A Tacoma Retrospective

Dag Johansen,* Kåre J. Lauvset,* Robbert van Renesse,†
Fred B. Schneider,† Nils P. Sudmann,* and Kjetil Jacobsen*

November 30, 2001

Abstract

For seven years, the Tacoma project has investigated the design and implementation of software support for mobile agents. A series of prototypes has been developed, with experiences in distributed applications driving the effort. This paper describes the evolution of these Tacoma prototypes, what primitives each supports, and how the primitives are used in building distributed applications.

1 Introduction

In the Tacoma project, our primary mode of investigation has been to build prototypes, use them to construct applications, reflect on the experience, and then move on. None of the systems we built were production-quality

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

†Department of Computer Science, Cornell University, Ithaca, New York 14853. Supported in part by ARPA/RADC grant F30602-96-1-0317, AFOSR grant F49620-00-1-0198, Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory Air Force Material Command USAF under agreement number F30602-99-1-0533, National Science Foundation Grant 9703470, and a grant from Intel Corporation. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

nor were they intended to be. The dramatic collapse of our namesake[1]—the Tacoma Narrows bridge—played an important role in the evolution of suspension-bridge design [Pet92] by teaching the importance of certain forms of dynamic analysis. It was in that spirit that we set out with the TACOMA project to build artifacts, stress them, and learn from their collapse.

Each version of TACOMA has provided a framework to support the execution of programs, called *agents*, that migrate from host to host in a computer network. Giving an agent explicit control over where it executes is attractive for a variety of technical reasons:

- Agent-based applications can make more efficient use of available communication bandwidth. An agent can move to a processor where data is stored, scanning or otherwise digesting the data locally, and then move on, carrying with it only some relevant subset of what it has read. By moving the computation to the data—rather than moving the data to the computation—significant bandwidth savings may result.

- Because agents invoke operations on servers locally (hence cheaply), it becomes sensible for servers to provide low-level RISC-style APIs. An agent can synthesize from such an API operations specifically tailored to its task. Contrast this with traditional client-server distributed computing where, because communications cost must be amortized over each server operation invocation, servers provide general-purpose high-level APIs.

- Agents execute autonomously and do not require continuous connectivity to all servers where they execute. This makes the agent abstraction ideal for settings in which network connections are intermittent, such as wireless and other forms of ad hoc networks.

Agent-based systems also provide an attractive architecture for upgrading the functionality of fielded software systems. Software producers rightly consider extensibility to be crucial for preserving their market share. It is not unusual for a web browser to support downloading of "helper applications" that enable new types of data to be presented. And much PC software is installed and upgraded by downloading files over the Internet. The logical next step is a system architecture where performing an upgrade does not require overt action by the user. Agents can support that architecture.

[1]The name TACOMA is an acronym for <u>T</u>romsø <u>A</u>nd <u>Co</u>rnell <u>M</u>oving <u>A</u>gents.

Version	*Innovation*
Tacoma v1.0	basic agent support
Tacoma v1.2	multiple language support and remote creation
Tacoma v2.0	rich synchronization between running agents
Tacoma v2.1	agent wrappers
Tacoma lite	small footprint for PDAs
TOS	factoring mobility and data transformation

Table 1: The most important versions of the Tacoma platform.

Engineering and marketing justifications aside, agents raise intriguing scientific questions. An agent is an instance of a process, but with the identity of the processor on which execution occurs made explicit. It might, at first, seem that this change to the process abstraction is largely cosmetic. But with processor identities explicit, certain communication becomes implicit—rather than sending messages to access or change information at another site, an agent can visit that site and execute on its own behalf. The computational model is then altered in fundamental and scientifically interesting ways. For example, coordinating replicas of agents—processes that move from site to site—in order to implement fault-tolerance requires solving problems that do not arise when stationary processes are what is being replicated. Other new challenges arise from the sharing and interaction that agents enable, as mobile code admits attacks by hosts as well as to hosts.

