Separating Mobility from Mobile Agents

Kåre J. Lauvset* Dag Johansen* Keith Marzullo†

January 16, 2001

Abstract

In this paper we argue that the traditional model of a mobile agent provides a poor programming structure. We base our argument on our experience since 1993 in building distributed applications and mobile agent platforms. We have observed that every distributed applications contain three distinct aspects, which we call function, mobility *and* management. *Separating an application into these three aspects and programming them separately affords great flexibility and leverage when designing mobile agent applications for distributed system management. Because of these observations, we have developed a programming model and a mobile agent system in which these three aspects are clearly separated.*

1 Introduction

Mobile agents, like *processes*, are separate units of concurrent execution. They differ in how they view the processor upon which they run. For processes, the processor is abstracted away: each process can consider itself to be running on an independent virtual machine. For mobile agents, the processor is a first-class entity: a mobile agent can explicitly reference and change where it is running. We call this the *traditional* model of mobile agents.

The reasons for using mobile agents are well-known: moving computation to data to avoid transferring large amounts of data; supporting disconnected operation by, for example, moving a computation to a network that has better connectivity; supporting autonomous distributed computation by, for example, deploying a personalized filter near a real-time data source. Many mobile agent systems have been constructed and are in the public domain (for example, [2, 9, 15, 16]). But, despite these well-known advantages and widely available software, mobile agents are not yet being used as a common programming abstraction. In this paper we discuss one major reason for the lack of acceptance.

We have been working since 1993 [8], under the name of TACOMA, on operating system support and application of mobile agents. We have addressed issues including fault-tolerance [6], security [12], efficiency [7], and runtime structures and services [5]. We have built a series of mobile agent middleware systems and evaluated them by building real applications [4, 14]. We have found that mobile agents are especially useful for large scale systems configuration and deployment, system and service extensibility, and distributed application self-management. The programming model TACOMA supports has changed over these years to reflect our experience with writing real applications.

A problem we found is that, while the traditional model of mobile agents is not controversial, the programming abstraction that follows from it is poor. And, most mobile agent systems present a programming model that closely follows the traditional model. A programming model closer to the *worm* [13], first suggested by Shock and Hupp, appears to be better suited for our goals. This programming model separates the functional aspect from the mobility aspect of the computation. Doing so is an instance of the well-known design principle of *separation of concerns*.

In this paper we explain why the traditional mobile agent programming abstraction is poor. We then give a programming abstraction that separates mobility from the functional aspects of a mobile computation. We conclude the paper with a mobile agent application that demonstrates the utility of the separation of mobility.

2 Mobile agents

The traditional model of mobile agents has a seductive simplicity. It also lends itself to thinking about

*Department of Computer Science, University of Tromsø, Tromsø, Norway. This work was supported by NSF (Norway) grant No. 112578/431 and 126107/431 (Norges Forskningsråd, DITS program).

†Department of Computer Science and Engineering, University of California San Diego, La Jolla 92093-0114, California, USA. In doing this work, Marzullo was supported by NSF (Norway) grant No. 112578/431 (DITS program).

programming mobile agents as *itinerant agents* that control their own destiny as they execute and wander around the network. Our experience with implementing mobile agent platforms and applications, though, led us to conclude that as a programming abstraction it is not as appealing as it first appeared to be.

2.1 Traditional Model

There are several problems that arise when directly using the traditional mobile agent model as a programming abstraction:

Explicit mobility. Most mobile agent systems provide some primitive that allows an agent to move to some given destination and resume execution. The resulting programs often have a structure that resembles programming sequential programs with `goto` statements. It is well-known that `goto`'s obfuscates the structure of program code [1]; a similar obfuscation occurs with mobile agents that explicitly control their itinerary based on their program state. And, we have not yet seen a realistic example of an algorithm where policies regarding location are a natural part of the algorithm itself. Until such an algorithm appears we are better off keeping location policies separate to simplify distributed application development.

Implicit state-capturing. Implementing mobile agents following the traditional model suggests an abstraction in which the agent's state is transparently captured and migrated. However, state capture is complex and expensive whether it is done automatically or semi-automatically. Specialized logic always has the potential of making better decisions regarding which parts of the state to capture.

