Adding Mobility to Non-mobile Web Robots

Nils P. Sudmann and Dag Johansen
Department of Computer Science
University of Tromsø, Norway
nilss@cs.uit.no, dag@cs.uit.no

Abstract

In this paper we will show that it is possible to combine mobile agent technology with existing non-mobile data mining applications. The motivation for this is the advantage mobile agents offer in moving the computation closer to the data in a distributed system. This can save bandwidth and increase performance when the data is condensed as a result of data mining.

1 Introduction

Data mining traditionally involves huge data sets which may be distributed over a large number of nodes in a heterogenous network. Data mining applications are traditionally implemented as fixed clients pulling data from remote servers and doing data mining locally at the client. Since data mining algorithms seek to create a meta description of the mined data which is more compact than the data itself, there is a possible gain in executing these algorithms at the servers themselves.

Mobile agents are able to relocate themselves to the source of the data, and this can be a potential way to achieve better performance by avoiding pulling data over the network. Furthermore, this approach preserves bandwidth for other uses.

Merging non-mobile data mining applications with the technology of mobile agents, requires a new kind of agent system. Specifically, the need to support existing data mining applications with little or no modification, requires that the agent system supports different implementation languages and mechanisms that enables agents to move non-mobile software objects between sites.

The first mobile agent systems [14, 8, 3] preceded the Java programming language. However, since its introduction, most, if not all new mobile agent systems have been built using Java and its associated Java Virtual Machine (JVM). Many systems of this kind are publicly known, but in essence, they provide the same functionality; move a Java agent from one host to another. One advantage of using JVM is the potential portability, another is a matter of deployment. A major disadvantage, however, is that many design choices are now limited by the security and programming model enforced by Java and JVM. As such, we have kept a separate branch in the TACOMA (Tromsø And COrnell Moving Agents) project focusing on mobile agent systems based on other languages than Java.

Another interesting aspect of the multitude of mobile agent systems developed is the amount and variety of system support included in these systems. The problem has been to define the scope of functionality that should be offered in agent systems. We will argue that agents should be able to carry with them whatever system support they need, thereby keeping the required functionality of the landing pad at a bare minimum.

A second generation agent system has been built based on more than 6 years of experience with mobile code [5, 7]. During this time we have applied mobile agents in a wide specter of distributed application domains, including data mining, distributed multi-media processing, software management, and distributed alarms [4]. In this paper, we will present how a single, but very important lesson from applied mobile agents, have influenced the design of TAX 2.0 (TAcoma on uniX). Also, we will show how these design choices have made it possible to encapsulate and move non-mobile code.

The outline for the rest of this paper is as follows; in section 2 we introduce one of the important lessons we have drawn from the time we have worked on agent systems and the impact it had on the design of the TAX 2.0 system. In section 3 we briefly describe the TAX system. Then, in section 4, we discuss the scope of system support in a mobile agent system and the need for a mechanism that supports composite agents. A case study using the TAX concepts and mechanisms introduced, is given in section 5. Finally, section 6 concludes the paper.

2 An important lesson; language diversity

The advantages of using Java as a basis for *developing* a mobile agent system makes it a prime choice for this task. Java provides a uniform execution environment (at least in theory), security model, and safety properties across heterogenous platforms. The usefulness of these properties in mobile agent systems can be said to have been proven in concept when we observe the number of Java based agent systems developed by both academia and more commercial organizations[1].

However, what is less clear is how well suited for application development these agent systems are. It has been argued that mobile agents are a useful paradigm in the development of larger distributed applications. But, in our experience, larger distributed applications tend to consist of interconnected modules from a wide variety of vendors.

One aspect of this is that modules, being a mobile agent or not, are developed in the most suited programming language. This comes in direct conflict with our previous observation, that most agent systems are Java based, and that all mobile agents for these systems are restricted to Java or byte-code which can run on a JVM. Thus, while a JVM based agent system is perhaps the most easy to implement, the end product may not be as useful to the application programmer.