In order to understand some of the scientific and engineering research questions raised by the new programming model, we experimented with a series of Tacoma prototypes. Each prototype provided a means for agents to migrate, and each provided primitives for agents to interact with one another. The most important versions of Tacoma are listed in Table 1. Version 1.0 provided basic support for agents written in the Tcl language. Version 1.2 added support for multiple languages and agent creation on remote hosts. In version 2.0, new synchronization mechanisms were added. Agent wrappers, the innovation in version 2.1, facilitated modular construction of agents. To enable support for agents on small PDA devices, we built Tacoma lite with a small footprint. And TOS, the most recent version of Tacoma, allows the mobility and data transformation aspects of agents to be cleanly separated.

A recurring theme in our work has been to consider Tacoma as a form of glue for composing programs, rather than as providing a full-fledged computing environment for writing programs from scratch. And almost from the start, TACOMA avoided designing or prescribing a language for programming agents. This language-independence allows an agent to be written in the language that best suits the task at hand. It also allows applications to be constructed from multiple interacting agents, each written in a different language. But, as will be clear, our goal of language-independent support for program migration has had broad consequences on the abstractions TACOMA supports.

The rest of this paper is organized as follows. Section 2 describes some of the primitives we experimented with in the various TACOMA versions. A novel approach to structuring agents—based on wrappers—is the subject of section 3. Section 4 explores the need for an agent integrity solution and describes our approaches. Section 5 presents our experiences in connection with building a mobile agent application. Agent support for PDAs and cell phones is discussed in Section 6. Section 7 describes recent developments in TACOMA, and section 8 contains some conclusions.

2 TACOMA Primitives

2.1 Abstractions for Storing State

In moving from one host to the next, any agent whose future actions depend on its past execution must be accompanied by state. With *strong mobility*, state capture is automatic and complete—not unlike what is done in operating systems that support process migration [Zay87, TLC85, PM83] where the system extracts the state of a process, moves it to another processor, and then starts the process running there. Determining which state components to extract for a given process has proved tricky and expensive for process migration in the presence of run-time stacks, caches, and various stateful libraries; reincarnating the state of a process on a machine architecture that differs from the one on which the process was running is also known to be a difficult problem. Nevertheless, strong mobility is supported by Telescript [Whi94], Agent-Tcl [Gra95], and Ara [PS97]. The convenience to application programmers is a strong attraction.

With *weak mobility*, the programmer of an agent must identify and write code to collect whatever state will be migrated. Java provides an object serialization facility for extracting the state of a Java object and translating it into a representation suitable for transmission from one host to another. So it is not surprising that weak mobility is what many Java-based agent systems (Aglets [LO98], Mole [BHRS98], and Voyager by ObjectSpace) adopt—especially since capturing the execution state of a Java thread without modifying the JVM is impossible, as shown by Sumatra [ARS97]. Note, however, that Java's object serialization mechanism incorporates the entire object tree rooted on a single object. Unless the agent programmer is careful to design data structures that avoid certain links, the high costs associated with strong mobility can be incurred.

TACOMA supports weak mobility. This decision derives from our goals of having run-time costs be under programmer control and of providing language-independent support for agents. Only the agent programmer understands what information is actually needed for the agent's future execution; presumably the agent programmer also understands how that information is being stored. So, in TACOMA, the agent's programmer is responsible for building routines to collect and package the state needed by an agent when it migrates.

Folders, Briefcases, and File Cabinets. The state of a TACOMA agent is represented and migrated in its *briefcase*. Each agent is associated with, and has access to, one briefcase. The briefcase itself comprises a set of *folders*, named by ASCII strings unique to their briefcase. In turn, folders are structured as ordered lists of byte strings called *folder elements*.

Various functions are provided by TACOMA to manipulate these data structures. There are operations to create folders, delete folders, as well as to add or remove elements from folders. For transport and storage, the TACOMA archive and unarchive operations serialize and restore a briefcase. Because each folder is an ordered list, it can be treated as either a stack or a queue. As a queue, we find folders particularly useful for implementing FIFO lists of tasks; as a stack, they are useful for saving state to backtrack over a trajectory and return to the agent's source.

