection of course presumes in the first place that the participants themselves are available for the duration of the interaction. Furthermore, serving stock communication needs like information filtering and information presentation by remote data transfer assumes reasonable bandwidth available as seen in the previous sections. On the whole, remote message interaction assumes a relatively tight coupling between the machines.

However, none of those assumptions of reliability, availability and bandwidth generally hold for mobile computing systems. Wireless links, especially wide-area ones, are subject to practically inevitable disruptions, noise and bandwidth limitations; mobile devices are inherently short of energy due to the limited acceptable weight of their batteries. This invariably causes frequent and long-lasting switch-offs, reducing availability. On the whole, mobile units are too *loosely coupled* to let remote message passing still appear an appropriate basis of mobile computer interaction.

Mobile agents, on the other hand, are better prepared to these problems, since they achieve a *decoupling* of the interaction partners by eliminating the ongoing remote connection. The agents perform interactions coded into their program asynchronously and completely local on the destination site, as opposed to a conventional remote and synchronous dialogue. Of course this is particularly welcome for mobile computers, as it needs less bandwidth, is less vulnerable to connection problems and does not require the origin device to be available during the interaction. As a matter of fact, mobile agents take advantage of the available stationary resources on behalf of the mobile unit by dynamically moving a critical portion of the mobile application to a stationary site.

Besides this general reduction of dependence on the connection, even up to completely disconnected operation, mobile agents also provide solutions for problems caused by device mobility itself. Their ability of dynamically changing their place of execution enables applications like mobile unit substitution and *location-*

specific adaption. When a mobile unit is geographically moving through a wide-area network, say its user is walking with it through a city area, it will encounter diverse organizational domains, such as office buildings, shopping malls, hospitals, public authority buildings, or company premises. Each such domain may offer specific local services to guest computers entering the area, ranging from display services like directories, maps, news quotes or product presentations, over interactive services like navigation and inquiries, up to client components of applications like office automation systems. Mobile agents can be used to deliver the necessary interface software to entering computers as illustrated in fig. 8, automatically adapting them to local customs and opportunities. Such usage of mobile agents highlights their potential to resolve unexpected interface or function mismatches between interacting systems, effectively realizing a kind of dynamic, remote configuration.

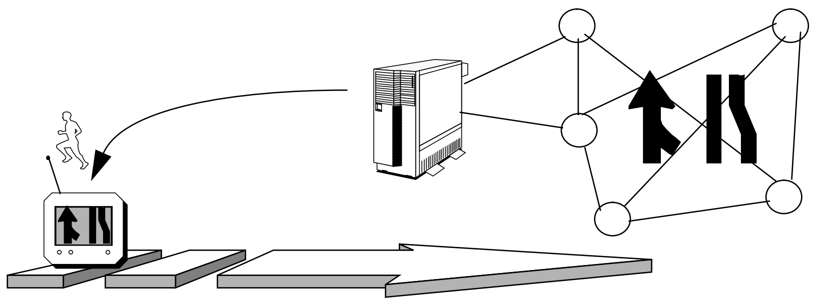


Fig. 8: Sending a Local Service Interface Agent to an Entering Mobile Unit

Mobile computing networks typically comprise a stationary part of fixed machines connected by wired links, and a mobile part, using wireless links to mobile devices. As the stationary part provides better availability and connectivity, it is a straightforward idea to shift some functionality from the mobile to the stationary network. Ara agents, being able to move dynamically and smoothly, are particularly suitable for such shifting. Agents staying in the stationary part of the network can act as *substitutes of mobile users* or applications while their principal is not reachable or prefers not to be contacted, as depicted in fig. 9 [BBI+93]. A straightforward application for such a substitution is e-mail handling: An agent positioned in the stationary network, on the path to its principal, may screen the latter's incoming e-mail messages, acknowledge receipt, forward selectively, and even reply to certain messages autonomously, not unlike a secretary. The principal machine may safely be switched off meanwhile. The agent may be positioned explicitly by its principal as well as change its location autonomously, e.g. move when the agent's current host is about to be switched off or requests it to leave. If continuing availability during a transient switch-off period is not required, an agent may also use the checkpointing facility to store itself safely meanwhile, rather than moving to another machine.

Besides improving availability, a representative agent at the border of the stationary network can also save on transmission costs, which can be substantial over wireless links. To this end, the agent can be used to *condense* the data flowing to its principal[1]. Condensation is used here as a rather general notion, to be further defined relative to the concrete application; condensation could mean e.g. compression of image data, or, in

1. Analogously to the duality of information production and consumption described at the beginning of this section, a substitution agent can be used to "*inflate*" data flowing *from* its principal as well.

the case of e-mail processing, handling certain messages autonomously, while forwarding only the rest to the principal. Transmission costs and delays can be cut further if the agent strives to stay close to its principal machine, moving along the border of the stationary network while tracking the moving principal. Incidentally, the mobility notions involved in mobile agents and mobile computers actually meet here.

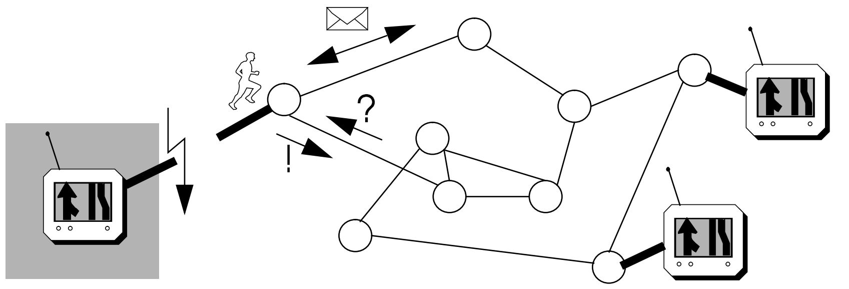


Fig. 9: A Mobile Agent Acting as the Substitute of a Mobile Unit

As a demonstration application in mobile computing, a *meeting scheduling* system for mobile users will be realized using Ara, employing mobile agents as substitutes of a human user. The application will offer support for scheduling meetings with other participants, using mobile agents which strive to best match the meeting preferences of their human principals, and react autonomously to switch-offs and movements of the mobile user, e.g. by changing to another host. The application will make economic use of wireless communication and handle crashes of participating machines. It will offer a graphical user interface and will be tested in practical operation on mobile computers within the Ara project group.

3. Ara Programming Concepts and Features

Now that the basic concepts of Ara have been described and put into perspective with applications, the stage is set to explain how these concepts appear concretely to the mobile agent programmer. This section will describe the Ara core application programming interfaces for the two languages currently available for Ara, Tcl and C[1]. The description will be set at a significantly more practical level than the other sections (except section 4) of this report. It will not elaborate each and every detail; please turn to the on-line help pages in the Ara software distribution for further information.

3.1 General Conventions of the Ara Core Interface

Before going right into programming, some preliminaries have to be settled. Ara agents access system functionality by calls to the core application programming interface (API), a set of functions provided by the Ara core for use by agents. Technically, these functions are compiled native code contained in each Ara system, each being accessible from an agent through a calling interface (called a *stub*) in the respective language, which is part of the language interpreter (see section 5.6). It should be noted here that the overhead of a core call, although it reminds of an operating system kernel call, is no more than one procedure call

1. Note that the MACE interpreter accepts C++ as well, but at the time of this writing, the Ara core offers a C function interface only (a C++ class interface will follow in one of the subsequent releases).

(namely, the stub). The various stubs for the same core function differ in syntax, but not in semantics. The stubs for use by C agents (remember Ara uses a C interpreter for these) are usually named identical to the native core functions, e.g. `Ara_ServicePointCreate()`, since there is a nearly exact correspondence between stubs and core functions here. The Tcl stubs, on the other hand, have their own names, partly due to the "object oriented" calling syntax in Tcl (see section 3.5.2 for an example). To distinguish Tcl and C syntax in the following, C function names are prefixed with `Ara_` and suffixed with parentheses like `Ara_Go()`, while Tcl command names are prefixed with `ara_` as in `ara_go`. Compiled agents may use the core functions directly, without stubs.

Note that the term "C agent" denotes an agent originally written in C and compiled to the MACE bytecode, which now exists in this interpreted bytecode form only. This should not be confounded with "compiled agents", which are usually written in C as well, but exist in native machine code form.

The Ara core assigns internal identifications ("ids") to all system objects, and applications reference such objects only through these ids, in order to protect the objects from undue accesses (see section 5.3, "Protection")[1]. In the C interface, the type `Ara_Id` is provided for object ids. This is the return type of all functions returning a system object, e.g. `Ara_ServicePointCreate()`. There is a special value of this type, `ARA_ID_NONE`, designating an invalid id and used to indicate an error return from such a function. The corresponding Tcl commands throw an error (see below) instead of returning a special value, as is the custom in Tcl.

While it is assumed that the reader has sufficient knowledge of C, one aspect of the Tcl language should be pointed out here: Tcl's only data type is the character string; accordingly, when types such as `Ara_Id` are mentioned in the following, this refers to the C interface, while the Tcl interface uses a corresponding string representation of the respective type[2]. For example, the C stub of the core function to restore an agent has the signature

```
    int Ara_Restore(Ara_Id agent, int mygroup);
```

expecting an object id and an integer (actually, a boolean value) as a parameter, while returning an integer as the result. The corresponding Tcl stub, on the other hand, expects and returns only strings, interpreted as encodings of the various values. Additionally, Tcl allows default parameters, so the Tcl stub corresponding to the C function above has the signature

```
    ara_restore ?-mygroup? ?<agent id>?
```
[3]

where e.g. `<agent id>` is Tcl's string representation of the agent's object id, and the command's result is the string encoding of an integer. Note that optional arguments, while syntactically required to appear in the C stub, are still called "optional" with the understanding that special values (usually `0` or `ARA_..._NONE`) are used when these arguments are meant to be ignored.

1. Compiled agents, however, may be exempted here — see section 3.9.
2. Please turn to the on-line help pages of the concerned command for the string representation of a specific type.
3. `?...?` denotes an optional term in Tcl interface specifications, since the more familiar `[...]` is already used for command result substitution in Tcl.

A final preliminary remark concerns the error behavior of the API functions. In the C interface, the stubs exhibit the so-called standard error behavior unless stated differently, i.e. they flag a potential error by a special return value, usually an integer error code[1], which is `ARA_OK` on success and `ARA_ERROR` otherwise. In the Tcl interface, the corresponding stub throws an error and returns an error message as the result. This error message can be retrieved in C as well by means of calling `Ara_GetError()` after the erroneous call. The resulting message is valid only until the next call to an API function (including `Ara_GetError()`). This might look as follows:

```
if (Ara_Go(...) != ARA_OK) {
   printf("Cannot go ... due to the following error: %s\n",
          Ara_GetError());
}
```

or, in Tcl, like this:

```
if [catch {ara_go ...} errmsg] {
   puts "Cannot go ... due to the following error: $errmsg"
}
```

Note that the `catch` command is used in Tcl to catch errors thrown during execution, optionally along with the error message. It is used here to catch any potential errors thrown by `ara_go`. An uncaught error in Tcl propagates backwards through the calling chain, terminating the current execution if no `catch` command is encountered.

3.2 The First Steps

Ara agents are processes, as introduced in section 1.2, “The Life of an Agent”. When the Ara system starts up, there is one initial process running (the “root” process)[2]. This is the process which prints the initial prompt to the terminal when the system is run in interactive mode, and which reads and processes the initial input. Interactive mode is the default; when the system is started with a file name argument, it runs in non-interactive (or batch) mode. Input in batch mode comes from the indicated file, and the system terminates after reaching the end of the file. In interactive mode, on the other hand, input is read from the terminal until the `ara_shutdown`[3] command is entered. In both modes, the system is also terminated if the root process terminates. For the purpose of this description, interactive mode is assumed, as this is more instructive.

The root process is a Tcl process, which is more convenient for interactive experimentation than e.g. C. Any standard Tcl[4] command (see [OUS94]) can be typed to the prompt, e.g.

```
% foreach fruit {orange banana} {
+% puts $fruit
+% }
orange
banana
%
```

1. ... or a special “null” value of the return type, e.g. `ARA_ID_NONE` etc.

2. To be precise, there are actually several more processes running, but these are not visible to applications.

3. This command is intended for easy interactive termination only; hence it is not available in batch mode.

4. Tcl version 7.4 at the time of this writing.

The user typed a `foreach` command to the `%` prompt, which is a loop over a list of values in Tcl, and the system executed the command, printing out the list elements (as the two lines without a prompt). Note that a `+` in the prompt indicates that some enclosing command is not yet typed completely.

By the way, there is another version of the prompt which indicates the process currently connected to the terminal. The `ara_toggle_prompt` interactive command is used to alternate between the two prompts:

```
% ara_toggle_prompt
2213942038,1% puts Hello!
Hello!
2213942038,1% ara_toggle_prompt
%
```

Here the figure in the prompt is the printed representation of the current process's object id. This can be helpful when several processes are connected to the same terminal intermittently.

3.3 Basic Agent Handling

3.3.1 Agent Creation

There is one function to create an agent for each agent implementation language. For the sake of uniformity, compiled agents are subsumed here as well; they can be thought of as being implemented in "native language". A Tcl agent is created from Tcl by the `ara_agent` stub command, giving it a Tcl script argument to execute, and trailing optional arguments which are passed on to the script as its command line arguments:

```
ara_agent <script> <arg>*[1]
```

This will create a new agent running in parallel to the other agents in the local system, executing the indicated Tcl script[2]. The newly created agent (in the form of its id) is returned as the command's result.

The agent command might be used like this:

```
set slave [ ara_agent {
               puts "I'm the slave, my argument is [lindex $argv 0]." [3]
            } \
            foo
          ]
puts "I'm the master, created slave $slave."
```

This would print something like

```
I'm the master, created slave 2213942038,5.
I'm the slave, my argument is foo.
```

where `2213942038,5` is the string representation of the newly created agent's id.

1. s`*` denotes zero or more repetitions of s.
2. The actual command, like many of the following commands and functions, has some additional option arguments which are, however, not described here in order to let the central concepts appear more clearly. The full descriptions can be found in the on-line help pages.
3. This is a Tcl expression resulting in the value of the first argument of the script.

Most of the time, agents implemented in a specific language will create only agents of the same language, but agents of different languages may be freely mixed as well. For instance, there is a C stub function equivalent to the `ara_agent` command, which a C agent might use to create a Tcl agent:

```
Ara_Id  Ara_CreateTclAgent(char* script, int argc, char* argv[],
                           Ara_Allowance global, Ara_Allowance local,
                           int size, flags);[1]
```

Of course both the Tcl stub and its equivalent C stub map to the same native core function behind the scenes.

Analogous to creating Tcl agents, there is another pair of stubs to create C agents. They are

```
ara_mace_agent <file name> <arg>*
Ara_Id Ara_CreateMaceAgent(char* fileName, int argc, char* argv[],
                           Ara_Allowance global, Ara_Allowance local,
                           int size);
```

Here the indicated file contains the MACE bytecode the new agent is to execute; again the optional arguments are passed to the agent as command line arguments.

Compiled agents may be created specifying the name of a file and function *f:g* which the new agent is to execute. The function *g* is expected to be defined in an object code file with base name *f* and a platform dependent file name suffix. Some arbitrary parameter data for the new agent can also specified with the stubs:

```
ara_comp_agent <function name> [<arg>]
Ara_Id Ara_CreateCompAgent(char* fileAndFunction,
                           byte* data, int dataLength,
                           Ara_Allowance global, Ara_Allowance local,
                           int size);
```

The specified data is a memory area intended to hold parameter data for the new agent. When further languages will be added to Ara, these will introduce additional agent creation stubs in the various languages similar to the above.