One of the most important lessons we drew from agent application development in TACOMA is that supporting mobile agents written in *different programming languages is a good thing*. TACOMA has been language independent from the very beginning, and application developers have aggressively used this freedom and developed mobile agents written in the most appropriate language for the task at hand.

For instance, one student project constructed a distributed pipeline to manipulate video streams in the MPEG format. This project used mobile agents written in C, a Perl script as glue between the web server and TACOMA, and finally a Java applet to control the pipeline using a standard web client.

However, we have also experienced that developing an agent system *supporting different languages is hard*. Each language has to be integrated into the system model, even though they have widely different security models, requirements on their execution environment, and safety properties. However, in our experience, it is worthwhile when we look at the benefits for the developers using the system.

Another observation made by application developers was that *safety enforcement is not always needed nor desired*. This means that restricting agent systems to the security model of a single programming language, usually end up offering agents a severely restricted execution environment. This introduces additional overhead even if the agent is working on behalf of a systems administrator and can be trusted. If sufficient trust can be achieved, an agent should have all the capabilities of a regular process.

[1]List of known mobile agent systems, October 1999; `http://www.informatik.uni-stuttgart.de/ipvr/vs/projekte/mole/mal/mal.html`

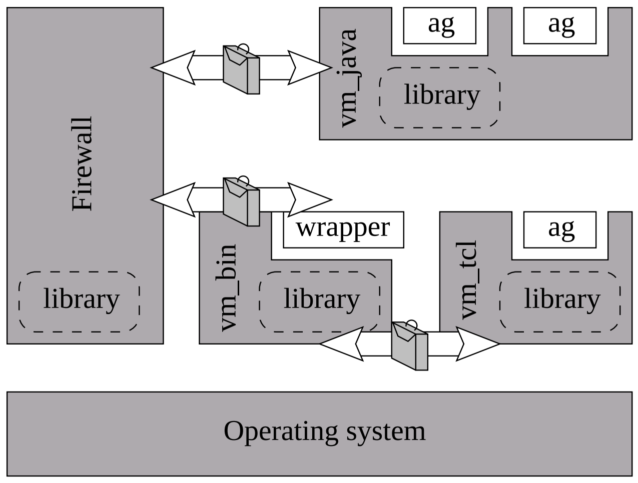

Figure 1. TAX overview.

3 TAX 2.0

We have employed our experience in the building of two new agent systems, one is an experimental system addressing the larger issues of an agent computing environment (TOS/ACE) [6], the other, which is the subject of this paper, is a refinement of the original TACOMA system supporting multiple languages.

TACOMA 2.0, or TAX, is the latest in a series of prototype agent systems developed for Unix dialects. The prototypes have been evaluated by employing agents as part of several projects and as a tool in a larger distributed setting called StormCast [5].

TAX is an architecture (figure 1) which supports several different programming languages through the notion of *virtual machines* (VMs). Different virtual machines run as separate heavy weight processes and are protected from each other through the memory protection of the operating system. Communication between different virtual machines goes through a local *firewall*. The firewall acts as a reference monitor and mediates all local communication between agents, and communication to remote firewalls and agents on remote machines.

The TAX implementation consists of three major components, a shared library offering the basic primitives to manage state and communicate, a firewall that acts as a communication broker between virtual machines (and consequently the virtual machines themselves).

3.1 The library

There is some common functionality needed by the different virtual machines in the system. Every VM has to offer the ability to handle the state of the agent, communicate with other agents and submit an agent to the firewall for transportation and deployment on a different VM. Much like the API of an operating system, TAX gathers these functions in a standard shared library. The advantages of using a library is that this code can be reused by virtual machines and the many programming languages that interface with C libraries.

The TAX library offers primitives to operate on state that the mobile agent collects and needs for future computations [9]. Basically, the transportable state of a mobile agent (code, arguments, results), is collected in a *briefcase*. A briefcase is then a consistent snapshot of the executing agent as it is transported between hosts. The mobile agent will always have access to and can modify its own briefcase, one side effect of this is that the agent is able to drop state no longer needed. This minimizes the volume of data that has to be transmitted over the network when the agent moves. Briefcases are also the unit of exchange between communicating agents. Each briefcase is essentially an associative array of *folders*, each containing an ordered lists of *elements*. An element is an uninterpreted sequence of bits and the most basic data type in TAX.