Not only must state accompany an agent that migrates, but it may be equally important that some state remain behind.

- It is unnecessary and perhaps even costly to migrate data that is used only when a given site is visited.
- Secure data is best not moved to untrusted sites, so such data may have to be saved temporarily at intermedicate locations.

- Having site-local state allows agents that visit a site to communicate—even if they are never co-resident at that site.

TACOMA therefore provides a *file cabinet* abstraction to store collections of folders at a specific site.

Each site may maintain multiple file cabinets, and agents can create new file cabinets as needed at any site they visit. Every file cabinet at a site is named by a unique ASCII string. By choosing a large, random name, an agent can create a secret file cabinet because guessing such a name will be hard and, therefore, only those agents that have been told the name of the file cabinet will be able to access its contents. File cabinets having descriptive or well-publicized names are well suited for sharing information between agents.

Whereas a briefcase is accessed by a single agent, file cabinets can be accessed by multiple agents. To support such concurrent access, an agent specifies when opening a file cabinet whether updates should be applied immediately or updates should be applied atomically as the file cabinet is closed.

Note that TACOMA's file cabinet and briefcase abstractions do not preclude using TACOMA to support an agent programming language providing strong mobility. The run-time for such a language would employ one or more folders for storing the program's state. Our experience so far, however, has been that programmers in most any language have no difficulty working directly with folders, briefcases, and file cabinets as a storage abstraction—it has not been necessary to hide these structures behind other abstractions. And Java's object serialization has been used for TACOMA agents written in Java, just as Python's associative arrays have been used for TACOMA agents written in the Python language to access the contents of briefcases and folders.

For historical reasons, TACOMA folders store and are named by ASCII strings. Had XML or KQML [FFMM94] been in widespread use when our project started, then we would probably have selected one of these representation approaches. In fact, any machine- and language-independent data representation format suffices.

2.2 Primitives for Agent Communication

In TACOMA, agents communicate with each other using briefcases. The TACOMA `meet` primitive supports such inter-agent communication by allowing one agent to pass a briefcase to another. Its operational semantics evolved as we gained experience writing agent applications, with functionality added only when a clear need had been demonstrated.

The meet in TACOMA v1.2 initially was similar to a local, cross address-space, procedure call. Execution of

meet A' with bc

by an agent A caused another agent A' to be started at the same site with a copy of briefcase bc provided by (and shared with) A. A blocked until A' executed finish to terminate execution. Arguments were passed to A' in briefcase folders; results were returned to A in that same briefcase.

This first version of meet was soon extended to allow communication between agents at different sites. In addition, a means was provided for an invoking agent to continue executing in parallel with the agent it invoked. Execution of

meet $A'@host$ with bc block

by an agent A caused execution of A to suspend while another agent A' executes to completion at site *host* and with a copy of briefcase *bc*. Were the block keyword omitted, execution of A would proceed in parallel with A'.

With TACOMA v2.0, the non-blocking variation of meet was replaced with two new primitives: activate and await.

- Execution of await by an agent A' blocks A' until some other agent names A' in a meet or activate. An agent name A could be specified in the await to cause A' to block until it is A that executes the corresponding meet or activate.

- Execution by an agent A of meet A' or activate A' first checks to see if there is an agent A' blocked at an await that can be activated and restarts that. If there isn't, a new instance of A' is created and executes concurrently with A.

The meet, await, and activate primitives can be used by agent programmers to implement a broad range of synchronization functionality, including an Ada-style rendezvous operation. We eschewed building direct support for an rendezvous because such high-level constructs too often are, on the one hand, expensive and, on the other hand, only a crude approximation for what is really needed by any specific application. Lower-level primitives, like activate and await, that can be composed, do not suffer these difficulties. And being equivalent to co-routines, they should be adequately powerful. Our one foray in the direction of high-level synchronization constructs was a *waiting room* abstraction, which enabled agents to store their state (briefcases) and suspend execution. Application programmers found the mechanism costly and cumbersome to use.