It is often convenient for an agent to know its own id. The `ara_me` function returns just this:

```
ara_me
Ara_Id Ara_Me();
```

The agent creation API is completed by the stubs for agent cloning. When cloning, an agent creates a copy of itself, duplicating its internal execution state[2]. Both agents return from the stub; however, the result returned in each case discriminates the two copies: In the original agent, the (id of the) new agent is returned, while `ARA_ID_NONE` respectively the empty string is the result in the new agent. The signatures reveal that the cloning function is named in reminiscence to the fork system call in the Unix operating system:

1. The `flags` and `size` arguments correspond to optional arguments of the `agent` command not of interest here; the optional `Ara_Allowance` arguments will be explained below.
2. This implies that the *external* execution state is *not* duplicated (see section 1.3, “Agent Mobility: Going from Place to Place” for an explanation of this difference). This is a technical restriction explained in section 5.5, “Cloning and Checkpointing”, which may be abolished in a future release. So far, it may be mended by the new agent explicitly recreating the external state, which should be possible to the largest part, because the external environment has not been changed by the cloning procedure.

```
ara_fork
Ara_Id Ara_Fork(Ara_Allowance global, Ara_Allowance local, int size);
```

Note, however, that this function is used much less frequently in Ara than in Unix, since the Unix function is often used to create an actually different process (by subsequent replacement of the new process's memory image), which can be achieved more simply in Ara, using an explicit agent creation function.

One final feature of agent creation to be noted here is the treatment of agent allowances. Remember that each agent has a global and a local allowance limiting its resource consumption during its lifetime and at the local place. At the time of creation, the allowances of an agent are determined, and there are two ways to do this: In the first, the new agent is not given a private allowance, but made to share a common allowance with the creating agent. Such agents sharing a common allowance form an agent group. A group lives on a common allowance, distributing it among the group members according to their own policy. Shared allowances are the default on agent creation. In contrast, the second way to determine a new agent's allowance is to give the agent its own private one. This allowance must be specified explicitly on agent creation, and it is deducted from the allowance of the creating agent; the latter agent effectively transfers some of its own allowance (which may come from a private or group allowance) to the new agent. In this case, the new agent might be viewed as forming a group with only one member. The initial root process is a member of a special group with unlimited allowance. A private allowance can be specified in C by setting the `Ara_Allowance` parameters of the creation stubs to non-default values; in Tcl, this is done by giving optional arguments, e.g.

```
ara_agent ?-la <local allowance>? ?-ga <global all.>? <script> <arg>*
```

Note that both the global and local allowances of the new agent must be less than or equal to that of the creating agent, and that the local allowance must of course be less than or equal to the global one. This concept ensures that the total allowance of an agent does not change by distributing it among various subagents or partner agents, so that an agent cannot acquire unauthorized allowance by any means.

As a matter of fact, agent groups are a more fundamental concept in Ara than simply joining agents with a common allowance. Each agent belongs to exactly one group at a time, and the group defines most of the agent's reachability in the system. Groups may form a hierarchy: Whenever an agent is created with a private allowance as described above, the new group becomes a subgroup (or child group) of the creating agent's group. An agent has a general "access right" over all agents in its group and in its group's child groups, grandchild groups etc. This access right over an agent is not detailed any further, i.e. it entitles its holder to any operation whatsoever over this agent.

An agent will leave its group when terminating or moving on to another place; in the latter case, it will find itself the only member of a new isolated group at the new place. When the last member of a group terminates, the group is implicitly deleted and the group's residual allowance is returned to its parent group, if that exists. Note that the current Ara implementation does not provide means to explicitly leave and join a group; this will be available in the next release.

The set of all agents currently existing in the system can be retrieved using the following stubs:

```
ara_agents
int Ara_Agents(Ara_Id** buffer, int* bufferLength);
```

3.3.2 Agent Termination

An agent may terminate voluntarily at any time by exiting. Agents may also make others terminate by killing them, provided the necessary access right. In both cases, an integer value may be left behind as the terminated agent's result.

```
ara_exit ?<value>?
void Ara_Exit(int value);
```

```
ara_kill <agent> ?<value>?
int Ara_Kill(Ara_Id agent, int value);
```

A parent agent may retrieve the termination value of another agent by waiting for it; obviously a successful waiting also indicates that the other agent has indeed terminated. Waiting is possible both for a specific agent, giving its id, and for any agent at all, which is the default.

```
ara_wait ?<agent>?
int Ara_Wait(Ara_Id agent, int* processResultPtr);
```

To end the root process (and thus the whole system) in interactive mode, the `ara_shutdown` short-cut can be typed to the prompt.

3.3.3 Agent Scheduling

Having populated the system with agents, some functions are needed to control their parallel execution. Ara agents are governed by a simple time sharing process scheduling model where each agent process is one of three states at a time: *Running*, *ready*, or *waiting* (also called "blocked"). There is always exactly one running process; the others are either ready, meaning they will become running as soon as their they receive a share of execution time, or they are waiting until they are set ready again. The scheduling states are usually managed implicitly by the core, but there are also functions to change an agent's state explicitly, provided the executing agent has the access right over the concerned agent.

In particular, *suspending* an agent means setting it to the waiting state (if this is the running agent, an immediate switch of control occurs), while *activating* a waiting agent sets it to the ready state. A running agent may also set itself to the ready state. This is called *retiring*, effectively performing a voluntary, temporary release of control. The stubs for scheduling read as follows:

```
ara_suspend ?<agent>?  ;# defaults to the running agent
int Ara_Suspend(Ara_Id agent)
```

```
ara_activate <agent>
int Ara_Activate(Ara_Id agent);
```

```
ara_retire
int Ara_Retire();
```

The core enforces time sharing between the agent processes by preemption[1], suspending the running process whenever its share is used up and setting one of the ready processes running[2]. It should be noted that once an application engages in explicit suspending and activating, care must be taken to avoid deadlocks; retiring, on the other hand, should be harmless.

Note that in the current Ara release, which implements most host accesses by direct operating system calls, agents performing I/O operations retain exclusive control without intervening scheduling actions until the operation is completed. Moreover, the time spent in the operation is not debited to the agent's share of execution time. Therefore, agents should be careful in performing host access operations of unknown duration. As mentioned briefly before, the common blocking I/O operation of reading from the standard input stream does not exhibit this problem, being implemented without retaining control.

3.4 Timing

The Ara core provides a simple synchronous real time service. The current time on the system clock may be inquired, and an agent may suspend its execution for a certain amount of time, which is called *sleeping*. There is no facility as yet for asynchronous timed invocations. Here are the necessary stubs:

```
ara_now ?-nsec?
long Ara_Now(long* nsecPtr);

ara_sleep <sleeping time in ms>
int Ara_Sleep(int sleepingTimeMs);
```

3.5 Service Points

3.5.1 Announcing and Meeting

A service point is created when an agent *announces* it by assigning it a symbolic name. The announcing agent thereby assumes the role of the *server* agent at this service point, and the service point becomes visible under this name for meetings at the local place[3]. An attempt to announce a service point under a name currently in use at the same place will be refused as an error. On success, the newly created service point is returned in the form of its id:

```
ara_announce <name>
Ara_Id Ara_ServicePointAnnounce(char* name);
```

An agent may *meet* a service point at the local place by specifying its name, thereby becoming a *client* to it. If the desired service point has not been announced yet, the meet operation blocks by default until that happens, but the agent may also use a non-blocking variant which returns an error code in this case. On success, the (id of the) service point is returned:

1. Compiled agents are an exception to this (see section 3.9).
2. The current implementation schedules ready processes according to a one-level round-robin policy. Scheduling priorities are expected for a future release.
3. When the directory service becomes available with Ara, an announcement will optionally make the service point globally visible through this service.

```
ara_meet ?-dontwait? <name>
Ara_Id Ara_ServicePointMeet(char* name, int wait);
```

An agent can play the role of a client or server at various service points at the same time, meeting or announcing several of them. A service point can have any number of clients, but only one server, which assumes responsibility for the service point.

3.5.2 Submitting Requests

To make use of a service point, a client *submits requests* to it, to be replied by the server. The submit operation will return the reply, thus realizing a form of synchronous interaction, implemented by blocking the client until the reply. The syntax and semantics of the requests and replies are up to the client and server: From point of view of the service point, they are simply random arrays of bytes. Requests can be submitted using these stubs:

```
<client’s service point> <request>
int Ara_ServicePointSubmit(Ara_Id servicePoint,
                           char* request, size_t requestLength,
                           char** replyPtr, size_t* replyLengthPtr);
```

The syntax of the Tcl stub may seem somewhat peculiar; in fact, this command syntax of naming the object first, followed by the function to be performed on the object, is called “object oriented” in Tcl, and is recommended for access to complex objects. Anyway, the stub returns the reply to the request as its result, provided the server did not reject it (see below), in which case an error is thrown. Note that requests and replies from Tcl agents are necessarily character strings instead of byte arrays, since Tcl cannot represent binary data. Additionally, due to the object oriented syntax, a few “request” strings may have actually predefined meaning in Tcl; currently, this is only the `leave` request explained below in 3.5.4.

The C stub `Ara_ServicePointSubmit()` handles general binary requests and replies, as does the internal service point implementation[1]. The reply is returned in the final two parameters, which may be preset to indicate a memory area intended to receive the reply; this area will be used if its size is sufficient, while a fresh one will be allocated (and returned in these parameters) otherwise. This feature might be used as follows:

```
char replyBuffer[1024];
char* reply = replyBuffer;
size_t replyLength = sizeof replyBuffer;

Ara_ServicePointSubmit(servicePoint, "foo", strlen("foo")+1,
                       &reply, &replyLength);
/* Process data in reply[] */
if (reply != replyBuffer) {
   Ara_Free( reply);
}
```

1. Note that clients should not submit binary data to servers which can handle character strings only and vice versa — the data would be truncated at the first zero byte otherwise.

3.5.3 Fetching and Replying to Requests

The requests submitted to a service point are queued there, until the server *fetches* them. Fetching a request usually blocks until there is at least one to fetch[1], but there is also an option to return an empty request in that case:

```
<server’s service point> fetch ?-dontwait?
int Ara_ServicePointFetch(Ara_Id servicePoint, int wait,
                          Ara_FetchedRequest* fetchedRequestPtr);
```

Again, the `fetch` stub uses the object-oriented syntax. In addition to a reduction of typing, this has the advantage of introducing command name spaces per object type: Function specifiers like `fetch` in this example can be reused for other types than service points without name clashes.

A fetched request as returned by the stubs is implemented as a struct object of type `Ara_FetchedRequest` with three components: The request data area, the name[2] of the requesting agent, and an identifying token to be used for replying. In Tcl, these components are represented as a three-element list. The server may use the requestor’s name as it sees fit, e.g. to decide how much to reply, or whether to reply at all. The following stubs are used to *reply* to a service request:

```
<server’s service point> reply <request token> <reply data string>
int Ara_ServicePointReply(Ara_Id servicePoint,
                          char* data, size_t length,
                          Ara_ServiceRequestToken token);
```

If a server decides that it would rather not reply to a specific request, it can *reject* it.

```
<server’s service point id> reject <request token>
int Ara_ServicePointReject(Ara_Id servicePoint,
                           Ara_ServiceRequestToken token);
```

3.5.4 Renouncing, Leaving and Closing

A client may end a meeting at a service point by *leaving* it; a server may cancel its announcement by *renouncing* the service point.

```
<client’s service point> leave
int Ara_ServicePointLeave(Ara_Id servicePoint);

<server’s service point> renounce
int Ara_ServicePointRenounce(Ara_Id servicePoint);
```

Renouncing a service point is the only method to delete it completely from the system. If the service point has submitted requests pending, they are implicitly rejected. When an agent terminates while still being a client or server to some service point(s), it implicitly leaves respectively renounces all these.

1. There is also a facility to fetch *all* requests currently pending at a service point, which is not shown here to avoid distraction.
2. At the time of this writing, agent names consist solely of the agent’s id, which makes them unique, but conveys no further information beyond identity. Names will be extended in a future release to include structured information about an agent, such as the identity of its principal, a symbolic description, its time of creation etc.

There is one final pair of service point operations: Opening and closing. A server may *close* a service point for new requests, with the effect that any request submitted thereafter will immediately throw an error indicating that the service point is closed. Any attempt of new clients to meet the service point will be treated as though the service point would not have been announced yet (i.e. these clients are usually blocked). Requests which had been pending or fetched, but not yet replied to, however, remain untouched and may still be fetched respectively replied to as usual. Closing a service point is intended to indicate temporary overload; later, the server may *open* it again, returning to normal operation. Any service point is initially open after announcing it.

```
<server’s service point> close
<server’s service point> open

int Ara_ServicePointClose(Ara_Id servicePoint);
int Ara_ServicePointOpen(Ara_Id servicePoint);
```

3.5.5 A Simple Example

This example shows the most common usage pattern of service points, featuring a client and a server agent at a service point. The server agent offers a service point named `Inventory-Service`, and expects requests for the currently available supply (as a number) of certain items of interest. The server serves such requests until requested for a pseudo-item named `finished`. The client agent in the example meets this service point, requests the supply of items named `small-objects`, and leaves again. Here is the example Tcl code for the server and the client:

```
set supply(small-objects)  20
set supply(medium-objects)  8
set supply(large-objects)   3

set sp [ara_announce Inventory-Service]
puts “This is the Inventory server, waiting for requests...”
set request [$sp fetch]; # Request format: {data token client-name}
set item [lindex $request 0]
while {[string compare $item finished]} {
   $sp reply [lindex $request 1] $supply($item)
   set request [$sp fetch]
   set item [lindex $request 0]
}
$sp renounce
puts “Inventory service closed down.”
```

Tcl Source Code of the Inventory Server

```
set sp [ara_meet Inventory-Service]
if {$sp == ““} {
   puts “Cannot meet Inventory Service - exiting.”
   exit
}
set result [$sp medium-objects]
puts “This is the client, found $result medium objects in supply.”
$sp leave
```

Tcl Source Code of the Inventory Client

Running the server and the client as agents at the same place will produce the following output:

```
This is the Inventory server, waiting for requests...
This is the client, found 8 medium objects in supply.
```

The reader may guess what would happen if the client executed `$sp finished` before leaving. To illustrate the correspondence between the Tcl and C interfaces, here is the same server coded in C:

```
static int GetSupply(char* item)
{   /* Return the number corresponding to item */  }

...
Ara_FetchedRequest request;
char replyString[20];
Ara_Id sp = Ara_ServicePointAnnounce("Inventory-Service");
puts("This is the Inventory service, waiting for requests...");
Ara_ServicePointFetch(sp, 0, &request);
while (strcmp(request.data, "finished")) {
   sprintf(replyString, "%d", GetSupply(request.data));
   Ara_ServicePointReply(sp, replyString, strlen(replyString)+1,
                         request.token);
   Ara_ServicePointFetch( sp, 0, &request);
}
Ara_ServicePointRenounce( sp);
puts("Inventory service closed down.");
```

C Source Code of the Inventory Server

3.6 Mobility

The go operation for migration has been introduced before as the means for an agent to move between places. While the basic act of moving is as simple as it seemed, some technical issues still have to be explained in order to utilize all its functions. The full interfaces read as follows:

```
ara_go ?-la <local allowance at destination>? \
       ?-ga <global allowance at destination>? \
       <destination place name> ?<agent>?

int Ara_Go( Ara_Id agent, Ara_PlaceName destination,
            Ara_Allowance local, Ara_Allowance global);
```

An agent is always staying at a place, and the migration operation will make it move to another place as named[1] in the destination argument. Agent mobility can also be exercised on a single machine by making agents move between places located on the same machine. Remember, however, that the current Ara implementation provides only one place per system; as a temporary fix, this can be circumvented by starting several separate Ara systems on one machine[2], each system with its own place. Agents can then move between the places of these systems within the same machine.