Furthermore, the library offers two basic communication primitives, called `bcSend()` and `bcRecv()`, which are used by virtual machines to communicate with the firewall. On top of these functions the library offers more complex functions like `activate()`, `await()`, `meet()`, which are used for synchronization and communication. Basically, `activate()` is equivalent to a send, `await()` is a blocking receive, and `meet()` is a RPC [2]. Mobility is covered by `go()` which moves the agent to another VM and terminates the current instance if the move is successful. `Spawn()` does the same, except that it creates a new agent with a different instance number, which is then reported back to the calling agent. This resembles the Unix `fork()` system call.

3.2 The firewall

Agents running on different virtual machines need to be able to communicate. The basic TAX library and the briefcases produced by it assures that agents can send data and receive data in a heterogenous environment. But, there is also a need for a central object, a broker, on each machine that has information on the agents running locally on the different virtual machines. Also, we need a local authority which enforces access rights, based on first level authentication of the origin of the agent. These are the two most important tasks of the TAX firewall.

```
tacomauri  [tacoma://hostport/] agpath
hostport   host [:port]
agpath     [principal/] agentid
agentid    name:instance|name|:instance
name       alphanum [name]
instance   hex [instance]
```

Examples:
tacoma://cl2.cs.uit.no:27017//vm_c:933821661
tacoma://cl2.cs.uit.no/tacoma@cl2.cs.uit.no/ag_cron
tacomaproject/:933821661

Figure 2. EBNF notation for agent URI.

The firewall is a multi-threaded process, where each thread guards a virtual machine. It manages communication between local virtual machines and communication to remote firewalls, enforcing access rights as it does so. The firewall also does an initial authentication, based on parameters such as the presence of a signed agent core or the presence of an authenticated and trusted sender.

The second important function of the firewall is to provide some basic dispatching and routing functionality. Messages passing through the firewall are queued with a timeout value if the receiving agent is not ready to receive, or has not yet arrived at the site. The firewall also provides basic matching functionality if the full name of the receiver is unknown. An agent is addressed by host, port, principal, name, and instance. Figure 3.2 shows the shorthand EBNF notation for an agent-URI. If the optional remote part is left out, the firewall will assume a local target. Furthermore, if the principal is left out, only two principals are considered as valid; the local system, or the principal of the mobile agent. The last part can be given as either a name, an instance number or both. Only supplying the name can be useful if one wishes to establish communication with a broader class of agents like service agents. The instance number may be used if one wishes to make sure one continues to communicate with the same entity.

Virtual machines need to be able to register and unregister agents running inside them with the firewall, in order for the firewall to be able to locate them when communication is addressed to these agents. Furthermore, agents with sufficient privileges need support for operations such as listing running agents, determining their run time, and killing or stopping agents. All this is achieved by addressing messages directly to the firewall.

3.3 The virtual machines

In TAX it is the responsibility of the various virtual machines to execute code in a safe and secure manner. Thus some level of trust has to be established in that the agent cannot act maliciously, before the VM executes the code. There are generally two ways one can achieve this. The first is to obtain sufficient trust in the code which is about to execute, for instance through digital signatures. Second, the virtual machine can achieve guarantees that the code does not act maliciously, for instance by guaranteeing the necessary safety features.

For instance, the trivial virtual machine `vm_bin` executes binaries directly on top of the operating system, provided the binary is signed by a trusted principal. In this way, the virtual machine allows the agent to execute in an efficient way once sufficient trust has been established. In general, virtual machines may use any safety mechanism most appropriate for the language it supports, like sand-boxing, PCC [10], SFI [13], code signing, and so on.

The method in which this is achieved is left to the virtual machine, the firewall simply trusts it to execute agent code safely and correctly. Virtual machines may support execution of several agents. In this case the virtual machines may, for performance reasons, resolve internal communication without involving the firewall.