Service Agents and Other Implementation Details. The TACOMA run-time at each site makes various services available to agents executing at that site in the form of *service agents* in much the same way as Ara [PS97] and Agent-Tcl [Gra95] do. An agent A obtains service by executing a meet that names the appropriate service agent. In TACOMA v1.0, for example, each site provided a *taxi agent* to migrate an agent A to a site named in a well-known folder in A's briefcase.

The extended functionality of meet in TACOMA v1.2 obsoleted taxi agents. But our goal of supporting multiple languages led us to augment the set of service agents with the new class of *virtual machine service agents.* Each host ran a virtual machine service agent for each programming language that could be executed on that host; that virtual machine service agent would execute any code it found in the xCODE folder. So, for example, an agent A would migrate to site host and execute a Java program P there by storing P in the JAVACODE folder and executing a meet naming the virtual machine service agent for Java, JVM@host.

The VM_BIN virtual machine service agent executes native binary executables. We allow for heterogeneity in machine architectures by associating a different briefcase folder with each different type of machine. VM_BIN identifies the folder that corresponds to the current machine, extracts a binary from it, and runs the result.

For obvious security reasons, virtual machine service agents must guarantee that agents they execute only interact with the underlying operating system and remainder of the site's environment through TACOMA primitives they provide. Service agents, however, do have broader access to their environment—they are a form of trusted processes.

In addition, TACOMA allows agents to be accompanied by digital certificates (and stored in the xCODE-SIG folder). These certificates are interpreted by the service agents to define accesses permitted by the signed code. The current version of the system gives any agent accompanied by a certificate

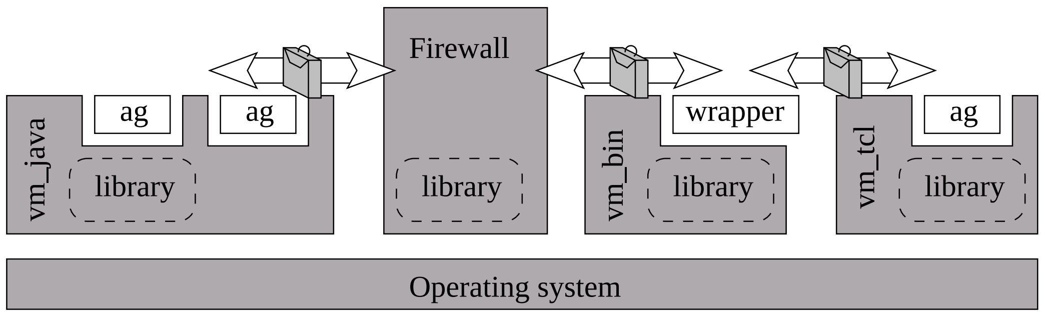

Figure 1: The architecture of TACOMA v2.1.

complete and unrestricted access to the environment for its signed code; in any serious realization, we would associate types of access with particular signers, and we would also allow a signer to specify in the certificate additional restrictions on access.

Figure 1 illustrates the overall architecture of more recent (i.e., post v2.1) TACOMA systems. Each host runs a collection of virtual machine service agents that are responsible for executing agents and that contain a library with routines for agent synchronization and communication as well as for briefcase, file cabinet, and folder manipulation. There is also a *firewall* process used to

- coordinate local meetings and
- send messages to firewalls at other sites in order to migrate an agent from one site to another.

Thus, meet and activate operations are forwarded to the local firewall for handling.

3 Wrappers for Structuring Agents

A TACOMA wrapper intercepts the operations performed by an agent and either redirects them or performs pre- and/or post-processing. The effect is similar to stackable protocol layers as seen, for example, in Ensemble [vRBH+98]. The wrapper itself is a TACOMA agent. Redirection is performed by the TACOMA run-time—specifically the firewall and any virtual machine service agents responsible for interpreting agent code. To create a wrapper, the appropriate virtual machine service agent is contacted using meet and with a briefcase whose folders detail operations (i.e. meet executions) to intercept and give code to run when those operations are intercepted. The wrapped agent and its wrapper must be executed on the same host, but they may use different virtual machine service agents and, therefore, they may be written with different programming languages.