When an agent goes to another place, it must take along an allowance for its expenses there. If the agent has a private allowance, i.e. it does not share it with a group, it takes this along completely by default. If, however, the agent shares its allowance with a group, it should indicate on moving how much of the group allowance it wishes to take along. This is the purpose of the global allowance argument of the migration operation; it defaults to the group allowance divided by the number of group members. Note that this default results in the complete allowance in the common case of a singleton group, which is what is usually desired. The agent leaves its group on migration, and the indicated allowance is deducted from that of the group to become the agent's private one.

Remember that agents can also have local allowances valid for their current place only. Therefore, besides taking along a global allowance as described, the agent may also specify a local allowance on migration, to be valid at the destination place (default is the full global allowance). At the time of this writing, the local allowance specified by an arriving agent is always accepted by the receiving place (since there is currently only one, default place per system, and its policy is to accept all agents and allowances).

There are two restrictions to the migration operation. First, as mentioned before, the external state of an agent, i.e. its relations to other system objects and resources like service points, files, or windows are not migrated along with the agent, since those objects depend on the local machine. In particular, a migrating agent loses its relation to its group. Second, there is a somewhat obscure restriction on migration of agents programmed in Tcl: They are not allowed to migrate from within Tcl's traces and asynchronous `proc`'s. This is due to technical reasons and might be abolished in the future, but should not pose a real problem anyway.

1. The API for place names will be explained in the subsequent section 3.7.
2. Note that the various Ara systems on the same machine have to be started each with its own environment settings to prevent collisions. This is explained in the installation guide in the Ara software distribution.

Finally, the agent argument to the migration operation remains to be explained. This can be used to specify *which* agent is to go. Of course this defaults to the running agent, but the migration mechanism in Ara has been implemented so general as to also allow migrating another agent, asynchronously to its execution, provided the active agent possesses the access right over the migrated one. This feature may be used e.g as a building block for emergency measures or load distribution. Note, however, that the concerned agents must be prepared to such migrations in some form.

3.7 Place Names

Place names are not system objects, but application data, in order to allow agents taking them along on migration. However, their exact format is hidden from the application by a small interface, which is still expected to grow as place names will become more structured in future system releases.

In C, place names are represented as objects of type `Ara_PlaceName.`Objects of this type will usually be obtained from the directory service once it is available, but they also have a character string representation, which can be used by applications to construct place names[1]. Conversion between `Ara_PlaceName`'s and their string representation are performed by these functions:

```
Ara_PlaceName Ara_PlaceNameCreate(char* initValue);
void Ara_PlaceNamePrint(Ara_PlaceName placeName, char* outputString);
```

In Tcl, being string based, the string representations must be used directly, and conversions are performed internally. Note that in C `Ara_PlaceName`'s which have been constructed should be deleted after use by `Ara_PlaceNameDelete` (see below) to avoid a memory leak.

An agent can find the name of its current, i.e. local, place through the following stubs:

```
ara_here
Ara_PlaceName Ara_Here();
```

Place names can be compared for identity, returning a boolean result, by

```
ara_placename equal <place name> <place name>
int Ara_PlaceNameEqual(Ara_PlaceName, Ara_PlaceName);
```

The C interface also provides functions to delete, copy, and assign place names; again, this makes no sense for Tcl.

The stubs look as follows:

```
void Ara_PlaceNameDelete(Ara_PlaceName);
Ara_PlaceName Ara_PlaceNameCopy(Ara_PlaceName source);
void Ara_PlaceNameAssign(Ara_PlaceName* dest, Ara_PlaceName source);
```

1. In the current Ara implementation, providing only one place per system and basically only TCP as a communication protocol, the string representation of place names is *machine*[:*port*], *machine* being the DNS name or IP address of the destination machine, and *port* being the TCP port of the target Ara system's default place. Annex B gives a brief outlook on the expected future format of place names.

3.8 Checkpointing

An agent may create a checkpoint of its current (internal) state using one of the following stubs:

```
ara_checkpoint ?-exit ?<exitVal>?? ?-ga <global allow. after restore>?
?-la <local...>?
int Ara_Checkpoint(int* restored, int exit, int exitValue,
                   Ara_Allowance global, Ara_Allowance local);
```

Similar to the forking function of section 3.3.1, this function returns in two different contexts: After creating the checkpoint, control returns like any normal call. If the agent is ever restored from the that checkpoint later, it will resume control returning from this function as well, however indicating the fact of restoration by a different result: The Tcl stub returns `1` after a restoration, and `0` on normal return; the C stub returns the same in the `restored` output variable, and shows standard error behavior for the rest. The checkpoint is stored as a disk file in a dedicated directory under a name derived from the id of the checkpointed agent. This file is kept until the checkpoint is discarded. The allowance parameters define the allowance for the agent after a potential later restoration; the defaults are the same as with migration. Note that this allowance is stored with the checkpoint, and deducted from the allowance of the checkpointing agent. Finally, it is a common case that an agent plans to exit immediately after checkpointing itself; the "exit" flag provides just this, optionally leaving a termination value as usual. This flag is not only a convenience, but the only way for an agent to store its *complete* allowance in the checkpoint, since any execution continued after checkpoint creation, no matter how short, would require its own share of the available allowance.

Restoration works by specifying the id of the agent to be restored; the corresponding file is then used to recreate the checkpointed agent, and the checkpoint file is deleted. The restored agent is placed in its own isolated group by default, but it may also be received into the restoring agent's group. The agent to be restored defaults to the current one. Note that restoring the current agent implies the "termination" of its "present" incarnation, since that is replaced with the incarnation from the time of checkpointing. The stubs read as follows:

```
ara_restore ?-mygroup? ?<agent>?
int Ara_Restore(Ara_Id agent, int mygroup);
```

When restoring another agent, the Tcl stub returns nothing, while the C stub shows standard error behavior. Obviously, in the case of restoring the current agent, restoration does not return. Remember that restored agents cannot expect to find their external state unchanged (this was explained in section 1.2). When restoring another agent, the application should either make sure that no incarnation of this agent has survived anywhere, or should be prepared to resolve such collisions.

When checkpoints are no longer needed for potential restorations, they can be explicitly discarded as follows, deleting the corresponding file:

```
ara_discard ?<agent>?
int Ara_Discard(Ara_Id agent);
```

When discarding a checkpoint, the allowance stored with it is recycled by crediting it to the discarding agent's allowance. Discarding returns a boolean result indicating whether such a checkpoint had indeed existed.

Both restoring and discarding another agent's checkpoint do not perform any access right checking in the current implementation.

The most common use of checkpointing is to create a fall-back line whence to recover after a fatal failure. An agent might use this before going to a “dangerous” place, registering at some “wake-up service” to restore it unless the registration is cancelled within a certain time:

```
if {![ara_checkpoint]} {
   $wake-up register [ara_me] 100 ;# "Wake me up after 100 time units"
   set back [ara_here] ;# The name of the local place
   ara_go $away
   ... ;# Do some dangerous work at $away
   ara_go $back
   $wake-up cancel [ara_me]
} else {
   # Oops, did not return in time from $away - start recovery,
   # e.g. find out if the agent has crashed.
   ...
}
```

Another common usage is to guard “dangerous” procedures with checkpoints, under the assumption that some other agent will restore the failed one:

```
proc p_guarded {...} {
   if {[ara_checkpoint]} {
      return "Failed in proc p!"
   } else {
      p ;# Do the real work
      ara_discard
} }
```

3.9 Compiled Agents

Compiled agents provide a native speed alternative to the standard interpreted Ara agents for cases where certain security and portability requirements are not strictly necessary. Most prominently, this applies to agents which are resident and trusted at a site. This is the way to go to extend an Ara system with specific resident functionality, e.g. a new service, to be offered to visiting agents: The service is coded as a server agent (in C, presumably), compiled to native code, and a compiled agent is created from this. This can be performed on demand at run-time when an agent requests that service[1]. While the core is common to all Ara systems, the set of stationary server agents is what distinguishes individual sites, potentially ranging from hand held devices with only a few server agents to corporate installations with hundreds of them.

Compiled agents are different in two respects, apart from their increased execution speed. They cannot usually migrate, and the security of their execution cannot totally be warranted. With regard to their common usage as described, these restrictions do not pose a problem. Apart from migration and security, they can be treated like any other agent: The complete API as introduced in section 3 applies to compiled agents as well. The lack of migration capability is rooted in the very nature of native code, operating on a specific microprocessor after all, which precludes a direct transfer to a machine with a heterogeneous architecture. Actually, migration of compiled agents *has* been implemented in an indirect way, by exploiting the source code,

1. Compiled agent creation is implemented using dynamic link editing of the running Ara system.

provided it is available[1]. The source code is transparently moved along with the migrating agent and recompiled at each destination. Concerning, however, the security issues raised by compiled agents, some further explanations are appropriate.

Compiled agents are not executed within an interpreter, but have access to the physical processor, memory, and operating system. They might write or jump to arbitrary memory locations and perform unpredictable system calls. It is difficult to enforce (efficient) run-time checks upon them while staying independent of the operating system and hardware[2]. For this reason, Ara does not currently perform any run-time checking of compiled agents. It is, therefore, generally recommended to load compiled agents obtained from trusted sources only. In the case of a remote source, this means a digitally signed transfer[3] from a trusted site. For compiled agents loaded from the local file system, it must be ensured that this is from a location protected against access by untrusted agents.

Besides the risk of undue access, another security concern with compiled agents is caused by their exclusive possession of control once the system enters compiled code. A compiled Ara agent retains control until voluntarily and synchronously releasing it. This implementation was chosen since an asynchronous, forced interruption would leave the agent in a machine-dependent state, considerably complicating its further handling, while gaining nothing for normal, interpreted agents. For this reason, compiled agents must be trusted to return control to the system frequently enough to preserve parallelism among the agents in the same system. This can be done e.g. by retiring at regular intervals or calling a special function for time slice checking. An example for this can be found in the system's internal communication process (see section 5.2).

Finally, the API for compiled agents is slightly different in three syntactical or technical respects from that used by C agents: First, once they are started, compiled agents are trusted as explained above. This implies that the security measure of protecting system objects by an indirection through object ids no longer makes sense. Accordingly, compiled agents usually access system objects directly through memory addresses (i.e. C pointers): Instead of having one `Ara_Id` type for all objects, there are separate types such as `Ara_ServicePoint` which are returned by object creation functions and expected by object access functions. Special values such as `ARA_SERVICEPOINT_NONE` etc. are used in place of `ARA_ID_NONE` to designate null values. Second, within the Ara core, agents are usually called "processes" for historical reasons. This is mirrored in the naming used in the compiled agent API, e.g. the object type for agents is called `Ara_Process`. This difference bears no further significance: Whenever "process" is used as in `Ara_Kill(Ara_Process victim)`, it is safe to simply think of an agent. Third, the core function to create compiled agents has a slightly wider interface than the C stub given in section 3.3.1, allowing to create processes executing code which already exists in memory in compiled form (instead of loading it from a file):

```
Ara_Process Ara_CreateCompProcess( Ara_CompCode code, char* functionName,
                                   byte* data, int dataLength,
                                   Ara_Allowance global, Ara_Allowance local,
                                   int size, int flags);
```

1. The source code is assumed to be contained in a file named *f*.c, where *f* is the name specified at agent creation (see section 3.3.1).
2. There are some rather specific approaches to this, e.g. [LSW95, WLA+93] use a modified compiler.
3. Support for this will be added in a future Ara release.

The `code` parameter may be used to specify the function to be executed by the new process; `functionName` must be `NULL` then. `Ara_CompCode` is a type definition for `int (*)(byte*, int)`.

3.10 Allowances

Allowances have played a role several times before already, as arguments to the agent creation and migration operations, the details of which have been covered in the respective sections (3.3.1 and 3.6, respectively). Accordingly, the purpose of allowances to limit an agent's resource accesses, and their treatment in agent creation and migration should be clear by now, as well as the concept of global and local allowances. What remains to be described is the API for explicit access to allowances. It should be remarked here that allowances, including groups, are a recent concept in Ara, and not complete yet.

Remember that at the time of this writing, allowances are defined for execution time and memory consumption only; other resources like disk space, system objects created (i.e. agents, service points, network connections etc.) or visited places will follow. An agent can *inquire* its own (and others', access rights provided) current local and global allowance at any time; this might be used to choose between actions of differing resource requirements, or to check whether the local place has indeed honored the local allowance desired on entering it. The stubs look as follows:

```
ara_get_allowance ?<agent>?
int Ara_GetAllowance(Ara_Allowance* global, Ara_Allowance* local,
                     Ara_Id agent);
```

`Ara_Allowance` is a C struct containing the individual allowances for execution time and memory as integer members, while the Tcl representation of allowances is a two-element list of these numbers. The `ara_get_allowance` stub returns the global and local allowance as a list of two such allowances.

Not surprisingly, it is not possible to arbitrarily set an allowance, a security measure to prevent tampering by greedy agents. There is, however, an operation to *transfer* some allowance from one agent to another, provided the necessary access right. The transferred "amount" is added to the receiving agent's allowance, and may be deducted from the source agent's allowance[1]. This ensures that allowances can be distributed among agents, yet the total sum of all allowances cannot be increased. Note that the direction of the transfer can be reversed, corresponding to giving and taking. Allowance transfers are mostly intended for cooperating agents (re)distributing resources among each other, and agents running out of resources having additional allowance transferred to them (e.g. from a system service contacting their home site). Other uses of allowances, however, are conceivable as well, e.g. as objects of trade between agents. The offer for a remote service might be bundled together with an allowance sufficient to cover the travel and carry out the access, to make the offering comparable independent of the expenses involved with its use.

When transferring allowances between two agents, it is important to discriminate between transfers with respect to the local, global, or both variants of it. As a general prerequisite, the transferred amount is deducted from the source agent's allowance, so the latter must be sufficient to cover the transfer. This assured, transfers of purely global allowances are performed without further questions. Transfers of local allowances, on the other hand, additionally require that both agents are located at the same place, in order to preserve the total

1. The exact semantics of such a "transfer" depends on the type of resource, e.g. on whether the allowance is qualitative or quantitative.

local allowances issued by this place; further, the amount transferred must stay within the bounds of the recipient's global allowance. Note that a transfer of purely local allowance does not change the global allowances of either source or recipient agent; a simultaneous transfer of both variants may be used for this. Here are the stubs for allowance transfer; the source agent defaults to the current one:

```
ara_transfer_allowance ?-la <local allowance>?  ?-ga <glob. allowance>?\
                  <recipient agent> ?<source agent>?

int Ara_TransferAllowance(Ara_Allowance global, Ara_Allowance local,
                          Ara_Id sourceAgent, Ara_Id recipientAgent);
```

As mentioned before in the allowance discussion of the agent creation and migration operations, agents are implicitly assigned a group at creation time, possibly sharing a common allowance. At the time of this writing, there are no operations to explicitly leave[1] or enter a group. Presently, an agent can find the size, i.e. the number of members of its own (or another agent's, access right provided) group as follows:

```
ara_groupsize ?<agent>?
int Ara_GroupSize(Ara_Id agent);
```

At the time of this writing, there is no facility for agents to catch allowance exhaustion; instead, agents having exhausted their allowance will be terminated by the system. This should not be a severe problem, since an agent can, of course, inquire about its current allowance at any time and direct its actions according to that. In a future release, agents will be able to register arbitrary code to be executed asynchronously when resources run low, to take appropriate measures to avoid being terminated, such as leaving the system or having additional allowance transferred to it.