Furthermore, virtual machines are essential in that they make TAX itself language independent. As mentioned, the method used by the VM to facilitate execution of agent code is left entirely up to the VM. This is not limited to security and safety concerns, but includes the question of supporting automatic state capturing.

The only other requirements placed on the virtual machines is that they issue briefcases for communication, since a briefcase is the TACOMA data structure that is language and architecture independent. Furthermore, VM must respond to commands issued by the firewall. This is needed to enable the firewall (or other agents having sufficient privileges) to control agents running on the VM.

VMs may manage some system resources by themselves, like CPU and memory, subjected to constraints imposed by the firewall. However, in order to manage arbitrary resources properly, resources other than memory and CPU time are handled by service agents. This allows resource allocation mechanisms to handle requests regardless of which VM the requesting agent is running on. For instance, to gain access to the file-system, a mobile agent interacts with the `ag_fs` or `ag_ccabinet` service agents.

3.4 Example execution

As an example of the concepts introduced above, we will examine more closely the steps an agent implemented in C goes through to become activated (see figure 3). First, the briefcase containing the agent will be delivered to `vm_c` (step 1). `Vm_c` activates `ag_cc` which extracts the code (step 2) and then activates `ag_exec` (3) with the code and the compiler as arguments. `Ag_exec` runs the compiler (4) and stores the binary in the briefcase received from `ag_cc`, and returns it to `ag_cc` (5). `Ag_cc` then returns the binary to `vm_c` (6) which uses `vm_bin` (7) to activate the agent. At this point the compiled C code contained in the briefcase arriving at the top is executed.

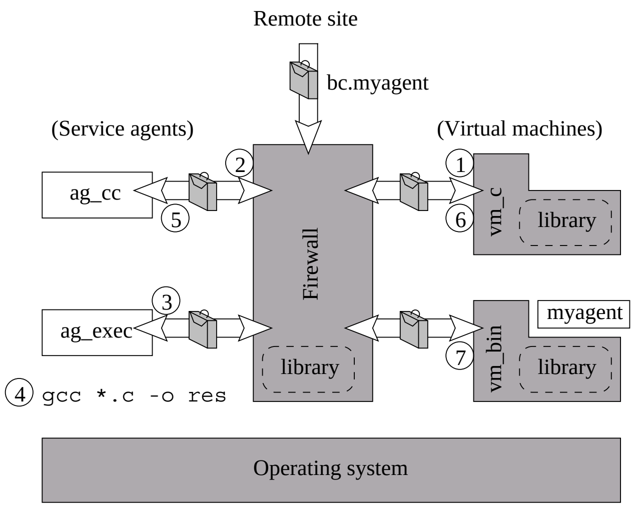

Figure 3. TAX execution example of a C agent.

Now, consider the C code in figure 4. If this is the code carried by the agent in the previous paragraph, it will when executed, display a message before removing the first element from the list of elements contained in the folder HOSTS. If this element is empty the agent terminates, else it tries to relocate to the VM specified by the URI named in the element, and the whole process repeats itself.

4 Supported functionality in agent systems

With more than 60 known mobile agent systems, one can ask questions about the difference between them. Most mobile agent system designers agree on the basic functionality; move code and run it in a itinerant style. But, beyond this their systems differ greatly in what they support. For instance, a much disputed question in the mobile agent community is if automatic state capturing of a running agent is essential or not.

Many also integrate solutions to more complex (but still traditional) distributed problems, like; location independent naming, group communication, directory services, support for transactions, and fault tolerance through active or passive replication, and much more.

```
int agMain{briefcase bc} {
  element e;
  char    next;

  while (1) {
    displaySomehow(``Hello world'');
    e = fRemove(bcIndex(bc,``HOSTS''),1);
    if (!e) {
      exit(0);
    }
    next = eData(e);
    if (go(next, bc)) {
      displaySomehow(``Unable to reach %s'',
                      next);
    }
  } /* Never reaches this point */
}
```

Figure 4. A simple hello world agent

This leads us to our next observation; *The scope of system support* is difficult to determine, some agents are single-hop and execute within the same administrative domain, while others are multi-hop and execute in the hostile environment of the Internet. To illustrate this, we use an example from data mining. A mobile agent in this application domain can be launched from a client host on an itinerant path visiting a set of server hosts containing voluminous data. On each host the mobile agent parses through a specified set of data, and brings along the (intermediate) result to the next host. This way, the client does not have to pull all the data from the remote data servers over the network for local processing. The mobile agent will, at each host, filter necessary data, and only bring back the reduced set of data that is valuable for the application.

