Operating System Support for Mobile Agents

Position paper for 5th IEEE Workshop on Hot Topics in Operating Systems

Dag Johansen[*] Robbert van Renesse[**] Fred B. Schneider[**]

1. Introduction

An *agent* is a process that may migrate through a computer network in order to satisfy requests made by its clients. Agents implement a computational metaphor that is analogous to how most people conduct business in their daily lives: visit a place, use a service (perhaps after some negotiation), and then move on. Thus, for the computer illiterate, agents are an attractive way to describe network-wide computations.

Agents are also useful abstractions for programmers who must implement distributed applications. This is because in the agent metaphor, the processor or *place* the computation is performed is not hidden from the programmer, but the communications channels are. Contrast this with the traditional approach of employing a client at one site that communicates with servers at other sites. Communication is not hidden and must be programmed explicitly. Moreover, pieces of a computation performed at different sites must be coordinated, placing an added burden on the programmer.

By structuring a system in terms of agents, applications can be constructed in which communication-network bandwidth is conserved. Data may be accessed only by an agent executing at the same site as the data resides. An agent typically will filter or otherwise reduce the data it reads, carrying with it only the relevant information as it roams the network; there is rarely a need to transmit raw data from one site to another. In contrast, when an application is built using a client and servers, raw data may have to be sent from one site to another if, for example, the client obtains its computing cycles from a different site than it obtains its data.

Most current research on agents has focused on language design (e.g. [W94]) and application issues (e.g. [R94]). The TACOMA project (Tromsø And COrnell Moving Agents) has, instead, focused on operating system support for agents and how agents can be used to solve problems traditionally addressed by operating systems. We have implemented prototype systems to support agents using UNIX and using Tcl/Tk [O94] on top of Horus [vRHB94].

---

[*] Department of Computer Science, University of Tromsø, N-9037 Tromsø, Norway. Johansen is supported by grant No. 100413/410 from the Norwegian Science Foundation.

[**] Department of Computer Science, Cornell University, Ithaca, New York 14853, U.S.A. Van Renesse is supported by ARPA/ONR grant N00014-92-J-1866. Schneider is supported by the ARPA/NSF Grant No. CCR-9014363, NASA/ARPA grant NAG-2-893, and AFOSR grant F49620-94-1-0198.

The remainder of this paper outlines insights and questions based on that experience. In section 2, we briefly discuss abstractions needed by an operating system to support agents. Section 3 discusses some problems that arise in connection with electronic commerce involving agents. How to schedule agents by using other agents is the subject of section 4. Some preliminary thoughts on implementing fault tolerance are given in section 5. Section 6 discusses the status of our implementations.

2. Abstractions and Mechanisms for Agents

An agent must be accompanied by data in order for its future actions to depend on its past ones. For this reason, our implementations associate with each agent a *briefcase*, which contains a collection of named *folders*. A folder is a list of elements, each of which is an uninterpreted sequence of bits. Because it is a list, it can be treated as a stack or a queue. This makes folders reminiscent of the familiar objects used to group documents. Unlike files in a traditional operating system, folders must be easy to transfer from one computing system to another, since this operation occurs frequently. Thus, elaborate index structures are not suitable for implementing the folders that accompany agents.

It is also important that agents be able to read and write folders that are bound and local to a site executing the agent. Site-local folders allow more efficient use of network bandwidth. If an agent requires certain information only when it is executing at a given site, then it is inefficient to carry along that information to every site the agent visits. A site-local folder allows an agent to leave such information behind. In addition, site-local folders allow communication between agents that are not simultaneously resident at a given site. For example, consider a flooding algorithm to deliver a message at all sites in a network. One implementation would have each agent deliver the message and then create a clone of itself at every adjacent site. Unfortunately, here the number of agents increases without bound. If, instead, an agent also records its visit in a site-local folder, then an agent can simply terminate—rather than clone—when it finds itself at a site that has already been visited.

Just as an agent's folders are grouped into briefcases, we have found it useful to group site-local folders. We refer to such a grouping as a *file cabinet*. File cabinets support the same operations as briefcases, but we expect these operations to be implemented differently. In particular, since it is rare to move a file cabinet from site to site, file cabinets can be implemented using techniques that optimize access times even if this increases the cost of moving the file cabinet from one site to another.

One agent causes another to execute using the **meet** operation, where a briefcase allows information to be exchanged between the two agents. The **meet** operation is thus analogous to a procedure call, and the specified briefcase is analogous to an argument list (with each folder containing the value of one argument). For example, an agent *A* executing

meet *B* **with** *bc*

causes agent *B* to be executed at the current site with briefcase *bc*; *A* continues executing only after *B* terminates the **meet** operation. Note that after the **meet** terminates, *B* may continue executing concurrently with *A*.