3.11 Dynamic Memory

Languages with explicit memory management need a facility to dynamically obtain raw memory from the system. C is an example for such a language, while Tcl is not. As memory consumption must be accounted to the consuming agent's allowance, all dynamic memory allocations must be performed through dedicated core functions. The C API provides stubs for the two classical memory management functions, allocation and deletion of a memory block:

```
void* Ara_Alloc(size_t size);
void Ara_Free(void* p);
```

As usual, memory blocks allocated by `Ara_Alloc()` must be freed using `Ara_Free()` in order to reuse that memory. Unfreed memory will be freed implicitly on agent termination. Note that while languages without explicit memory management do not (have to) use these stubs, their memory consumption is accounted nevertheless, since in that case the interpreter uses the concerned core functions implicitly.

3.12 Input and Output

As explained in section 1.4, "Accessing the Host System", currently Ara does not yet provide I/O functions of its own, and allows applications to use their own native I/O facilities instead. For instance, a Tcl agent may read from a file as usual by

1. Remember that a migrating agent implicitly leaves its group (behind).

```
set file [open myfile]
read $file
close $file
```

while a C agent would use the familiar

```
FILE* fp = fopen("myfile", "r");
fread(buffer, size, n, fp);
flose(fp);
```

The implications of this arrangement have also been mentioned in section 1.4. Actually, one I/O Ara function does exist already, namely that for reading a line of characters from the standard input stream (usually a terminal). Its stubs are as follows:

```
gets stdin   # i.e. standard Tcl’s ‘gets’ command serves as the stub
char* Ara_Gets(char* buffer, int bufferLength);
```

4. A Programming Example

Having looked at the Ara programming features in some detail, it will also be helpful to see the complete picture, that is, an actual application in full source code[1]. In the limited space available, this section will present a small demonstration application, featuring a mobile agent to search the World Wide Web for “interesting” documents. The agent will visit sites, examine their data, collect results, and continue its itinerary according to its findings. In terms of section 2.1, this is a typical information research application.

The strategy of the search agent is to move along the hyperlinks found in interesting documents, in the hope that these might lead to other interesting documents, as is often the case given the roughly content-based topology of the web. The search continues until a predetermined allowance of resources has been consumed, or until there are no more interesting links to be examined. As the result of its search, the agent brings a list of the URLs (i.e. web addresses [BMM94]) of all discovered documents back to its home place. To provide access to the documents of a site, a stationary server agent is employed which provides a service point acting as a document retrieval service: When presented with the URL of a document, it will reply with the contents of this document. Conceptually, this server is a functionally restricted proxy agent (in terms of section 1.4) of the local web server demon; however, to keep the example short enough, the presented server agent accesses the documents directly through the file system, rather than through conversation with a web server.

The question remains of how to define what exactly makes a document “interesting” in terms of this search. Incidentally, the agent is to search for documents about mobile agent technology. Many sophisticated conditions are conceivable to delimit the desired content, but rather than digressing into text analysis here, it shall suffice for the sake of illustration that the document contain the string “mobile agent”.

4.1 The Document Server Agent

Being stationary, the server is implemented as a compiled agent for improved performance; the implementation shown here is written in C. The Ara software distribution also contains an equivalent implementation in Tcl for the sake of comparison. Basically, the server announces a service point named “Document Retrieval

1. The source code is also contained in the Ara software distribution.

Service”, and then fetches requests for documents in a loop until termination. Each request is expected to have the format of a path name of a document file in the server’s local file system, and the contents of the corresponding file are read and returned to the client as a reply. This results in the following overall structure:

```
serviceP = Ara_ServicePointAnnounce("Document Retrieval Service");
do {
    Ara_ServicePointFetch( serviceP, ARA_SERVICEPOINT_WAIT, &request));
    normalizedPathName = Parse document file name from request.data;
    length = size of document file;
    replyBuffer = Ara_Alloc(length);
    fread( from file named normalizedPathName, into replyBuffer);
    Ara_ServicePointReply( serviceP, replyBuffer, length, request.token);
    Ara_Free(replyBuffer);
} while not finished;
Ara_ServicePointRenounce( serviceP);
```

Top-level Structure of the Document Server

The full source code is now presented in detail. The server is implemented as a main function `Server_Main`, which serves as the top-level function *f* of the underlying compiled agent upon agent creation (see section 3.3.1, “Agent Creation”). As is common in C, the source code begins with some inclusions of needed interfaces; note in particular the inclusion of `ara.h`, the Ara core API for compiled agents. The main function commences with the definition of three macros required by Ara for handling the function’s state (see section 5.4 and the documentation in the Ara software distribution), which are not of further interest here, as they are defined to be empty.

```
#include <stdio.h>
#include <string.h>
#include <ara.h>

int Server_Main (data, dataLength)
  byte* data;
  int dataLength;
{
# define Ara_ParcelLocalState(process)
# define Ara_UnparcelLocalState(process)
# define Ara_CleanupLocalState
```

After defining the needed local variables, another Ara state handling macro follows which marks the beginning of the function’s code. The server initializes itself by determining the path prefix of the documents it wishes to provide for client access. All such documents are assumed to reside in the file system subtree rooted at this prefix, and any document file name requested by clients will be interpreted as relative to the prefix. Incidentally, the prefix in this example is chosen as the directory the server was started from, to allow for easy experimentation with different prefixes. The initialization is completed by announcing the service point for delivering the documents, whereupon the server enters the loop to wait for document requests.

```
      # define EMERGENCY_STRING "Out of memory"
        char prefix[1024];
        char* normalizedPathName;
        Ara_ServicePoint servicePoint;
        Ara_FetchedRequest request;
        char requestBuffer[1024];
        char* replyBuffer;
        FILE* stream;
        int length = 0;

        Ara_DeclAndCheckSwitch1;

        getcwd(prefix,sizeof prefix);
        strcat(prefix,”/”);
        server = Ara_ServicePointAnnounce("Document Retrieval Service");

        do {
```

Each pass through the loop begins with fetching a request from the service point; the server uses the blocking variant of fetching in order to wait until at least one request is available. The macro enclosing the core call for fetching is part of the state handling again. Requests are fetched using an `Ara_FetchedRequest` struct object, which also refers to a memory area to receive the request data. This area may optionally be initialized with one prepared by the server for this purpose, as is shown here with the `requestBuffer`. In any case, the `Ara_FetchedRequest` will contain the request data on return from fetching, and the prepared memory, if specified, is used, provided it was large enough; otherwise, a fresh area will have been allocated. After fetching, a memory buffer to hold the physical path name of the requested document is allocated.

```
            request.data = requestBuffer;
            request.length = sizeof requestBuffer;
            Ara_SwitchCall1(Ara_ServicePointFetch(servicePoint,
                                            ARA_SERVICEPOINT_WAIT, &request));
           normalizedPathName = Ara_Alloc(request.length+strlen(prefix)+1);
```

Now the path name buffer is filled with the requested file name, appended to the subtree prefix described above. If the request data space had indeed been freshly allocated during fetching, it may be deleted now. The path name is subjected to some parsing, and it is checked that it still bears the prefix after that; this is a security measure explained with the parsing function below. Now the requested document file can be opened, and it is read in whole into a freshly allocated buffer. A trailing zero byte is appended to make the data easily handled as a character string in the case of a text file.

```
    if (normalizedPathName != NULL) {
        sprintf(normalizedPathName,"%s%s", prefix, request.data);
        if (request.data != requestBuffer) {
            Ara_Free(request.data);
        }
        if (    ParseFileName(normalizedPathName) != NULL
            && !strncmp(normalizedPathName, prefix, strlen(prefix))) {
            if ( (stream = fopen(normalizedPathName, "r")) != NULL) {
                fseek(stream,0L,2);
                length = ftell(stream);
                fseek(stream,0L,0);
                if (length < strlen(normalizedPathName)+80) {
                    replyBuffer = Ara_Alloc(strlen(normalizedPathName)+81);
                } else {
                    replyBuffer = Ara_Alloc(length+1);
                }
                if( replyBuffer == NULL) {
                    replyBuffer = EMERGENCY_STRING;
                } else if (!fread(replyBuffer,1,length,stream)) {
                        sprintf(replyBuffer,"Error while reading file %s",
                                normalizedPathName);
                    }
                replyBuffer[length] = '\0';
                length += 1;
                fclose(stream);
```

If all went well, the reply data is now assembled in the `replyBuffer`. If, on the other hand, errors occurred in accessing the file, an error message is stored in the `replyBuffer` instead as follows:

```
            } else {
              replyBuffer = Ara_Alloc(strlen(normalizedPathName)+80);
              if( replyBuffer == NULL) {
                  replyBuffer = EMERGENCY_STRING;
              } else {
                  sprintf( replyBuffer,
                          "Illegal operation: File %s does not exist",
                          normalizedPathName);
              }
```

```
                length = strlen(replyBuffer);
              }
          } else {
              replyBuffer = Ara_Alloc(strlen(normalizedPathName)+80);
              if( replyBuffer == NULL) {
               replyBuffer = EMERGENCY_STRING;
              } else {
                sprintf(replyBuffer, "Illegal operation: Permission denied");
              }
              length = strlen(replyBuffer);
          }
      } else {
          replyBuffer = EMERGENCY_STRING;
          length = strlen(replyBuffer);
      }
```

The processing of a request is completed by passing the retrieved data as a reply to the service point, and freeing all buffers.

```
      Ara_ServicePointReply(server, replyBuffer, length, request.token);
      Ara_Free(normalizedPathName;
      if( !strcmp(replyBuffer, EMERGENCY_STRING))  {
         Ara_Free(replyBuffer);
      }
```

The server will now begin the next pass through the loop, until it terminates. In this example, the server agent can be terminated by submitting a special request “exit” to the service point. This is only for convenient experimentation; a real server would presumably not let itself be terminated by a client, but would rather decide this on its own based on additional conditions, or be terminated by some superior controlling agent. Upon termination, the service point is renounced, and the main function ends with some macro undefinitions corresponding to those at the beginning.

```
     } while(strcmp(request.data, "exit"));

     Ara_ServicePointRenounce(servicePoint);
     return 0;

   # undef Ara_DeleteLocalState
   # undef Ara_UnparcelLocalState
   # undef Ara_ParcelLocalState
   }
```

The last item to report of the server is the auxiliary function used for file name parsing. This function removes any pseudo-directories named `..` from a path name by “performing” their effect on the rest of the path. This is a security measure on the server’s side, since only files in the file system subtree designated for client access are to be exposed. A malicious client could try to subvert this restriction by inserting `..` into the file name, effectively reaching out of the designated subtree. This is the source code of the function:

```
static char* ParseFileName( pathName)
  char* pathName;
{
  char* help;
  int i, j;
  while (help = strstr( pathName, "//")) != NULL) {
       j = help - pathName-1;
       while( j >= 0 && pathName[j] != '/') {
            j --;
       }
       if( j == -1) {  Ara_Free( pathName);
           return NULL;
       }
       for(i=help-pathName+3;  i<=strlen( pathName); i ++, j++) {
           pathName[j] = pathName[i];
       }
  }
  return pathName;
}
```

4.2 The Search Agent

The search agent is implemented in Tcl, which is a language well adapted to text processing, e.g. through regular expression handling. Roughly, the agent works through a list of URL references to documents to be searched, moving to each document's site, and retrieving its content from the local document retrieval service (see former section). Each retrieved document, provided it turns out to be interesting as described above, is searched then for hyperlinks referencing further documents, and all such references are added to the work list for later inspection. Additionally, all references to interesting documents are collected, and this collection is printed as the search result upon returning to the home site. At regular intervals, the residual allowance is checked if it is still sufficient to continue the search. The overall structure of the agent thus looks like this:

```
Initialize and ask the user for an initial list of urlsToVisit to start the search at
SearchWeb:
  while (there are urlsToVisit and AllowanceSufficient) {
    ara_go site of next document
    set servicePoint [ara_meet "Document Retrieval Service"]
    while (there are urlsToVisit at this site and AllowanceSufficient) {
      set document [$servicePoint path of next URL]
      if ($document contains "mobile agent") {
        Collect this URL
        SearchDocument $document, extracting URLs and adding them to urlsToVisit
      }
    }
    $servicePoint leave
  }
ara_go $home
PrintResult
```

Top-level Structure of the Document Search Agent

The full source code of the agent's main program begins with the initialization of the global variables, the purpose of which is explained in the source code comments. Most important is `urlsToVisit`, the general list of references still to be searched. At the beginning of the program, this list is filled with some document URLs typed in by the user; simple site names may also be typed here, which are interpreted as references to the top-level document at that site.

```
set filesRead ""   ;# contains all files already read
set document ""   ;# the content of the currentFile
set urlsToVisit "";# list of files (site/path) still to read
set currentSite "";# name/adress of the Site currently worked at
set currentFile "";# file (incl. path) currently examined
set filesHere ""   ;# files found on this site which are still to be
                    # examined; files contain the path without site
set home [ara_here];# the site to return to after work (the start site)
set sufficientTime 0 ;# flag indicating after returning home whether
                    # execution time was sufficient
set sufficientMemory 0 ;# the same for memory

puts "Please enter the machine(s) or URL(s) to start the search at:"
while {[gets stdin next]} {
  if {[regexp -nocase {http://} $next]} {
    # Assume that $next is a URL to start the search at.
    regexp {http://(.*)} $next dummy next
  } else {
    # Assume that $next is a machine to be searched; start at its index
    # document.
    set next "$next/index.html”
  }
  lappend urlsToVisit $next
```

After initialization, the web search is started. The agent is instructed to search for documents matching the (most simple) regular expression “mobile agent”, and to finish the search early if its residual allowance of CPU time ever falls below 5 seconds, or if its memory allowance falls below 100 KB. After returning home, the results are printed, and the user is notified if the agent had to finish early, as opposed to completely covering all reachable documents.

```
SearchWeb {5 100k} "mobile agent"
ara_go $home
PrintResult
if { !$sufficientMemory} {
  puts "Gone out of memory allowance"
  puts "Total allowance left: [ara_get_allowance]"
}
if { !$sufficientTime} {
  puts "Gone out of time allowance”
  puts "Total allowance left: [ara_get_allowance]"
}
exit
```

Since the agent will return when its allowance runs low, it should be started with sufficient allowance in the first place, the exact amount of which will of course depend on the size of the web to be searched.