Agents of this class may need additional functionality beyond the basic single hop agents. They may need stronger fault-tolerance and security guarantees for the agent and they may need combinations of streamed, group and/or location independent communication.

One could strive for the ultimate mobile agent system with an execution environment that supports everything we have mentioned and possible some more. However, this seems to be unpractical, a design like this makes the goal of a language independent system an even more daunting task and may even not be possible in systems tied to a particular programming language.

An important problem is where to put the needed functionality of agent applications, how much do we put into the agents, and how much should be offered by the host environment. We have argued that the scope of this functionality is difficult to determine. Also, putting this functionality into the host environment becomes a never-ending project and creates a management problem. The remaining option is to somehow put all of the required functionality into the agent itself, with the danger of creating applications with large, unwielding agents. What seems to be needed is a way to decompose agents into discrete interchangeable objects.

This in fact translates into the general problem and approach to a solution in how one should structure middleware, and resembles the conclusions reached by other projects dealing with complex compositions of middleware functionality, such as ISIS [1]. They faced the same problems with complexity and general bloatedness of the system, and this lead the a stackable architecture in Horus [12], and eventually Ensemble [11].

We suggest that agents should be constructed as a set of modules (troops of agents), in which the communication is stacked in some order. In this way, agents may carry with them the special support they need, and modules may conceptually be reused and interchangeable.

We decided to use *wrappers* to expand upon existing functionality of agents, without modifying the agents themselves. Wrappers provide a way to compose applications from different parts. We use the concept of wrappers in TAX to let agents carry with them the specific system support they need, thereby making the required set of services at each landing pad minimal.

For instance, a group communication wrapper can be used to wrap an application agent. As the wrapper is instantiated, it is given parameters such as group membership (all agents sharing common class), and desired properties of communication (casual, FIFO, atomic, etc). If the agents are to move, one can add a location transparent wrapper around the broadcast wrapper.

To make wrappers work one needs a well defined interface to wrap. We designed TAX with this in mind and created a minimal interface between agents. Agents can perform only two actions that are observable to the system: Sending a briefcase and receiving a briefcase. Sending a briefcase is equivalent with a method RMI (remote method invocation) in object oriented systems, but without static binding or any type checking. This is more error prone than a strongly typed system, but is essential in order to be able to interface different languages with a minimum of hassle and maximum flexibility for the application programmer. It is this interface a wrapper can observe and intercept messages to. Once a message is intercepted, the wrapper needs to examine it to determine the actual action attempted.

Wrappers in TAX are treated by the system as a regular agent. The system passes any briefcase from the agent to the wrapper, and any briefcase addressed to the agent is sent to the wrapper first. Wrappers may be stacked in arbitrary depth by TAX, and may originate from the local system or be part of the mobile agent itself.

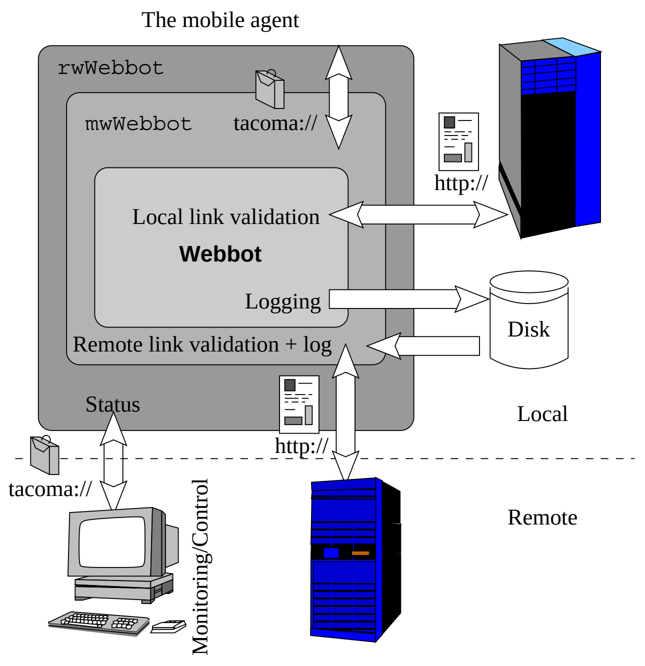

Figure 5. A wrapped Webbot.