Surprisingly, no additional abstractions are required to implement our basic computational metaphor. Services for agents—communication, synchronization, and so on—are provided directly by other agents. For example, an agent moves from one site to another by **meet**ing with the local *rexec* agent. The *rexec* agent expects to find two folders in the briefcase with which it is invoked: a *HOST* folder names the site where execution is to be moved and a *CONTACT* folder names the agent to be executed at that site. The *CONTACT* folder might contain the name of an agent that is a shell or a compiler. Such an agent would expect to find a *CODE* folder in the briefcase, which it would then translate and execute. Since the contents of this *CODE* folder might be the source code for the agent that originally met with *rexec*, it is possible for an agent to travel from one site to another. Note that this scheme allows an agent to move to a destination site having a completely different machine language.

Given an *rexec* agent, it is not difficult to program a *courier* agent, which transfers a folder to a specified agent on a specified machine. This allows agents to communicate without having to **meet** (on a common machine). It is also not difficult to program our *diffusion* agent, which executes a specified agent locally and then creates a clone of itself at every site that appears in the set difference of the site-local *SITES* folder and the briefcase *SITES* folder.

3. Obtaining and Paying for Services

Once agents are employed for commerce—as some proponents [W94] of the metaphor intend—support for a negotiable instrument becomes necessary. We, therefore, decided to explore the implementation and use of *electronic cash*. Electronic cash is nothing more than an unforgeable and untraceable capability that enables its owner to obtain goods and services. By implementing an electronic analogue to a well understood concept, we hoped to produce a system that remained understandable to the computer illiterate. We also hoped that electronic cash would provide a mechanism for controlling run-away agents. Specifically, charging for services would limit possible damage by a run-away agent.

Even as simple an operation as transferring electronic cash from one agent to another turned out to be surprisingly subtle to implement. With the familiar physical form of cash, money transfers are achieved by moving physical objects (coins or pieces of paper). This works only because it is difficult to manufacture copies of such objects. In a computer system, however, "copy" is a cheap operation. The usual solution to this problem would be to employ indirection and store all electronic cash in a single trusted agent. One agent could then transfer money to another by invoking an operation provided by this trusted agent.

We must reject solutions based on indirection because they necessarily violate our untraceability requirement for funds transfers. Following [C92], the solution we adopted was to implement each unit of electronic cash (ECU) as a record containing an amount and a large random number. Only certain of these random numbers appear on the records for valid ECUs. Each agent stores records for the ECUs it owns. An agent transfers funds by placing these records in a briefcase that is then passed to the intended recipient of those funds.

The recipient of such a briefcase, however, has no guarantee that the sending agent has not already spent (a copy of) the ECUs being transferred. To solve this problem, a trusted *validation* agent is employed. This agent can check whether a record it is shown corresponds to a valid ECU. If it is valid, then a record for an equivalent ECU is returned, but this record has a new random number (effectively retiring an old bill and replacing it by a new one). An attempt by an agent to spend retired or copied ECUs will be foiled if a validation agent is always consulted before any service is rendered. Notice that using a validation agent supports our untraceability requirement, since the validation agent does not require knowledge of the source or destination of a transfer.

A second problem that we encountered in supporting electronic cash concerned implementing the exchange of funds for services. It must not be possible to obtain a service without paying for it or to pay without obtaining the service. This precludes the obvious two-step protocols, because as long as electronic cash is untraceable either party might cheat the other. For example, the customer might claim to have paid when it has not, or the service-provider might claim not to have been paid when it has. What would seem to be required is support for transactions, so that we are guaranteed that both actions ("paying" and "providing the service") occur or that neither action occurs.

We rejected adding support for transactions to our system for two reasons:

(1) Having such a mechanism would impact performance and would be effective only if it were trusted.

(2) Such a mechanism would be alien to the computer illiterate, because such a mechanism does not exist in current business practice.

Our solution was to employ the threat of audits, a scheme that is well-known in current business practice.

- Participants document their actions so that a third party (a court, in real life) can perform an audit to find violations of a contract.
- An aggrieved agent requests an audit.

Documenting actions sometimes requires the presence of a third agent and the use of cryptographic protocols—we omit the details here. Having to interact with such a third agent will be familiar to computer-illiterate users (at least, to those who have purchased a house).

4. Scheduling

In our prototypes, scheduling allows the enforcement of policies that govern when and where an agent is executed. Sites in a computer network are presumed to be autonomous, so facilities must be provided for system administrators to control the resources comprising a site. Agents are also presumed to be autonomous, though. Thus, implementing support for scheduling requires mechanisms to match the needs of agents with the providers of services while, at the same time, respecting constraints imposed by system administrators.