The heart of the search agent is the `SearchWeb` procedure, basically consisting of two nested loops. The outer loop processes the general list of URLs still to be examined, usually located at various remote sites. Once moved to a specific site of these, the agent meets the local document retrieval service, and retrieves and searches all known documents located at this site. Note also that before each loop entry, the agent checks its residual local allowance to leave in time before this runs low. In the example here, the agent returns home in such a case.

```
proc SearchWeb {minAllowanceNeeded} {
  global document
  global filesHere
  global filesRead
  global currentSite
  global currentFile
  global urlsToVisit
  global sufficientTime
  global sufficientMemory

  while {   ([llength $urlsToVisit] > 0)
        && [AllowanceSufficient $minAllowanceNeeded]} {
    set next [lindex $urlsToVisit 0]
    set urlsToVisit [lreplace $urlsToVisit 0 0]
    # next: the next site/path/file to be visited

    regexp {([^/]*)(.*)} $next dummy currentSite currentFile
    # the file (incl. path) and the site are extracted
    lappend filesRead [ParseFileName "$currentSite$currentFile”]
    # mark the file as read

    if {![catch {ara_go $currentSite}]} {
      # successfully changed to the current site

      puts "Search agent [ara_me] arrived at $currentSite”

      lappend filesHere $currentFile

      set servicePoint [ara_meet "Document Retrieval Service”]

      while {   [llength $filesHere]
            && [AllowanceSufficient $minAllowanceNeeded]} {
        # there is still a file at this site not examined, and there are
        # enough resources left to examine at least one file

        set currentFile [lindex $filesHere 0]
        set filesHere [lreplace $filesHere 0 0]
```

At this point, a specific document, identified by its path name stored in `currentFile`, has been chosen for examination. The search agent retrieves the content of this document by requesting it from the service point of the retrieval service. If the document could be retrieved, it will be examined and searched further:

```
        set document [$servicePoint $currentFile]
        lappend filesRead "$currentSite$currentFile”
        if { $document != "Illegal operation: File does not exist” &&
             $document != "Illegal operation: Permission denied” &&
             $document != "Out of memory”} {
           SearchDocument $searchExpr
        } else {
           puts "\”$currentFile\”: $document”
        }
        unset document
```

The variable holding the document content is explicitly deleted (`unset`) after processing the document. This is a typical optimization measure prior to a migration, in order to avoid taking along data which is not really needed any more. If this deletion were omitted, the variable would be implicitly deleted by later overwriting it with the content of the next retrieved document; this, however, might not happen until after the next migration.

In order to reduce multiple visits to the `currentSite`, the agent examines all candidate documents located there before moving on to another site, no matter in what sequence they had been added to `urlsToVisit`. Note, however, that multiple visits cannot be completely avoided in the case where the agent learns of the existence of a candidate document only after having visited that document’s site. The references to local files are thus transferred from `urlsToVisit` to `filesHere`:

```
        while {[set further \
                    [lsearch $urlsToVisit "$currentSite/*”]] >= 0} {
          regexp "$currentSite\(.*)” [lindex $urlsToVisit $further] \
                 dummy file
          set urlsToVisit [lreplace $urlsToVisit $further $further]
          lappend filesHere $file
        }
      } ;# filesHere
```

When there are no more documents known to be searched at the local site, the agent leaves for the next site, and continues its loop until all references have been processed:

```
      $servicePoint leave
      puts "Search agent [ara_me] leaves $currentSite”
    } else {
      puts "Search agent cannot go to $currentSite”
    } ;# if go
  } ;# while urlsToVisit
}
```

Processing an individual document is performed by the `SearchDocument` procedure. This extracts all hyperlink references from the document, and each reference, local or remote, is transformed into a normal form: The `http://` protocol specification is removed[1], as well as any interspersed `//`, `/../` or `/./`. Local references are transformed into absolute path names, while global ones have an additional site name added at the front. Note that local references are not added to the general `urlsToVisit` list, but collected in the local `filesHere` list, as expected by the `SearchWeb` procedure in its effort to search local files before remote ones.

```
proc SearchDocument{searchExpr} {
  global document
  global filesHere
  global filesRead
  global currentSite
  global currentFile
  global urlsToVisit

  if {![regexp $searchExpr $document]} {
    return
  }
  if {![GetTitle]} {
    puts "\"$currentFile\": Illegal format: No HTML-<TITLE>"
    return
  }

  if {![regexp {(/+.*/)([^/]*)} $currentFile dummy path]} {
    set path /
  }
  # Initialises path as the path of the currentFile

  while {[regexp -nocase \
        "<A( |\t|\n)+HREF( |\t|\n)*=( |\t|\n)*\"(\[^<\]*)\"( |\t|\n)*>" \
           $document dummy dummy dummy dummy reference]} {
    # while a new reference is found in the currentFile

    regsub -nocase \
        "<A( |\t|\n)+HREF( |\t|\n)*=( |\t|\n)*\"(\[^<\]*)\"( |\t|\n)*>" \
            $document "" document
    # deleting the reference in document

    set url [regexp -nocase "http://$currentSite" $reference help]
    if {$url || ![regexp -nocase {http://} $reference help]} {
      # it is a reference to this site (search for http-references only)

      if {$url} {
        # extracting the reference (including the complete path)
        regexp -nocase "http://$currentSite\(.*\)" $reference help \
               reference
```

1. For the sake of simplicity, the client searches only for HTML documents accessible by the HTTP protocol (i.e. having URLs beginning with `http://`).

```
        }

        if {![regexp {^/} $reference dummy] }  {
          set reference $path/$reference
        }
        # the path is added to the new reference if it is a relative one

        set reference [ParseFileName $reference]

        if {[lsearch $filesRead \
                     [ParseFileName "$currentSite/$reference”]] < 0 &&
            [lsearch $filesHere $reference] <0} {
          lappend filesHere $reference
        }
        # if the new file has not been read before it is appended to the
        # list filesHere

      } else {
        # it is a reference to a remote site

        regexp {[^:]*://(.*)} $reference dummy reference
        set reference [ParseFileName $reference]
        # format of reference is now site/path/file
        if {[lsearch $filesRead $reference] < 0 &&
            [lsearch $urlsToVisit $reference] < 0} {
          lappend urlsToVisit $reference
        }
      }
    } ;# while
  }
```

The search agent is completed by four auxiliary procedures for printing out the table of collected document references, parsing a file name, collecting the title of an HTML document, and checking whether the residual allowance of the agent is still sufficient. Note that file name parsing is performed here not for reasons of security, as in the server agent, but to arrive at a unique representation for each URL, in order to prevent multiple searches of one document referenced by several equivalent, but not identical URLs. The source code of the procedures follows here:

```
proc PrintResult {} {
  global table
  foreach title [array names table] {
    puts "Title: $title”
    puts "Files: http://$table($title)”
    puts  ""
  }
}

proc ParseFileName {file} {
  while {[expr [regsub -all {/\./} $file / file] \
             + [regsub -all {//} $file / file] \
                + [regsub -all {([^/]+)/\.\./} $file / file]] } {}
  return $file
}

proc GetTitle {} {
  global document
  global table
  global currentFile
  global currentSite
  if {[regexp -nocase {<TITLE>([^<]*)</TITLE>} $document dummy title]} {
```

Enter the reference leading to this document into the table of document references. Note that this table associates document titles with a *list* of references, since there might be several copies of the same document at different locations, and the table is to list all of them. In any case, two documents bearing the same title are assumed to be identical.

```
    lappend table($title) [ParseFileName $currentSite/$currentFile]
    return 1
  }
  return 0;
}

proc AllowanceSufficient {minAllowanceNeeded} {
  global sufficientTime
  global sufficientMemory

  set sufficientTime [expr [expr [lindex \
                          [lindex [ara_get_allowance] 1] 0] == -1] || \
                     [expr [lindex [lindex [ara_get_allowance] 1] 0] > \
                                [lindex $minAllowanceNeeded 0]]]

  set sufficientMemory [expr [expr [lindex \
                          [lindex [ara_get_allowance] 1] 1] == -1] || \
                     [expr [lindex [lindex [ara_get_allowance] 1] 1] > \
                                [lindex $minAllowanceNeeded 1]]]
  return [expr $sufficientMemory && $sufficientTime]
}
```

5. Ara System Architecture and Implementation

This section explains selected internal concepts of the Ara system which are useful to gain deeper insights into its rationale and capabilities, and also to combine it with new software components. The material presented here is an offer for the technically interested reader, but not required for using the system, or understanding the previous sections.

5.1 Processes and Internal Architecture

Section 1 already introduced some basic concepts of the Ara architecture, namely agents, interpreters, and the core. Agents run within interpreters for their respective programming language, controlled and served by the common system core. It is a fundamental principle of Ara to execute agents as autonomous, concurrent processes. This supports their independent and possibly asynchronous execution, provides flexible control, and facilitates mutual protection. As the system needs to control the agents in a rather fine-grained manner, it does not rely on the comparatively heavy-weight process abstractions and relatively coarse control usually provided by the host operating system. Furthermore, common process implementations, be they light or heavy-weight, tend to be particularly platform-specific. Instead, the Ara core provides its own process abstraction as the basis for agent implementation. Within such an *Ara process*, there is usually some interpreter running, processing the program of a mobile agent; from point of view of the core, however, all processes are treated uniformly.

The core governs the agent processes, and mediates their interaction and host system access. Basic functions, such as migration, are provided to the agents by the core, while higher-level services are offered by server agents. While mobile agents are interpreted for reasons of portability and security, stationary agents may also be compiled, as discussed in section 3.9. Examples occur among the *system processes*: In addition to the agents visible at the application level, the Ara system also employs processes for certain internal purposes, in order to modularize the architecture. If such system processes are stationary, they are compiled. The process structure thus supports architecture modularity, without harming performance in any significant way[1]. An example for this is the communication process, which handles the network interface of the host system. System processes have special permissions and may directly access the host operating system, bypassing the core. As the process implementation is internal to the Ara system, the complete ensemble of

1. See the subsequent discussion on thread implementation.

agents, interpreters and core runs as a single application process on top of an unmodified host operating system, which considerably facilitates porting to specific platforms. Fig. 10 depicts this basic architecture of the Ara system as a refinement of fig. 1.

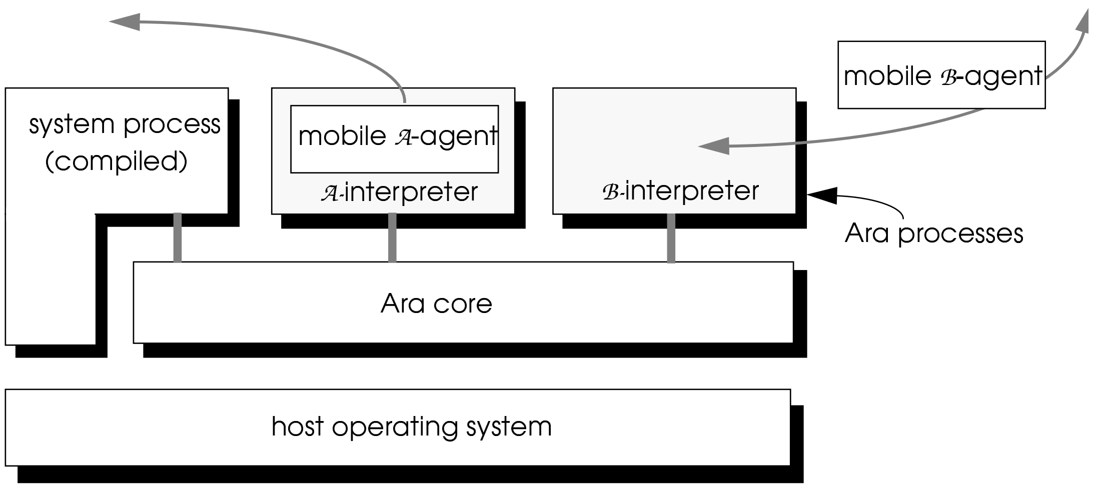


Fig. 10: Ara System Architecture

Ara processes are implemented as individual threads of control within a common address space[1], as provided by the host operating system. The core contains an efficient threads package, executing the threads concurrently in a non-preemptive way. Synchronization primitives such as messages and semaphores are employed for internal purposes; application agents, however, are presented higher-level facilities such as service points. The common address space allows highly efficient process management and interaction. Moreover, thanks to the non-preemptive process scheduling, there is no need for synchronization in the core and in the interpreters, further benefiting performance.

The reader may have noticed an apparent contradiction between the above description of process scheduling as non-preemptive, and the statement of section 3.3.3, “Agent Scheduling” that the core enforces time sharing between the agents by preemption. This seeming contradiction is resolved when discriminating between processes, which execute machine code at the processor level, and interpreted agents, which interpret a program at the language level[2]. From point of view of the agent, execution is preemptive, since control may be withdrawn from the agent between any two primitive instructions in its program. This is implemented by a core function for time slice surveillance, which is called by the interpreter out of the agent’s reach at some regular interval. This function performs a voluntary, i.e. non-preemptive process switch whenever it finds the agent’s time slice exhausted. This concept provides the preemption required for the execution of untrusted mobile agents, without introducing the complications of asynchronous processor contexts.

1. Note that the interpreted agents are completely protected from each other nevertheless (see section 5.3, “Protection”).
2. In the case of compiled agents, the level of interpretation does not exist, and scheduling is indeed non-preemptive even from point of view of the agent.

As the time slice surveillance function is called at a regular interval, this interval also utilized to serve as the time quantum for the logical clock governing the timing service explained in section 3.4.

5.2 The Communication Process

The communication process (“comm-process”) is the most prominent system process, handling the network interface of the host system for the purpose of agent migration. It accepts outbound agents in the form of a linearized byte array (“agent parcels”) and sends them to the comm-process at the parcel’s destination site. Conversely, it receives inbound parcels and passes them on to the core to recreate the living agent from them. The comm-process can handle any number of inbound and outbound parcels in parallel by interleaving the individual transmissions. This accounts for transmissions to or from sites with low-bandwidth connectivity without harming the throughput on high-speed connections.

The comm-process waits in blocked state while there are no ongoing parcel transmissions. Remembering the discussion in section 3.3.3, “Agent Scheduling” on blocking I/O operations, as soon as there are any transmissions the comm-process starts polling the set of them in a loop interspersed with explicit time slice checks (and implied potential releases of control) until all transmissions are finished, whereupon it will block indefinitely again. This scheme realizes network I/O in parallel with other processes executing, without consuming system resources while no transmissions are in progress.

Agent parcels are transmitted as a single, unidirectional data packet from the source to the destination site comm-process. The transmission may be acknowledged, if the concrete transport protocol supports this, but this is not required. At the time of this writing, the comm-process uses TCP as its only transport protocol, with an optional gateway to the AX.25 radio transmission protocol[1]. However, further protocols such as SMTP (e-mail) [CRO82] or HTTP [BFF96] will be added, and it is planned to let the comm-process choose a protocol for each parcel transmission adapted to the current connectivity, provided there is more than one alternative route to the destination place available.

5.3 Protection

Mobile agents executing within an Ara system are protected from each other, as is the core, in order to prevent malfunctioning or malicious agents from spying or tampering outside their own boundary. Protection in this general context means control of the data read and written and the external functions called by an agent; higher-level issues of object access authorization, such as entering a place, or fetching requests from a service point, are treated specifically.

Since mobile agents are interpreted, and since an interpreter obviously has complete control over the interpreted program, protection can be achieved independent of hardware facilities like privileged processor modes or page protection. Concerning the functions called, protection is trivial, since every call has to pass through a stub defined by the interpreter, which can be trusted to be correct since it is out of the agent’s reach. Regarding data access, on the other hand, protection is achieved through an address space concept: Borrowing from operating systems terminology, the set of data visible to an agent might be viewed as the agent’s address space — both if the data exists in the form of a randomly addressable memory area (as in C), or in the form of a set of unordered symbolic variables (as in Tcl or Java). The interpreter ensures that each agent has direct

1. This is used for experimentation within the Ara project, and is not included in the Ara software distribution.

access to its own address space only, while core functions must be used to interact with other agents and the core. The core itself and its objects, such as (other) agents and service points, exist outside the agents' spaces in "real" memory.