5 Case study; mining for dead links

As web sites grow, chances increase that these sites contain references that are invalid. Such links must be detected in order for them to be corrected, a non-trivial task when a single host can contain thousands of links, and a large organization can consist of tens of these hosts. One way of traversing the structure of a web site and detecting bad links is to use a software robot. A robot can start with one or more reference pages and traverse all links in some orderly manner, gathering statistics. The problem with most robots are that they are usually executed from a host different from the web server. This induces an extra load on the network, even if its only on a local network.

In this experiment we will demonstrate how the benefits of wrappers and virtual machines enables us to construct a mobile link validation agent, using a navigation wrapper around a data mining core. We are able to achieve this by reusing an existing freely available robot and without relying on special system support in the execution environment of the web server, beyond the basic TAX agent system. This example demonstrates a general principle in which we demonstrate that mobile agents can be used to add mobility to a general class of stationary data mining applications that need to be close to their data source.

The idea here is to take a stationary web robot and encapsulate it using a mobile agent wrapper. Different robots are generally implemented in a wide variety of languages[2]; Java, Perl, C, Tcl, MS Visual Basic, Lisp, and Python with Perl and C (C++) being the most widespread choices. Here we have the first benefit of a language independent agent system, in that we freely are able to choose from already existing robots.

Webbot is one such robot from the W3C organization[3], and the one we will use in our experiment. Webbot is implemented in C and can be used to gather statistics on web pages such as link validity, age, and type of web pages encountered. Webbot gathers these statistics by following links in depth first manner, subjected to certain constraints. Constraints include depth of the search tree and restricting URIs checked to those matching a specific prefix. We will use this robot to check the validity of links on all pages located on our local computer science department web server (we assume that all pages can eventually be reached from the topmost index page).

We use the restricted prefix of Webbot in our example to keep the robot from straying off the CS department web server. However, we are also interested in the validity of links inside our CS web server pointing outside. As it turns out, the Webbot also logs links not followed because of constraints, and so we are able to check the links pointing outside our CS department server in a separate step. Furthermore, if we where to check all the servers at the university campus (the whole uit.no domain) or need the supply different base URIs, Webbot needs to be run several times, and preferably relocated to a new host between each execution.

Our wrapper for Webbot does two things. First, it relocates the Webbot binary to the web server we wish to examine and executes it at that server. Furthermore, it examines the URIs logged as rejected by Webbot, and looks these URIs in a separate step. It then combines the URIs not found to be valid with the invalid URIs logged by Webbot. The resulting list of invalid URIs and the referring pages is then transmitted back to the host of origin.

We used two TAX wrappers in this example (figure 5). The first one, called `mwWebbot`, encapsulating the Webbot by initially carrying it in its briefcase. It uses the `ag_exec` service available at all TAX sites to execute the Webbot binary once it has relocated to the web server. `Ag_exec` extracts the binary matching the architecture of the local machine (an agent may submit a list of binaries matching different architectures to `ag_exec`), and executes it with the arguments called by `mwWebbot`. `mwWebbot` then collects the results and executes step two of the task, namely checking links rejected by the Webbot binary.

In order for us to monitor and keep control of the application, we added another wrapper around mwWebbot, called `rwWebbot`. This wrapper reports back to a monitoring tool

[2]`http://info.webcrawler.com/mak/projects/-robots/active/html/ahoythehomepagefinder.html`.

[3]`http://www.w3v.org/Robot`.

about the location of the agent it wraps (`mwWebbot` and Webbot) and can be queried about the status of the computation.