Scheduling is implemented by *broker* agents, which are ordinary agents whose names are well known. Some broker agents maintain databases of service providers; these brokers serve as matchmakers. An agent that requires a given service consults a broker to identify which agents provide that service. Brokers are expected to communicate among themselves and with the service providers, so that requests can be distributed amongst service providers based on load and capacity. The problem of maintaining the requisite state information and intelligently distributing service requests seems to be equivalent to that of routing in a wide-area network. We do not yet have experience with various routing protocols to know how they can be adapted to this new setting, but this is a topic under investigation.

Another use of broker agents is to enforce some *protected* agent's policies with regard to **meet**ing other agents. This is accomplished by keeping the name of the protected agent secret from all but its broker. The broker, then, provides the only way to **meet** with the protected agent. To do this, the broker maintains a folder for each agent that has requested a **meet**ing with the protected agent. This folder contains the agent that has requested the meeting (along with its briefcase). Notice that this scheme is possible only because folders are uninterpreted and typeless and, therefore, can themselves store agents and sets of folders.

5. Fault-tolerance

It is to be expected that sites in a computer network will fail. When such a failure occurs, agents at that site are no longer able to continue executing. To deal with this problem, we have been investigating ways to ensure that a computation can proceed, even though one or more of its agents is the victim of a site failure. The solutions we have studied involve leaving a *rear guard* agent behind whenever execution moves from one site to another. This rear guard is responsible for (i) launching a new agent should a failure cause an agent to vanish and (ii) terminating itself when its function is no longer necessary (because the agent it protects is itself ready to terminate). The details of implementing rear guards efficiently are complex, because the sites traversed by an agent computation may be cyclic and because a single agent may clone itself and fan out through a network.

6. Prototype Implementations

Our most recent version of TACOMA is based on Tcl [O94]. Each site in our system runs a Tcl interpreter, which provides the place where agents execute. An agent is implemented by a Tcl procedure; the text of the procedure is stored in the agent's *CODE* folder. Folders, briefcases, and file cabinets are Tcl data structures. File cabinets can be flushed to disk when permanence is required.

A collection of system agents provides a variety of support functions. The most basic of these is *ag_tcl*, which pops a Tcl procedure from the *CODE* folder and executes that procedure. Currently, two implementations exist for the *rexec* agent. The first uses the UNIX rsh command to start a Tcl interpreter on the remote host. The second uses Tcl/TCP, an extension to Tcl that allows Tcl processes to set up TCP communication channels. We are now completing a third implementation based on Tcl/Horus, a version of Tcl that uses Horus [vRHB94] to support group communication and fault-tolerance.

In our first prototype of TACOMA, we implemented the electronic cash of section 3. The implementation used the security mechanisms provided by UNIX; this simplified our implementation, but relies on UNIX for security. We are now investigating alternatives.

Our TACOMA prototype currently supports a scheduling service that assigns to processors based on load. It uses four different agents to implement a scheme like that outlined in section 4. One of these agents is the broker, another is responsible for monitoring the status of a site and reporting that to the brokers, one is a courier, and one issues tickets to allow access to the service.

To evaluate the metaphor we are using our prototype to construct a variety of distributed applications. First, we are reimplementing StormCast [J93], which uses a set of expert systems to predict severe storms in the Arctic based on weather data obtained from a distributed network of sensors. Second, we have started to build an interactive mail system where messages are implemented by agents.

References

[C92] Chaum, D. Achieving Electronic Privacy. *Scientific American* 267,2 (Aug 1992), 96-101.

[J93] Johansen, Dag. StormCast: Yet another exercise in distributed computing. *Distributed Open Systems* F.M.T. Brazier and D. Johansen, eds. *IEEE Computer Society Press*, California (Oct 1993), 152-174.

[O94] Ousterhout, John K. *Tcl and the Tk Toolkit* Addison Wesley, Reading, Massachusetts, 1994.

[R94] Riecken, D. (guest editor). Intelligent Agents. *Commun. of the ACM* 37,7 (July 1994), 19-21.

[vRHB94] Van Renesse, Robbert, Takako M. Hickey, and Kenneth P. Birman. Design and Performance of Horus: A Lightweight Group Communications System. Technical Report TR 94-1442, Department of Computer Science, Cornell University, Aug 1994.

[W94] White, J.E. Telescript Technology: The Foundation for the Electronic Marketplace. General Magic White Paper, General Magic Inc., 1994.