Confining access to an agent's own address space is trivial in the case of exclusively symbolic variables (since each interpreter maintains its own set of these). If, on the other hand, the interpreted language allows random memory accesses, the interpreter must provide a virtual memory image to the program, using some kind of address checking and translation. The MACE interpreter for the C language is an example for this.

In any case, agents need a means to name core objects existing outside the agent's address space. The object ids already mentioned in section 3 provide this means, in addition to their purpose of uniquely naming the core objects. Object ids have a representation in the agent's language (e.g. a string in Tcl) which can be viewed as an opaque "pointer" into core space. When a core function is called by an agent, argument objects are supplied in the form of ids, which are mapped to their objects by the stub for this function, using an object table maintained by the core. Again this is reminiscent of operating systems such as Unix, where applications name kernel objects such as files indirectly through ids, though not usually globally unique ones. The stubs also perform any interpreter-specific parameter checking, such as checking addresses for containment in the program's address space. This mapping and checking work has been placed in the stubs rather than in the core functions themselves in order to avoid unnecessary work when a core function is called from a trusted context, e.g. from the core itself.

5.4 Saving and Restoring the State of a Process

At the heart of mobile agent implementation in Ara is a mechanism to save the state of a process and restore it after transportation to another site. As explained in section 1.3, "Agent Mobility: Going from Place to Place", Ara agents can move without interfering with their execution, i.e. they continue from the same execution state in their program as they had reached when leaving the source system. This "orthogonal" migration implies that their execution state has to be extracted from the source system and reinstalled into the destination system as illustrated in fig. 11.

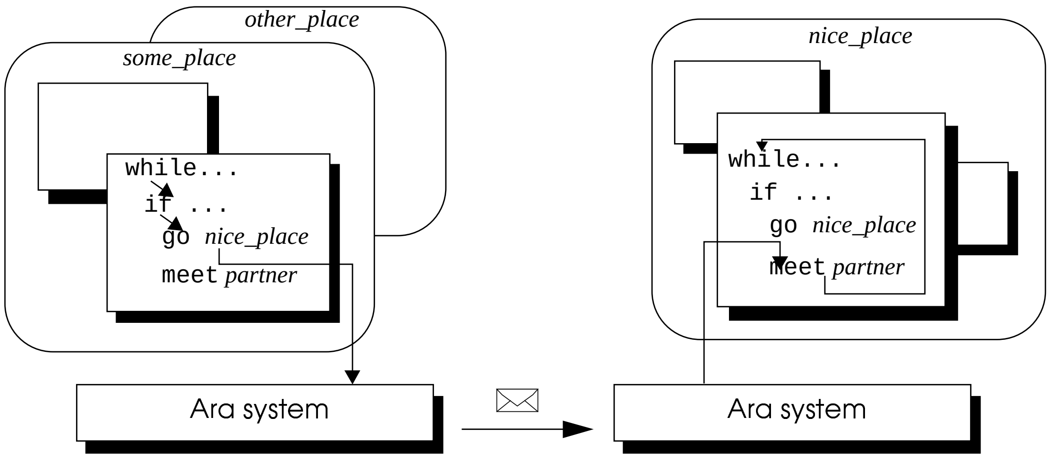


Fig. 11: Orthogonal Migration between Ara Systems

Such an extraction and reinstallation is a delicate procedure, the more so when the participating machines have different architectures, since the execution state of a process generally exists in a machine-dependent format, such as assuming a specific word length or byte order. In Ara, the state of an agent is transformed into a portable form prior to moving. This transformation is facilitated by the Ara system architecture: An agent's state is composed of the state of its specific interpreter on the one hand, and the state of its underlying general Ara process in the core on the other hand. The core performs the portable transformation of the Ara part on its own, whereas it uses a dedicated function defined by the interpreter ("upcall") for the other part. This function may build its implementation on a number of utility functions offered by the core, providing the transformation of common data types into a portable form, as well as a general concept to transform the states on the individual levels of a procedure call hierarchy on the run-time stack (see below). The run-time overhead of the complete transformation into the portable form (remembering section 5.2, this is called parcelling a process) basically depends on the complexity of the program state at the time of migration, and on the general complexity of the interpreter's state representation.

Interpreters for most interpreted languages are implemented in a procedural language (in C, most of the time). In procedural languages, the execution state of a running program is to a large part contained in the program's run-time stack at a given instant. This causes a problem with the normal implementation of interpreters when trying to save and restore the interpreted program's execution state in a portable way, since the normal interpreter implementation intertwines that state with the interpreter's run-time stack, and the latter is invariably machine-dependent. There are basically two alternative solutions to this problem: Either re-implement large parts of the interpreter, completely replacing the use of the run-time stack by a new scheme[1], or continue as before during normal execution, but transform the run-time stack into a portable representation when state saving or restoring is needed.

The primary design considerations of Ara's scheme for saving and restoring execution states were efficient overall execution and sufficient generality as to be applicable to any software implemented in C, while keeping that application so straightforward that it can be easily extended to unknown software (e.g. interpreters for additional languages) without understanding its internal workings. For these reasons, Ara adopted the latter of the above alternatives, a portable transformation of the current run-time stack at migration time. The devised scheme consists of an annotation of the source code of a given interpreter implemented in C, which requires only a tightly localized[2] understanding of the source, can be automated to a large part, and adds no measurable penalty to normal execution speed. This allows additional language interpreters to be adapted to the Ara core with reasonable effort. The adaption procedure has been applied so far to the Tcl interpreter[3] and the MACE interpreter, and is currently being applied to the Java interpreter.

1. This usually involves some explicit substitute for the run-time stack, as is done e.g. in a stack machine.
2. In nearly all cases, limited to one C function.
3. For the purpose of validation, an automatic saving and immediate restoring of the program after *every single* program step can be arranged. The modified Tcl interpreter passes this validation for the complete official Tcl test suite without an effect on its function.

5.5 Cloning and Checkpointing

Both cloning and checkpointing involves saving the state of a live process and restoring the process from this — cloning "restores" a copy of the saved process immediately, while checkpointing defers restoration until explicitly demanded. The problem of saving and restoring a process's state, however, has been solved to implement migration in the first place. It should come as no surprise, therefore, that the implementations of cloning and checkpointing are mostly based on the migration implementation. In fact, their realization was remarkably simple by falling back on migration: In a sense, a process cloning itself is realized as a process migrating to the local place, but being duplicated on arrival. Analogously, checkpointing is implemented like a process "migrating to the disk".

There is, however, one drawback with this implementation. Since processes migrating away from the local system leave behind their external state (see section 1.3, "Agent Mobility: Going from Place to Place"), this also applies to processes created by cloning and those restored from a checkpoint. For the case of cloning this is unfortunate, since the objects of the process's external state still exist unchanged after cloning, so that cutting the clone's relations to them is not really necessary. It is possible that the cloning implementation will be extended accordingly in a future release. The case with checkpointing is different, however, since the objects of the process's external state might indeed change arbitrarily between checkpointing and restoration, so there is no point in trying to preserve the external state.

5.6 Adaption of Further Programming Languages to Ara

One of the central motivations for the separation of language interpreter and system core in the Ara architecture was the possibility to later add further agent programming languages as they seem useful. To this end, the language interpreter must be adapted to Ara in various respects. This adaption, sketched in fig. 12, is well-defined and straightforward on the part of Ara.

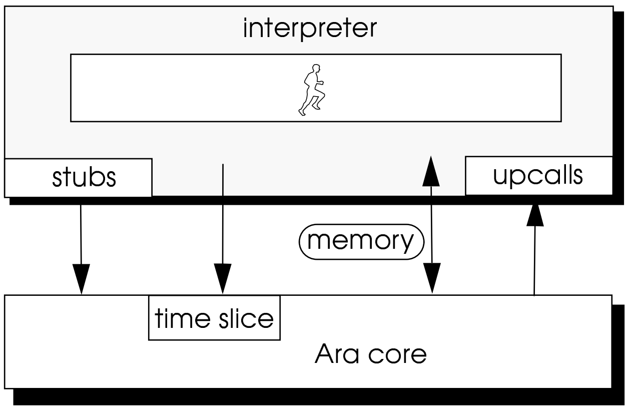


Fig. 12: Adapting a Language Interpreter to the Ara Core

In contrast to the interpreted agent, the interpreter itself is conceptually a part of the Ara system, is trusted, and supports the core in its tasks pertaining to the interpreted agent. The duties of an interpreter in this respect include the definition of calling interfaces, i.e. stubs, in its programming language for the functions imported from the core, and conversely to provide functions for interpreter management ("upcalls") to the core. The work of the stubs is mostly a matter of data format conversions and similar interface translations; in particular, they must map between object ids for core objects and the objects themselves, as described in section 5.3, “Protection”. Regarding the interpreter upcalls, the functions for the portable state extraction and restoration described in section 5.4 are the most prominent here; others include system and interpreter creation and deletion, and a function to start the interpretation. A general requirement for the interpreter is to satisfy demands for dynamic memory arising during the interpretation exclusively by the memory management functions provided by the core (the same as called by the stubs of section 3.11). This ensures that any memory consumption on behalf of the interpreted agent is properly accounted to its allowance. Further, the interpreter must possibly provide a virtual memory image as described in section 5.3. Finally, the interpreter has to assist the core in preemption, performing regular calls to the core function for time slice surveillance, as described in section 5.1, “Processes and Internal Architecture”.

In principle, any interpreted programming language can be adapted to Ara: A given interpreter must be extended by mechanisms to serve the described functions. The main expense in the adaption of a concrete interpreter is usually spent in the implementation of the state extraction and restoration, the complexity of which directly depends on the complexity of the state representation in this interpreter. While space does not permit to detail a real language adaption here, it is instructive to look at an exemplary adaption of an ad-hoc interpreter for some toy language. The Ara software distribution contains the source code for such a toy interpreter adapted to Ara. Note that programmers adapting a new language interpreter to the Ara core will need to use additional core functions, exceeding the application functions covered in section 3, “Ara Programming Concepts and Features”. For information on these, please turn to the software distribution , where they are collected in a declaration file.

6. The Hows and Whys of Mobile Agents

The field of mobile agents is a young one, with many more or less experimental systems appearing (see section “References”) and exploring the merits of diverging solutions. It seems appropriate, therefore, to conclude this report with a few thoughts on some current issues of debate in mobile agent technology — naturally, with a special look on the Ara system every now and then. At the beginning, there is a most fundamental question.

6.1 Why Mobile Agents at all?

Theoretically, mobile agents do not provide any new functionality over conventional message communication. A host could provide a remote access interface to the operations otherwise designated for locally visiting agents, such that any application could run at its home site, accessing the functions of that remote host by means of message exchange, e.g. by RPC. However, while such a scheme could indeed produce a behavior which is functionally equivalent to the corresponding mobile agent solution, this argument misses the point. The actual power of the mobile agent concept lies in the non-functional properties it endows a computation with, such as performance, resource efficiency, robustness, and ease of use. Mobile agent applications can perform better at processing significant quantities of remote data, they can make more efficient use of communication bandwidth, and they are more robust against machine and connection failures; section 2, “Mobile Agent Applications with Ara” illustrated this, so that examples will not be repeated here.

Viewed more abstractly, there is a tension between the desire to provide a flexible, general server interface and the need to perform complex service requests efficiently. A simple and clean interface is more flexible, since complex requests can be satisfied by combination of simpler ones and a certain amount of postprocessing; however, the work of combination and postprocessing must be performed at the client site, demanding increased remote communication. A rich server interface, on the other hand, avoids this problem by serving even complex requests by dedicated server-side functions; however, in practice it is not feasible to anticipate all such functions potentially desired by clients. Even more, many servers simply cannot be expected to provide a reasonably flexible interface. For example, remote information retrieval by mobile agents may be functionally equivalent to remote database access; however, the bulk of the information available worldwide does not exist in structured databases at all. Mobile agents can release this tension, as they effectively make the server exhibit exactly the function desired by the concrete application.

In addition to these non-functional, but clearly measurable advantages, mobile agents may also prove beneficial as a general concept in the design of networked applications, as proposed in [HCK95]. Many individual problems solved by mobile agents could also be solved by different means, e.g. more powerful protocols, sophisticated caching, extended server functionality, and richer parametrization. However, from a software engineering point of view, one uniform concept seems clearly preferable over a number of individual solutions.

The potential elegance of mobile agent programming shows when comparing the application programmer's view of a remote interaction through mobile agents with a message-based scheme: The latter is often complicated by the need to decompose the desired remote action into a number of messages, each of which must be sent separately, and may fail separately, all of which must be handled by the application. Mobile agents, on the other hand, travel in one atomic act, and are not subject to partial failure during the interaction.

Aside from the technical merits, it is rather a different question to what extent service providers will be willing to let mobile agents enter their premises. Quite apart from host security concerns, which may be resolved technically, providers might be reluctant to share control over how their services will be used with mobile agent manufacturers. Only time will tell how realistic this concern will be, and how strongly customers will demand mobile agent support on the other hand. For concerns about the resources consumed by unsolicited visiting agents, see the payment discussion in section 6.4.

6.2 Languages

Although various realizations of mobile code in some form have been known for a significant time [FAL87, STG90, STO94], they have been quite specific and have not had a major impact. The idea of mobile agents as a general networked computing concept has gained momentum only recently, most prominently through the Telescript language [WHI94, GEM95] and, in a more specific context, the Java language environment [GJS96]. Both these systems are centered around their own new programming language for mobile code or agents, and it is not surprising that the language, referred to as a "network programming language", is frequently seen as the central issue of mobile agent technology.

While it is certainly legitimate to ask what kind of programming language would be best suited to express mobile agents, focussing too much on the language might unnecessarily impede the advancement of the concept. Firstly, the integration with a specific, "mobile" programming language, with its own merits and shortcomings, easily blurs the borders between language and mobility, unnecessarily obscuring the concept.

Secondly, the adoption of a new language constitutes a considerable hurdle in practice, requiring new skills and tools and hindering the interoperation with existing software. This critique of mobile programming languages might be countered with the argument that the integration, while inevitably “obscuring” the borders between the integrated concepts, as well as requiring additional adoption expense, does indeed yield a significant benefit in programming. However, it is instructive in this respect to draw an analogy from the field of conventional distributed computing to mobile agents. During the rise of distributed computing throughout the 1980s, a considerable number of distributed programming languages [e.g. BLA87, BAL90] appeared, integrating distribution concepts as language primitives. Although several of them reached a remarkable maturity, none has been able to spread significantly; instead, libraries and run-time systems providing distribution functionality [e.g. OMG96] have achieved pervasive usage. The reasons for this are manifold, but a primary one is certainly the fact that distribution handling is not intertwined so intimately with the local processing as to require language support very strongly[1], relative to the disadvantages of changing the language. Considering that with mobile agents, mobility tends to surface even less frequently than, say, RPC in conventional distributed programming (remember that replacing expensive remote interaction was one of the initial motivations of mobile agents), this reasoning applies even more to mobile programming languages.

This is why Ara takes the approach of providing mobile agent functionality in the language-independent system core, with the language-specific interpreters adding basically only an interface between the core and the agents. The price for this flexibility is a certain restriction and overhead in agent interaction; for example, shared variables cannot be used as a means of communication between agents, since the implementation of a variable is specific to its language.

Most nascent mobile agent platforms originate from a specific programming language [CHT95, CMR+96, GRA96, HMD+96, LAN96, LI96, PED96, RAS+97, SBH96], but many have declared language independence a goal. However, only very few implementations [e.g. JRS95] actually offer more than one language at the time of this writing. It is remarkable in this respect that even the seminal Telescript system — a prototypical example of the integration of language and system — has recently been suggested by its creator to play the role of one of several language environments on top of a common platform [WHI96].