Implicit state capture is a problem that predates mobile agents. For example, migration at the level of processes is rarely supported. This is, in part, due to the overhead of automatic state-capturing. Indeed, one successful implementation of process migration, Condor, restricts what a process can do to simplify the problem. [11]

Single programming language trap. At first glance, Java seems custom-made for writing mobile agents: programs written in Java are portable, and there are strong commercial incentives for vendors to put JVM on all possible computing platforms ranging from multiprocessors to PDAs. But, Java is not the best programming language for all applications. Often, one cobbles distributed applications together from many different programs, including legacy code, scripts, and specialized utilities. Hence, it is beneficial to have the support for mobile agents not be too strong tied to a single programming language. An early design decision of TACOMA was to provide mobile language support as an operating system add-on instead of adding it to the runtime system of a programming language. We now separate mobility from the functional aspects of a mobile agent, and impose a single language only to program the mobility aspect of the mobile agent.

Even when restricted only to mobility, our experience with Java has encouraged us to consider using other languages. For example, Java has an unnecessarily complex native interface, it lacks some vital support for asynchronous network communication, and it is relatively low-level and verbose. Furthermore, we have experienced both backwards- and inter-platform incompatibility problems as well as inconsistent semantics in the way Java threads behave on different platforms. For these and other reasons, we have changed over to using Python.

2.2 Other Issues

The programming abstraction is not the only problem that has hampered the widespread adoption of mobile agents. Host integrity, for example, is another important problem. Accepting and allowing anonymous program code from anonymous sources to execute within their administration domain is not an option for most system administrators or PC owners. But, in the domain that we have found mobile agents most useful—systems configuration and deployment, system and service extensibility, and distributed application self-management—the problem with the programming abstraction is more critical.

2.3 Factored Mobile Agents

One of the observations we have made in building many different mobile agent applications is that distributed applications have three distinct aspects. We call them the *function*, the *mobility*, and the *management* aspects. The function aspect represents the parts that contributes to the mission of the application. This aspect is typically not location aware, in the sense that the location at which it executes does not affect the results. Legacy and COTS software constitute a significant portion of the function of many real-world distributed applications. The mobility aspect of the application implements the mechanisms and structures necessary for deploying the function where it is most appropriate. This includes

the installation of software components as it is performed in traditional distributed environments. The management aspect manages both function and mobility at a higher level. More specifically, it implements policies for *when*, *where* and *how* to execute the function. Examples of management policies of applications include fault-tolerance, server cloning to accommodate increased demand, and invoking security countermeasures in response to intrusion detection alarms.

In contrast to the traditional model, the *factored model* of mobile agents separates these three aspects. Mobile agent applications that support distributed system management match the factored model well. Given our interest in distributed system management, we have redesigned the TACOMA platform to directly support the factored model. We describe this platform, called νTOS, next.

3 Supporting Factored Agents

The factored model defines the three aspects of function, mobility, and management. A mobile agent platform that is based on the factored model needs to at least support the mobility aspect. The function, however, can be provided outside of the mobile agent platform. The management aspect requires both the mobility- and the function aspects. For example, a management policy on reconfiguration would needs abstractions to cause the reconfiguration, and the policy might base its decision to reconfigure on the current workload of the application. For purposes of simplicity and separation of concerns, the νTOS platform only directly supports mobility. Thus, the platform resides at a level in between traditional full-fledged mobile agent systems and remote execution facilities provided by, for instance, `ssh`, `rsh` and `rexec`.

νTOS is the latest in a series of TACOMA mobile code platforms developed with distributed systems management in mind. We have developed the platform and applications in an iterative manner to purify the set of provided abstractions. This branch of the TACOMA project started in 1997 with TACOMA v1.2 [7], a full-fledged and general mobile agent platform. Four major iterations, resulting in versions developed for Win32 and Java/JVM [10] have led us to our current version, which, as the name suggests, is rather small: it consists of 90 lines of Python code. Despite its diminutive size, it supports an itinerant style of code mobility and encrypted network communication. Encryption is based on symmetric keys and is used both for carrier transfer and for communication at application level.

Making the system as small as possible has been a goal in itself of several reasons. For example, larger software systems tend to hide details which is vital when it comes to determining the level of trust to put in the system.

The νTOS equivalent of a mobile agent is a *carrier*. A carrier is structured as a Python dictionary (that is, an associative array) with three key/value pairs:

1. *code* that gives the code of the carrier. This is the code that comprises what we call the management aspect of the application. The value of code is a Python program represented as a string of either Python source byte code.