In a test, the Webbot scanned 917 html pages containing 3 MBytes on our web-server. The web server contains more information than this, but since Webbot became unstable with a search tree deeper than 4, we had to limit the depth of the search tree. We found that executing a Webbot scan for invalid links on our CS department server locally is 16% faster than doing it over a 100MBit network. Critics may claim that 16% is not much, but our testing environment is very advantageous to the remote case. If the client and server is separated by a wide area network and the volume of data much greater, it is conceivable that the mobile Webbot would be even faster than its stationary counterpart.

6 Concluding remarks

Combining existing non-mobile applications with mobile agents require agent systems that are flexible and is not restricted to a single language. Building a mobile agent system supporting multiple languages and decomposition of agents is hard, but gives additional flexibility in that a wide variety of security and programming models can be supported. The problem of the scope of functionality that an agent system should offer, can be reduced by offering a mechanism that allows agents to carry with them the system support they need.

We demonstrated that this is possible by taking a piece of COTS software, the W3C freely available Webbot, and wrapping it with a mobility wrapper. This made it possible to remotely deploy the Webbot on several sites, and saved bandwidth and time in our attempt to find broken links at our CS department web server.

We have a system that supports both multiple languages and lets the agent carry with it the system support it needs. TAX has been released into public domain[4], and we are currently working on additional virtual machines, and a framework for automatic generation of layers of wrappers.

In conclusion, the ultimate fate of the mobile agent paradigm probably does not depend on the number of such systems developed, but the usefulness of these systems in developing real applications. However, we believe there exists such a usefulness.

Acknowledgments

We would like to thank Robbert van Renesse, Keith Marzullo, Kjetil Jackobsen and Kåre Lauvset for many fruitful discussions on the TAX architecture.

[4]`http://www.tacoma.cs.uit.no/index.html`

References

[1] K. P. Birman. The process group approach to reliable distributed computing. *Communciations of the ACM*, 36(12):36–53, December 1993.

[2] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2:39–59, February 1984.

[3] R. S. Gray. Agent Tcl: A transportable agent system. In *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95)*, Baltimore, Maryland, December 1995.

[4] D. Johansen. Mobile agent applicability. In *Proceedings of the Mobile Agents 1998*, Springer-Verlag LNCS series, 1998.

[5] D. Johansen, K. Jacobsen, N. P. Sudmann, K. J. Lauvset, K. P. Birman, and W. Vogels. Using software design patterns to build distributed environmental monitoring applications. Technical Report TR97-1655, Department of Computer Science, Cornell University, USA, December 1997.

[6] D. Johansen, K. Marzullo, and K. J. Lauvset. An approach towards an agent computing environment. In *ICDCS'99 Workshop on Middleware*, 1999.

[7] D. Johansen, F. B. Schneider, and R. van Renesse. *Mobility, Mobile Agents and Process Migration - An edited Collection*, chapter What TACOMA Taught Us. Addison Wesley Publishing Company, 1998.

[8] D. Johansen, R. van Renesse, and F. B. Schneider. An Introduction to the TACOMA Distributed System — Version 1. Technial report TR-95-23, University of Tromsø, Norway, June 1995.

[9] D. Johansen, R. van Renesse, and F. B. Schneider. Operating System Support for Mobile Agents. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HOTOS-V)*, pages 42–45. IEEE Press, May 1995.

[10] G. C. Necula and P. Lee. Safe kernel extensions without runtime checking. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Operating Systems Review, pages 229–243. ACM, October 1996.

[11] R. van Renesse, K. P. Birman, M. Hayden, and A. V. D. A. Karr. Building adaptive systems using ensemble. *Software - Practice and Experience*, 28(9):963–979, July 1998.

[12] R. van Renesse, T. M. Hickey, and K. P. Birman. Design and Performane of Horus: A Lightweight Group Communiations System. Technical Report TR94-1442, Cornell University, August 1994.

[13] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, December 1993.

[14] J. E. White. Telescript technology: The foundation for the electronic marketplace. General Magic white paper, General Magic Inc., 1994.