Quite apart from programming the agents’ actions, the term “agent language” is sometimes also applied to the language interacting agents use for mutual communication. In contrast to programming languages, there is no set of agreed basic functionality for such languages. It is clear that many applications could leverage off a structure for information exchange, but it is a current issue of research to find powerful, yet general patterns of agent communication (see [MLF95] for an example). Ara, in particular, leaves the choice of communication language open, offering only a general data exchange mechanism. For the time being, communicating agents may set up a customized interaction scheme on top of this between themselves, but it is conceivable that such a concept will be integrated into the system core at a later time.

1. Parallelism, as opposed to distribution, constitutes an instructive counter example: Parallelizing languages and compilers are well-established in high-performance computing. This can be attributed to the fact that parallelism appears and can be realistically exploited in a more fine-grained form than distribution.

6.3 Mobility

Mobility has been common in distributed systems in many forms, both mobile data and, somewhat less frequent, mobile processes. Distributed object systems and distributed operating systems in particular have strived to provide far-reaching mobility of all kinds of system objects, towards the ultimate goal of complete "location transparency", i.e. the property of an object that its physical location is neither discernible nor important. In particular, this involves system-wide, global objects such as data objects, locks, groups, communication channels etc. It might be argued that a mobile agent platform should include all these as well, to provide maximum programming convenience.

However, the target domain of mobile agents is characteristically different from that of distributed operating systems, as the latter cannot help in practice to assume a network of reasonable bandwidth and availability to make their distributed data structures work. Mobile agents, on the other hand, are targeted especially at wide-area networks with low bandwidth and intermittent availability, which tend to make global functions (i.e. those spanning more than one node) unwieldy to use. Apart from this pragmatic difference, there is also a conceptual mismatch between the goal of global functions to make location invisible and the principle of mobile agents to explicitly move between locations. This is rooted in the different underlying network assumptions again: To put it shortly, hiding distances can only work if the network is good enough; otherwise it is sensible to admit the distance and deal with it. Accordingly, the global functions in a mobile agent system should be kept to a minimum, as also advocated in [TSC95].

As a result, the Ara design has been reluctant to include global functions besides agent migration; higher-level global services should rather be built on top of migrating agents, benefiting from their robustness and flexibility. Since the inevitable overhead involved in sending a mobile agent is not justified for certain very simple instances of communication, for pragmatic reasons a simple remote communication facility is expected to be added as the only other global function besides migration.

Playing a pivotal role as described, the concept of migration deserves some additional discussion. While mobile code systems can certainly be useful without migration (e.g. the Java language environment does not directly provide migration), such systems do not lend themselves easily to tasks spanning several nodes, such as distributed information retrieval and semantic routing (see section 2.1, "Information Research"). For systems which do support migration, in practice there is a choice between two alternatives concerning the agent execution state to be moved: Either move only the global data of the agent program, restarting it at the destination site on these data [GOY95, HMD+96, JRS95, LAN96, LDD95, LI96, SBH96], or take the current execution context along as well, resuming the program from this context [GEM95, GRA96, CMR+96, PED96, RAS+97]; the latter had been termed orthogonal migration in section 1.3. These two variants of migration are theoretically equivalent, because in the first case, the execution context can be encoded in the data part, achieving a behavior which is at least functionally equivalent to the second, orthogonal case. Concretely, a copy of the program could be sent to the destination, along with some data describing its current application-level state. A fresh copy of the agent will then start at the destination site, using the accompanying data to restore the application state to that before the sending, and resume from there. In practice, however, such encoding and decoding of states would cause considerable effort to the agent programmer once the program gets larger. Even more, the overall control flow of the application program would be dominated by the presence of migration, interleaving application functionality with mobile agent functionality.

It is tempting to dismiss orthogonal migration on the grounds that it requires additional implementation expense in the agent platform, and that it can be emulated anyway. Certainly emulation is manageable for stock examples, but it would be short-sighted to leave that problem to all potential later applications, rather than solving it once and for all in the platform. This is why Ara realizes orthogonal migration: An Ara agent can migrate at any time by calling the concerned Ara core function; the system extracts the agent from the local structures then, transforms its complete state into a portable form and ships this. The receiving system performs an inverse operation, recreating the agent in exactly the same execution context, i.e. directly after the migration call. Migration is thus independent or orthogonal to program execution. This concept facilitates application development considerably, as well as the adaption of existing software. Orthogonal migration might also be termed "additive", since it cleanly adds migration to the programmer's toolbox, without hindering other tools such as structured programming or encapsulation. From point of view of the application programmer, this provides mobility in the simplest and yet most powerful way.

Note that orthogonal migration, while moving the *complete* current state, does not force a migrating agent to burden itself with unnecessary load in the case where it does not care any longer about certain parts of its state. In this case, unwanted data can simply be deleted prior to migration, and an unnecessarily complex execution context can be simplified by proceeding to a simpler context before actually migrating.

As an aside, orthogonal migration is sometimes called "transparent" instead, suggesting that the act of migration is "not noticeable" from point of view of the program state and control flow. Ara rather opted for "orthogonal", since "transparent" is usually reserved for actions not noticeable *at all* to their initiator (as with "transparent redirection"), which does not seem appropriate for migration as an intentionally initiated action. Truly transparent migration does occur in distributed operating systems managing a homogeneous cluster of processors, where computing processes are moved between the processors for reasons of load distribution without noticing this; that, however, is quite different from migration as mobile agents usually perform it.

To give due credit, Ara's concept of migration is clearly inspired by Telescript [GEM95], in particular the name of the corresponding operation ("go")[1].

6.4 Security and Payment

Security is a central issue with mobile agents, and, consequently, has been treated several times in this report, too. As a matter of fact, the focus has been on the security of the host system against undue actions of a visiting agent. This is understandable, since mobile code inevitably calls up the vision of viruses, and it is necessary, since host system operators must feel reasonably secure before they will admit any mobile agents[2]. However, host security is not a problem in itself, since the actions of an agent can be arbitrarily restricted, e.g. by interpretation. The problem is rather one of policy, to strike the right balance between restriction and empowerment of the agent.

1. The name of "meet", the operation to perform a rendez-vous between agents at the same place, is also credited to Telescript. Note, however, that the details of interaction in Telescript and Ara are very different. Telescript uses method calls across agents, while Ara effectively employs message exchange to account for agents implemented in different languages. Both concepts are synchronous, though.

2. While total security would certainly be preferable, the dangers must be weighted against the benefits, and it seems plausible that a certain level of danger will be tolerated, as it is widely the case with networking in general.

In contrast to host security, the security of the agent against undue actions of the host is a fundamental problem. Since a program is in the power of its processor by its very nature, the host has nearly complete control over the agent. In particular, the host can inspect the agent's program at will, can read and write its data more or less arbitrarily, and can alter the program execution in any way it chooses. Some of these problems cannot be solved by technical means (such as copying an agent's code), while others can be solved, but require considerable expenses, or impose undesirable restrictions.

When in the power of an untrusted host, in a certain sense the agent cannot trust itself. In general, protection of an agent against its host must therefore be arranged with the help of a third party trusted by the agent and out of control of the host in question. The most obvious candidate for this is the agent's home system; in cases where this is not feasible, unrelated third parties might play this role to some degree, provided there is no wide-spread "conspiracy" against this agent. Protecting the agent's code against tampering by the host is an example for this: The code may be digitally signed by the home system, and every system receiving the agent may verify that the code was not manipulated by the sending system[1]. In contrast to the code, protecting an agent's data against tampering or spying is not possible in general[2] — once again, anything accessible to the agent is accessible to the host as well. In particular, the suggestive idea of an agent bargaining with its host over the price of some service would fall a prey to this problem — the host could apprehend the agent's bargaining strategy by inspection of its program, and adjust its own behavior to arrive at the worst acceptable result from point of view the agent. Any application which requires privacy of the agent will have to consider this lack of protection.

In particular, this applies to applications involving electronic currency transfers. Currency schemes where the value delivery is constituted by the disclosure of a certain piece of data are not suitable for secure transactions between agent and host, since the disclosure cannot be controlled by the agent. Still, it is an attractive idea to equip mobile agents with a budget of some suitable electronic currency to purchase services on the spot. Such currency may be used in particular to compensate a host system for the resources consumed by the agent during its stay. Actually, the concept of allowances in Ara has been introduced not only for the purpose of bounding an agent's expenses, as it is presently used, but also with the intention of accounting and billing those expenses[3], and the use of electronic currency will be investigated in the Ara project. However, it is not necessarily true that in the long run, service providers would only admit an agent to their host machine if the agent is willing to pay for its stay. The situation is not too different from today's relation between providers and their remote clients: Even today, providers spend considerable expenses to run server machines to publish information and services, and the mere act of publishing, though consuming significant resources, is free for clients. Matters are different, however, concerning the offered services themselves, as opposed to the mere publication. There may indeed be a potential for electronic markets here, populated by mobile agents.

1. This method would exclude self-modifying code, but that appears to be a tolerable restriction.

2. Data which the agent collected on previously visited sites, but does not need before reaching a trusted site, constitutes an exception to this. Such data can be protected by public key cryptography, using the public key of the trusted site.

3. Actually, the consumed resources are accounted even in the present implementation for internal purposes, although there is no billing yet.

Annex A: Glossary of Ara Terms

Cross references to other terms in the glossary are printed in *italics* when occurring in descriptions.

Access Right	Operations on agents, such as terminating them, require the executing agent to possess the access right over the concerned agent. An agent possesses the access right over all agents of its own *group* and of any child *groups* thereof.
Agent	A mobile agent is a program with the ability to move during execution, while preserving its identity and state. For the sake of uniformity, even programs which do not really move are subsumed under this term. Agents bear a globally unique immutable name. Each agent is the (possibly single) member of one *group* at a time, which bears an *allowance* limiting its resource accesses. Agents are executed as parallel *processes*, usually by an *interpreter*.
Allowance	An allowance is a vector of permissions for various system resources, such as files, CPU time, or disk space. The elements of such a vector constitute quantitative permissions (e.g. for CPU time) or qualitative ones (e.g. for network domains to where connection is allowed). Each agent is equipped with a global allowance for its life time and may be further restricted by a local allowance while staying at a certain *place*. An agent shares both its global and local allowance with the other member agents of its *group*.
Checkpoint	A record of the complete *internal state* of an agent at some time. The concerned agent may be restored from the checkpoint, to resume its execution from this state.
Client	An agent which has *met* a s*ervice point*. The client may submit a *request* to the s*ervice point* in order to receive a *reply* to the *request*. An agent may play the role of client and *server* at many *service points* at a time.
Compiled Agent	An agent which is compiled to native machine code and executed directly, as opposed to being *interpreted*. Compiled agents must be absolutely trustworthy, since they are able to subvert the security measures of the system. Compiled agents cannot usually *migrate,* except in certain cases where their source code is available.
Core	The central part of an Ara system, implementing the basic concepts such as *agents*, *allowances*, *service points*, *migration* etc. Any access from an application agent to the host system or to another agent is mediated by the core for reasons of security and portability. The core treats agent independently of their programming language, using assistance from the language *interpreters* for language-specific tasks.
External State	The relations of an agent to external objects, such as other agents, *service points,* or files. These relations cannot be preserved across *migration* or *checkpointing*, as such external objects cannot be warranted to exist where the agent completes its *migration* or when it is restored from a *checkpoint*.

Fetch	A *server* may fetch *requests* submitted to a *service point*. Fetched requests are tagged with the name of the requesting *client*. The fetch operation optionally waits until there are any *requests* submitted.
Group	A group comprises a set of agents with a common *allowance*. Newly created agents join the group of their creator by default, but they may also be created to form a new, separate group, called a child of the creator's group. A child group receives a share of its parent's *allowance* and returns (what is left of) this when it becomes empty again.
Id	An id is a globally unique and immutable machine-generated identification of an Ara core object. Currently, *agents* and *service points* bear ids. Agents use ids to name objects as arguments to operations on them.
Interaction	*Agents* usually interact locally, by *meeting* at a *service point*. There will also be a facility to pass messages between remote agents in a future version of Ara.
Internal State	The complete execution state of an agent, except its *external state*. This comprises the agent's data, code, and execution context.
Interpreter	A mobile agent is executed within an interpreter for its programming language, for reasons of security and portability. An Ara system may contain interpreters for several languages on top of the common *core*. Besides executing the code of the interpreted agent, an interpreter must be adapted to the *core* in order to support it in the portable and secure execution of the agent. Currently, interpreters for the Tcl and C/C++ languages have been adapted to the Ara *core*, and an adaption of Java is on the way.
Meeting	An agent may meet a *service point* at the local *place* under its symbolic name to become a *client* at this *service point*. The meet operation optionally waits until such a *service point* exists. A *client* may leave the service point again, resolving its relation to the *service point*.
Migration	Migration is an act of active motion of an executing agent from one *place* to another, while preserving the complete *internal state* of the agent. In particular, its execution context is preserved, i.e. the agent resumes directly after the migration statement in its program. The agent's *external state* cannot be preserved. A migrating agent takes along some *allowance* and leaves its group behind, forming a new group at the destination. It is possible to make another agent migrate; however, this agent should be prepared to make sense of this.
Mobility	*Agents* are the only mobile objects in Ara. Mobility therefore appears in the form of agent *migration*.
Place	A place is a virtual location within an Ara system. An agent is always either staying at some place or *migrating* between two of them. A place establishes a domain of logically related services under a common security policy governing all agents at that

	place, by deciding on the admission of an agent attempting to *migrate* to this place, possibly imposing a local *allowance* on that agent for the time of its stay. Places bear globally unique names. The current Ara implementation provides only one place per system, admitting any migrating agent without restrictions.
Process	Agents running on an Ara system are executed as quasi-parallel processes, scheduled using a time-slicing scheme which is preemptive at the level of program instructions, but non-preemptive at the level of processor instructions. The current scheduling policy is round robin without priorities. Processes are implemented using a fast thread package in the *core*.
Reject	A *server* may *reject* a *request* submitted by a *client* instead of *replying* to it. Rejection will cause the *client*'s *request* submission to fail.
Reply	A message of arbitrary format returned as the answer to a *request* by a *server*. The *server* may also *reject* the *request* instead of replying.
Request	A message of arbitrary format submitted to a *service point* by a *client*. The submission operation will either return the *reply* to the request or a *rejection*.
Root Agent	The initial agent running when an Ara system starts up. In the current implementation, the root agent executes Tcl code read from the input stream (a terminal in interactive mode), and termination of the root agent terminates the system.
Server	An agent which has created a *service point*. The server may *fetch requests* submitted to the *service point* and *reply* to or *reject* them. The server may delete the s*ervice point* at any time, which implies closing it and *rejecting* all *requests* which had been submitted, but not yet *replied* to. An agent may play the role of server and *client* at many *service points* at a time.
Service Point	A service point is a meeting point for agents under a unique symbolic name, providing synchronous agent interaction. A service point has one *server* agent and arbitrarily many *client* agents. A service point may be temporarily closed and reopened by the *server*, affecting its behavior concerning attempts to *meet* or submit *requests*: Closed service points appear non-existent to *meet* attempts, and *request* submission attempts are *rejected*. *Requests* already submitted, but not yet *replied* to are not affected by closing. A service point is always tied to a *place*.
System Process	Various tasks within the Ara system are performed by *processes* with special privileges, e.g. to access the host operating system without mediation through the *core*. Such system processes are usually *compiled*. Apart from this, they are treated as any other *process* by the *core*.