2. *data* that gives the data the carrier references. The value of data corresponds with the state of a process-like mobile agent and the briefcase of a TACOMA mobile agent.

3. *path* that gives the itinerary of the carrier. The value of path is a sequence of host names or IP addresses in dot notation.

A carrier is launched with the `tos.forward` function.

The following example illustrates a trivial νTOS carrier that implements the well-known "hello, world" example. Lines 2–4 of Figure 1 define the

```
(1)   import tos
(2)   ca = {'code':'print data',
(3)         'data':'hello, world',
(4)         'path':['host0', 'host1', 'host2']}
(5)   tos.forward(ca)
```

Figure 1: The "hello, world" example.

carrier. First, `code` is simply a string that contains a Python program that prints a string. It could instead be a Python statement that evaluates to a string, perhaps by opening and reading a file containing the program. Second, `data` holds the string to be printed. Finally, `path` is a list of hosts that the carrier will visit in order. A carrier moves to the host at the head of this list when the execution of `code` terminates successfully and `path` has at least one element left. The value of `path` for the carrier at the new host will have this host removed. Finally, line 5 causes the launch of the carrier. Note that all the three components of a carrier can be updated after the carrier has been launched, either by the carrier itself or by some foreign service.

Carriers are executed by νTOS kernels. One starts a kernel by executing the Python module `tos`. Or, one can run a Python program that imports the `tos`

module and calls the function `tos.kernel()`. The kernel reads the file `.tosrc` from the working directory to obtain the port over which carriers are sent and the symmetric key that is used to encrypt network communications.

The security of νTOS is based on the security mechanisms of the host operating system, the permissions of the Python interpreter process, and the symmetric key stored in `.tosrc`. The protection this affords has proven to be sufficient for the system management applications that we have developed with νTOS.

νTOS does not suffer from the problems listed in Section 2.1. The structuring problem associated with explicit mobility does not arise because the mobility aspect has been separated. For example, in distributed systems management one often wishes to distribute code with strong guarantees similar to reliable broadcast or two-phase commit. Programming such a distribution protocol using the traditional mobile agent model is hard because the function of the mobile agent is merged with the protocol. With νTOS, the protocol is implemented with a carrier, which can even be automatically generated by a tool. And, νTOS, like all versions of TACOMA, does not provide implicit state transfer. Hence, the function aspect of a mobile agent can easily be implemented using whatever programming language.

4 A Factored Application

To illustrate the benefit of separation of mobility and management from function, we give an example distributed image rendering application based on ray-tracing. It is an example of the kind of embarrassing-parallel applications that are well suited for execution on the Computational Grid [3]. An image is rendered from a specification by having sub-jobs compute tiles of the image. The rendering proceeds by having sub-jobs query a schedule for new tiles to compute.

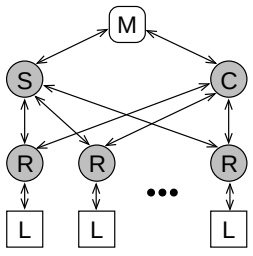

Figure 2: νTOS based rendering application.

Figure 2 shows the layout of the application. Shaded circles represent νTOS carriers. Three types of carriers are used in the application. First, the *Renderer* (R) render tiles of the larger image by using a standard ray-tracing software package. Second, the *scheduler* (S) schedules the set of sub-jobs among the renderers. Finally, the *Collector* (C) collects results from the renderers. White rectangles (L) represent the ray-tracing package we use (PovRay 3.1, available from `www.povray.org`). The topmost rounded rectangle (M) represents the client software from which the computation is initiated and the execution is monitored. The client software uses utilities from the ImageMagick package (available from `www.imagemagick.org`).

In this application we use carriers for distributing the software, for managing computations at individual computers and for client/server communication for the duration of a single rendering computation. The application experiences close to linear speedup with the addition of L components. It appears that using carriers as mediators does not decrease performance significantly.