A wrapped agent can be wrapped again, creating an onion-like structure. From the outside, the onion appears to move and execute as a monolithic unit; from TACOMA's perspective, each wrapper is itself a separate agent. Different wrappers thus could execute in their own security domains. And, as a corollary, a wrapper could serve as a trusted process, accessing functionality that the agent it wraps cannot.

We have to date experimented with three wrappers:

A Remote Debugger Wrapper. The TACOMA remote debugger intercepts all operations going to and coming from the agent it wraps. When operating in the *passive mode*, a notification is sent (using activate) to a remote monitor before passing each operation on, unmodified; when operating in the *active mode*, the remote debugger performs a meet with a specified controller (which can change the briefcase before passing on the operation). Our experience with remote debugger is quite positive—after it became operational, the need for explicit remote-debugging support in the TACOMA run-time system disappeared.

A Reference Monitor Wrapper. The wr_codeauth wrapper is a form of reference monitor. It is wrapped around untrusted service agents and imposes authorization checks to preserve the integrity of the TACOMA run-time environment. By implementing this security functionality using a wrapper, we avoided having to modify each of the service agents individually. It is also wr_codeauth that checks digital signatures and rejects operations not allowed by accompanying certificates.

A Legacy Migration Wrapper. The TACOMA Webbot wrapper was developed to change a legacy Web crawler into a mobile agent (see section 5). This wrapper moves crawler code from site to site.

Wrappers currently under development include one to provide fault-tolerance using the NAP protocols (see section 4) and one to implement multicast communication.

Wrappers were added to TACOMA (in v2.1) [SJ00] because we found our agents for applications becoming large and unwieldy from code to support functionality that might well have been included in the TACOMA run-time itself but hadn't been. Adding code to the run-time would have worked for our experimental set-up but clearly would not scale-up to large deployments, where we had no control over when and whether upgrades would be applied to the TACOMA run-time. Wrappers, then, were conceived as a means to provide extensibility to the base system.

Implementation of Wrappers. In a wrapped agent, it is helpful to distinguish the core and inner wrappers from the outer ones. The core and *inner wrappers* move from host to host; the *outer wrappers* do not move, being added by the host as a means to enforce policies and make site-specific functionality available. Typically, the outer wrappers are trusted and thus have enhanced privileges.

A wrapped agent moves from site to site because a meet is issued by one of the agents comprising the core and inner wrappers. This meet is intercepted by each surrounding wrapper, which archives the briefcase in the intercepted meet, stores that in a folder of its own briefcase, and then re-issues that meet (for interception by the wrapper one layer further out). By definition, a meet issued by the inner-most of the outer wrappers is not intercepted by another wrapper. Such a meet is thus handled by TACOMA's run-time system, with the effect that the agent migrates to the specified host. Once there, what is reactivated is actually the outer-most of the inner wrappers. This wrapper, however, will extract a briefcase from its folder and re-instantiate the agent it wrapped. The process continues recursively until the all the inner wrappers and core have been re-started.

4 Agent Integrity

The benefits of easily implemented computations that span multiple hosts must be tempered by the realization that

- computations must be protected from faulty or malicious hosts, the *agent integrity* problem, and
- hosts must be protected from faulty or malicious agents, the *host integrity* problem.

We have concentrated in TACOMA on the agent integrity problem, both because of expertise within the project and because this problem area was being neglected by other researchers. (In comparison, the host integrity problem has attracted considerable interest in the research community).

Replication Approaches

In an open distributed system, agents comprising an application must not only survive malicious failures of the hosts they visit, but they must also be resilient to potentially hostile actions by other hosts. We now turn our attention to fault-tolerance protocols for that setting. Replication and voting enable an application to survive some failures of the hosts it visits. Hosts that are not visited by agents of the application, however, can masquerade and confound a replica-management scheme. Clearly, correctness of an agent computation must remain unaffected by hosts not visited by that computation.