Annex B: What May Be Expected from Ara in the Future?

The Ara system is in the midst of development, and besides completion of those features described, but noted as unfinished in this text, further concepts will extend the system. The following outlook names some fields where work is in progress or scheduled to begin.

- The most prominent new concept will be a secure and portable *host interface*; this was discussed in section 1.4. This will include multiple places with user-definable security policies, and extending the allowance concept to all types of resources.
- The power of a mobile agent as conceded by a specific host system crucially depends on the agent's identity. Rather than trusting the identity of arriving agents on good faith, a secure *authentication* scheme will be added to agent migration, optionally authenticating the agent as required. Encryption of moving agents to prevent eavesdropping and spoofing will also be offered. Standard public-key cryptography will be applied for these purposes.
- In a realistically sized network, mobile agents cannot know the places of potential relevance to their task in advance. A *directory* service will be provided for this, mapping services to places. The directory will be integrated with the service point concept by optionally publishing a service point's name and location on creation.
- Further transport *protocols* besides raw TCP, such as SMTP and HTTP, will be added as transport options for migration. A side-effect of this is that place names will become more structured, which is additionally furthered by allowing more than one place per Ara system. A place name will be a list of URLs, one for each communication protocol by means of which the place can be reached.
- The new interpreted programming language *Java* offers interesting concepts for mobile code and enjoys rapidly increasing importance in networked programming. The Java interpreter will therefore be adapted to the Ara core, as a third language besides C and Tcl, supporting mobile agents programmed in Java.
- *Tools* for working with Ara agents will be developed, such as a visual monitor, a debugger, and a WWW-based agent launcher.
- *World Wide Web* support will be added to Ara, both at the user and the system level. For instance, the system will be equipped with be a user interface through a web browser, and server agents will provide web access to application agents.
- A simple inter-agent remote *messaging* facility will be provided.

Any future developments in Ara will be published on the official WorldWide Web page for the Ara system, to be found under the address `http://www.uni-kl.de/AG-Nehmer/Ara/`. Further questions about the system can be directed by e-mail to `ara@informatik.uni-kl.de` or by paper mail to the author under the following address:

Holger Peine
University of Kaiserslautern
P.O.Box 3049
D-67653 Kaiserslautern
Germany

Annex C: The Ara Software Distribution

Caveat: The included Ara software is version 1.0alpha, which is a snapshot taken during development[1]. This means that the system is useful as it stands, but many advanced features are still missing. No special efforts to optimize the run-time performance have been made yet. As always with software in this stage of development, the presence of errors must be taken into account. However, the author constantly strives to improve the system and welcomes any suggestions and bug reports.

The Ara software is implemented in the C programming language and has been ported to various versions of the Unix operating system (Solaris, SunOS, and Linux) so far. Specifically, the distribution 1.0a contains the following:

- Installation and configuration notes
- The complete source code tree, together with make files to build the system from this
- Some simple example programs (including all described in this text)
- Documentation, containing

  - Hypertext help pages in HTML about the Ara shell and auxiliary tools, and all API features.

  - The Tcl manual, both as 'man' pages and HTML.

  - Some notes on current security breaches, adding further languages to Ara, and compiled agents.

Acknowledgments

The author wishes to express his gratitude to Peter Buhler, whose support and ideas contributed to getting the Ara project on the track, and who developed the low-level threads package used in the Ara core; to Torsten Stolpmann, the author of the MACE system [STO95]; to Markus Scherer, who adapted the Tcl interpreter to Ara [SCH95]; and to Christian Tauwel, who helped with many odds and ends around the project.

References

[BAL90] BAL, H. et al. (1990) *Orca: A Language for Distributed Processing,* SIGPLAN Notices 25(5):17-24 (May).

[BER94] BERNERS-LEE, T. et al. (1994) *The WorldWide Web*, Communications of the ACM, August, 37(8):76-82.

[BBI+93] BADRINATH, B.R., BAKRE, A., IMIELINSKI, T. and MARANTZ, R. (1993) *Handling Mobile Clients: A Case for Indirect Interaction.* Proc. of the 4th Workshop on Workstation Operating Systems (Napa, California, Oct. 14-15), IEEE Computer Society Press, pp. 91-97.

[BFF96] BERNERS-LEE, T., FIELDING, R. and FRYSTYK, H. (1996) *Hypertext Transfer Protocol -- HTTP/1.0,* Internet RFC 1945, http://ds.internic.net/rfc/rfc1945.txt.

[BMM94] BERNERS-LEE, T., MASINTER, M. and MCCAHILL, M. (1994) *Uniform Resource Locators (URL),* Internet RFC 1738, http://ds.internic.net/rfc/rfc1738.txt.

[BLA87] BLACK, A. et al. (1987) *Distribution and Abstract Types in Emerald,* IEEE Transactions on Software Engineering, SE-13(1):65-76 (January).

1. It is possible that an improved version of the software will have been released by the time of this publication; please check the Ara WWW pages for this.

[CGH95] CHESS, D., GROSOF, B. and HARRISON, C (1995) *Itinerant Agents for Mobile Computing*, Research Report RC-20010, IBM Th. J. Watson Research Center. http://www.research.ibm.com:8080/main-cgi-bin/gunzip_paper.pl?/PS/172.ps.gz

[CHT95] CHEVALIER, P.-Y. and THOMSEN, B. (1995) *Mobile Service Agents,* http://www.ecrc.de/research/dc/msa/.

[CMR+96] CONDICT, M., MILOJICIC, D., REYNOLDS, F. and BOLINGER, D. (1996) *Towards a World-Wide Civilization of Objects*, to appear in Proc. of the 7th ACM SIGOPS European Workshop, September 9-11th, Connemara, Ireland. http://www.osf.org/RI/DMO/WebOs.ps.

[CRO82] CROCKER, D.H. (1982) *Standard for the Format of ARPA Internet Text Messages*, Internet RFC 822, http://ds.internic.net/rfc/rfc822.txt.

[FAL87] FALCONE, J.R. (1987) *A Programmable Interface Language for Heterogeneous Distributed Systems,* ACM Transactions on Computer Systems 5(4), November.

[FOZ94] FORMAN, G. and ZAHORJAN, J. (1994) *The Challenges of Mobile Computing*, Technical Report CSE-93-11-03, University of Washington, USA.

[FON93] FONER, L. (1993) *What's an Agent, Anyway? A Sociological Case Study,* MIT Media Lab Agents Memo 93-01, Massachusetts Institute of Technology, Cambridge (MA), USA. http://foner.www.media.mit.edu/people/foner/Julia/

[FRG96] FRANKLIN, S. and GRAESSER, A. (1996) *Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents,* Institute for Intelligent Systems, University of Memphis (TN), USA. http://www.msci.memphis.edu/~franklin/AgentProg.html.

[GEM95] GENERAL MAGIC, Inc. (1995) *The Telescript Language Reference*, Sunnyvale (CA), USA. http://cnn.genmagic.com/Telescript/TDE/TDEDOCS_HTML/telescript.html

[GJS96] GOSLING, J., JOY, B. and STEELE, G.(1996) *The Java Language Specification*, Mountain View(CA), USA. http://java.sun.com/doc/language_specification.html

[GOY95] GOLDSZMIDT, G. and YEMINI, Y. (1995) *Distributed Management by Delegating Mobile Agents,* Proc. of the 15th International Conference on Distributed Computing Systems, Vancouver, Canada, June. http://www.cs.columbia.edu/~german/papers/icdcs95.ps.Z

[GRA96] GRAY, R. (1996) *Agent-Tcl: A Flexible and Secure Mobile Agent system*, Proc. of the 4th annual Tcl/Tk workshop (ed. by M. Diekhans and M. Roseman), July, Monterey, CA, USA. http://www.cs.dartmouth.edu/~agent/papers/tcl96.ps.Z

[HCK94] HARRISON, C., CHESS, D. and KERSHENBAUM, A. (1994) *Mobile Agents - Are They a Good Idea?*, Research Report RC-19887, IBM Th. J. Watson Research Center. http://www.research.ibm.com/xw-d953-mobag-ps.

[HMD+96] HYLTON, J., MANHEIMER, K., DRAKE, F., WARSAW, B., MASSE, R., and VAN ROSSUM, G. (1996) *Knowbot Programming: System support for mobile agents*, Proceedings of the Fifth IEEE International Workshop on Object Orientation in Operating Systems, Oct. 27-28, Seattle, WA, USA. http://the-tech.mit.edu/~jeremy/iwooos.ps.gz

[HOA87] HORTON, M.R. and ADAMS, R. (1987) S*tandard for interchange of USENET messages*, Internet RFC 1036, AT&T Bell Laboratories and Center for Seismic Studies, December. http://ds.internic.net/rfc/rfc1036.txt.

[JRS95] JOHANSEN, D., van RENESSE, R. and SCHNEIDER, F. B. (1995) *An Introduction to the TACOMA Distributed System*, Technical Report 95-23, Dept. of Computer Science, University of Tromsø, Norway. http://www.cs.uit.no/Lokalt/Rapporter/Reports/9523.html.

[KAL86] KANTOR, B. and LAPSLEY, P. (1986) *Network News Transfer Protocol,* Internet RFC 977, U.C. San Diego and U.C. Berkeley, February. http://ds.internic.net/rfc/rfc977.txt.

[LAN96] LANGE, D. (1996) *Programming Mobile Agents in Java - A White Paper,* IBM Corp. http://www.ibm.co.jp/trl/aglets/whitepaper.htm

[LDD95] LINGNAU, A. DROBNIK, O. and DÖMEL, P. (1995) *An HTTP-based Infrastructure for Mobile Agents*, Proc. of the 4th International WWW Conference, December, Boston (MA), USA. http://www.w3.org/pub/Conferences/WWW4/Papers/150/.

[LI96] LI, W. (1996) *Java-To-Go: Itinerative Computing Using Java.* http://ptolemy.eecs.berkeley.edu/~wli/group/java2go/java-to-go.html.

[LSW95] LUCCO, S., SHARP, O. and WAHBE, R. (1995) *Omniware: A Universal Substrate for Web programming*, Proc. of the 4th International WWW Conference, December, Boston (MA), USA. http://www.w3.org/ pub/Conferences/WWW4/165/.

[MLF95] MAYFIELD, J., LABROU, Y. and FININ, T. (1995) *Desiderata for Agent Communication Languages*, Proc. of the AAAI Symposium on Information Gathering from Heterogeneous, Distributed Environments, AAAI-95 Spring Symposium, Stanford University, Stanford (CA). March 27-29, 1995. http://www.cs.umbc.edu/kqml/papers/desiderata-acl/root.html.

[OMG96] OBJECT MANAGEMENT GROUP (1996) *CORBA 2.0 specification*, OMG document ptc/96-03-04, http://www.omg.org/docs/ptc/96-03-04.ps.

[OUS94] OUSTERHOUT, J. K. (1994) *Tcl and the Tk Toolkit*, Addison-Wesley, Reading (MA), USA.

[OUS95] OUSTERHOUT, J. K. (1995) *Scripts and Agents: The New Software High Ground*, Keynote address at the 1995 USENIX winter conference. http://playground.sun.com/~ouster/agent.2up.ps.

[PED96] PERRET, S. and DUDA, A. (1996) *Mobile Assistant Programming for Efficient Information Access on the WWW*, Proc. of the 5th WWW Conference, May 6-10, 1996, Paris, France. http://www5conf.inria.fr/fich_html/papers/P42/Overview.html.

[PES97] PEINE, H. and STOLPMANN, T. (1997) *The Architecture of the Ara Platform for Mobile Agents*, in Kurt Rothermel, Radu Popescu-Zeletin (eds.): Proc. of the First International Workshop on Mobile Agents MA'97 (Berlin, Germany), April 7-8th. Lecture Notes in Computer Science No. 1219, Springer Verlag. http://www.uni-kl.de/AG-Nehmer/Ara/Doc/architecture.ps.gz.

[PRI96] PRICE, R. (ed.) (1996) *Proposed ISO/IEC International Standard for HTML*, Proposal of the JTC1 joint technical committee shared by the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC). http://www.w3.org/pub/WWW/MarkUp/JTC1-SC29/Overview.html.

[RAS+97] RANGANATHAN, M., ACHARYA, A., SHARMA, S., and SALTZ, J. (1997) *Network-Aware Mobile Programs*, Dept. of Computer Science, University of Maryland, MD, USA. To appear in USENIX'97. http://www.cs.umd.edu/~acha/papers/usenix97-submitted.html

[ROU96] ROUAIX, F. (1996) *A Web Navigator with Applets in Caml*, Proc. of the 5th WWW Conference, May 6-10, 1996, Paris, France. http://www5conf.inria.fr/fich_html/papers/P41/Overview.html.

[SBH96] STRASSER, M., BAUMANN, J. and HOHL, F. (1996) *Mole – A Java Based Mobile Agent System*, Proc. of the 2nd ECOOP Workshop on Mobile Object Systems, University of Linz, Austria, July 8-9. http://www.informatik.uni-stuttgart.de/ipvr/vs/Publications/1996-strasser-01.ps.gz

[SCH95] SCHERER, M. (1995) *Ein Laufzeitsystem für mobile Agenten*, diploma thesis, Department of Computer Science, University of Kaiserslautern, Germany.

[STG90] STAMOS, J.W. and GIFFORD, W.K. (1990) *Remote Evaluation*, ACM Transactions on Programming Languages and Systems 12(4):537-565, October.

[STO94] STOYENKO, A.D. (1994) *SUPRA-RPC: SUbprogram PaRAmeters in Remote Procedure Calls*, Software – Practice and Experience, 24(1):27-49, January.

[STO95] STOLPMANN, T. (1995) MACE - *Eine abstrakte Maschine als Basis mobiler Anwendungen*, diploma thesis, Department of Computer Science, University of Kaiserslautern, Germany. German text and English summary at http://www.uni-kl.de/AG-Nehmer/Ara/mace.html.

[STR90] STROUSTRUP, B. (1990) *The C++ Programming Language*, 2nd ed., Addison-Wesley, Reading (MA), USA.

[SUN95] SUN MICROSYSTEMS, Inc. (1995) *The HotJava Browser*, http://java.sun.com/java.sun.com/HotJava/index.html.

[THB96] THISTLEWAITE, P. and BALL, S. (1996) *Active FORMs*, Proc. of the 5th WWW Conference, May 6-10, 1996, Paris, France. http://www5conf.inria.fr/fich_html/papers/P40/Overview.html.

[TSC95] TSCHUDIN, C. (1995) *Messengers and Object-Oriented Agents*, Position paper for the workshop “Objects and Agents” at the 8th European Conference on Object-Oriented Programming, Aarhus, Denmark. http://cuiwww.unige.ch/OSG/Vitek/Agents/christian.ps.gz.

[WHI94] WHITE, J. (1994) *Telescript Technology: The Foundation for the Electronic Marketplace*, General Magic, Inc., Mountain View (CA), USA.

[WHI96] WHITE, J. (1996) *A Common Agent Platform*, position paper for the Joint WWW Consortium / OMG Workshop on Distributed Objects and Mobile Code, June 24-25, Boston, MA, USA. http://www.genmagic.com/internet/cap/w3c-paper.htm.

[WLA+93] WAHBE, R., LUCCO, S., ANDERSON, T. and GRAHAM, S.L. (1993) *Efficient Software-Based Fault Isolation*, Proc. of the 14th ACM Symposium on Operating Systems Principles, December, Ashville (NC), USA.