Evaluating system simplicity is rarely easy to do. Nevertheless, we found this application simple to build. We did not have to write much code: the number of lines of Python code (each less than 80 characters) for each component are approximately 65 for the scheduler, 50 for the renderer, 40 for the collector, and 20 for the client. The development time from scratch to a working system was about one day. One important reason for this astonishingly short development time is that there existed legacy software for all functional purposes like rendering, image cropping and mounting. That is, the short development time arose by separating function from mobility and management, and using COTS software to provide most of the functional aspect.

The application also lends itself to a structure that we call *complexity on demand*. By having management separated from function, it is straightforward to design the management to be adaptive based on the configuration and on the performance of the environment. For example, if the application were to run on a wide-area network, it would be straightforward to have the management supply fault tolerant scheduling against network partitions. A benefit of the factored model is that such an increase in complexity can often be contained within the management aspect.

5 Conclusion

We have argued that the traditional mobile agent abstraction is a poor structure, and have presented a new *factored* abstraction. This new abstraction separates the three aspects of *function*, *mobility* and *management* from each other. We described the νTOS infrastructure that supports the factored model of mobile agents and that is well suited for distributed systems management applications. We are currently using νTOS to build a Grid Computing environment called *Open Grid* that supports the assembly of open, extensible and complete computational grids for embarrassingly parallel applications.

Acknowledgements

We would like to thank the other members of the TACOMA team. We also thank the members of the UCSD AppLeS team, in particular Henri Casanova, Walfredo Cirne, and Graziano Obertelli.

References

[1] E. Dijkstra. Goto considered harmful. *Communications of the ACM*, 11(3):147–148, Mar 1968.

[2] D.Kotz, R.Gray, and S.Nog. Agent tcl: Targeting the needs of mobile computers. *IEEE Internet Computing*, pages 58–67, 1997.

[3] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Jul 1998.

[4] Dag Johansen. Mobile Agent Applicability. *Journal of Personal Technologies*, 2(2), 1999.

[5] Dag Johansen, Keith Marzullo, and Kåre J. Lauvset. An Approach towards an Agent Computing Environment. In *The Workshop on Middleware at the 19th IEEE International Conference on Distributed Computing Systems*, Austin, Texas, May 1999.

[6] Dag Johansen, Keith Marzullo, Fred B. Schneider, Kjetil Jacobsen, and Dmitrii Zagorodnov. NAP: Practical Fault-Tolerance for Itinerant Computations. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, Austin, Texas, May 1999.

[7] Dag Johansen, Nils P. Sudmann, and Robbert van Renesse. Performance Issues in TACOMA. In *Proceedings of the 3rd Workshop on Mobile Object Systems, 11th Europeean Conference on Object-Oriented Programming*, Jyväskylä, Finland, Jun 1997.

[8] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. Operating system support for mobile agents. In *Proceedings of the 5th. IEEE Workshop on Hot Topics in Operating Systems*, pages 42–45, Orcas Island, Wa, USA, May 1995.

[9] Danny B. Lange and Mitsuru Oshima. Mobile Agents with Java: The Aglet API. In Dejan Milojicic, Frederick Douglis, and Richard Wheeler, editors, *Mobility, Mobile Agents and Process Migration - An edited Collection*, pages 495–512. Addison Wesley, 1999.

[10] Kåre J. Lauvset, Dag Johansen, and Keith Marzullo. TOS: Kernel Support for Distributed Systems Management. In *Proceedings of the 16th ACM Symposium on Applied Computing (SAC 2001)*, Las Vegas, Nevada, Mar 2001.

[11] M. Livny M. Litzkow and M.W. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104–111, Jun 1988.

[12] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1), Feb 2000.

[13] John F. Shoch and Jon A. Hupp. The "Worm" Programs - Early Experience with a Distributed Computation. *Communications of the ACM*, 25(3):172–180, Mar 1982.

[14] Nils P. Sudmann and Dag Johansen. Adding Mobility to Non-Mobile Web Robots. In *Proceedings of the workshop on Knowledge Discovery and Data Mining in the World-Wide Web at the 20th IEEE International Conference on Distributed Computing Systems*, Taipei, Taiwan, Apr 2000.

[15] T.Walsh, N.Paciorek, and D.Wong. Security and reliability in concordia. *Proceedings of the 31st Hawaii International Conference on Systems Sciences*, pages 44–53, 1998.

[16] J.E. White. Telescript technology: Mobile agents. *General Magic White Paper, Appeared in Bradshaw, J., Software Agents AAAI/MIT press*, 1996.