One example we studied extensively is a computation involving an agent that starts at some *source* host and visits a succession of hosts, called *stages*, ultimately delivering its results to a *sink* host (which may be the same as the source). We assumed stages are not *a priori* determined, because (say) dynamic load-leveling is being used to match processors with tasks—only during execution of stage i is stage $i+1$ determined. The difficulties in making such a *pipeline computation* fault-tolerant are illustrative of those associated with more complex agent-computations.

The pipeline computation as just described is not fault-tolerant. Every stage depends on the previous stage, so a single malicious failure could prevent progress or could cause incorrect data to be propagated. Therefore, a first step towards achieving fault-tolerance is to triplicate the host in each stage.[2] Each of the three replicas in stage i takes as its input the majority of the inputs it receives from the nodes comprising stage $i-1$ and sends its output to the three nodes that it determines comprise stage $i+1$.

Even with this replication, the system can tolerate at most one faulty host anywhere in the network. Two faulty hosts—in the pipeline or elsewhere—could claim to be in the last stage and foist a bogus majority value on the sink. These problems are avoided if the sink can detect and ignore such

[2] Assume execution of each stage is deterministic. This assumption can be relaxed somewhat without fundamentally affecting the solution.

masquerading agents, so we might consider passing a *privilege* from the source to the sink. One way to encode the privilege is by using a secret known only to the source and sink. However, then the source cannot simply send a copy of the secret to the hosts comprising the first stage of the pipeline, because if one of these were faulty it could steal the secret and masquerade as the last stage. To avoid this problem, a series of protocol based on an (n, k) threshold schemes [Sha79] have been developed. Details are discussed in [Sch97].

Primary Backup Approaches

Redundant processing is expensive, so the approach to fault-tolerance of the previous subsection may not always be applicable. Furthermore, preserving the necessary consistency between replicas can be done efficiently only within a local-area network. Replication and voting approaches are also unable to tolerate program bugs. Thus, a fault-tolerance method based on failure detection and recovery is often the better choice when agent-based computations must operate beyond a local area network and must employ potentially buggy software.

We developed such a fault-tolerance method and presented it in [JMS$^+$99]. Our method has roots in the well known primary-backup approach, whereby one or more backups are maintained and some backup is promoted to the primary whenever failure of the primary is detected. With our method, the backup processors are implemented by mobile agents called *rear guards*, and a rear guard performs some recovery action and continues the computation after a failure is detected. We call our protocol *NAP*.[3]

The key differences between NAP and the primary-backup approach are:

- Unlike a backup which, in response to a failure, continues executing the program that was running, a recovering rear guard executes recovery code. The recovery code can be identical to the code that was executing when the failure occurred, but it need not be.

- Rear guards are not executed by a single, fixed, set of backups. Instead, rear guards are hosted by recently visited sites. Much of what is novel about NAP stems from the need to orchestrate rear guards as the computation moves from site to site.

[3] *NAP* stands for *Norwegian Army Protocol*. The protocol was motivated by a strategy employed by the first author's Army troop for moving in a hostile territory.

NAP provides fault-tolerance at low cost. The replication needed for fault-tolerance is obtained by leaving some code at hosts the mobile agent visited recently. No additional processors are required, and the recovery that a mobile agent performs in response to a crash is something that can be specified by the programmer.

We have been able to demonstrate that when only a few concurrent failures are possible, the latency of NAP is subsumed by the cost of moving to another host, the most common method of terminating an action by a TACOMA agent. However, NAP cannot be implemented in a system that can experience partitions, because no failure-detector can be implemented in such a system.

5 Web Crawler Application

Web crawlers follow links to web servers and retrieve the data found there for processing at some other server. They have been implemented in a wide variety of languages[4] with Perl and C dominant. By building a hyperlink validation agent from an existing freely available—but stationary—web crawler application program, we hoped to evaluate

- whether moving a TACOMA agent to the data leads to better performance by reducing communication costs and
- whether TACOMA can be used as a form of glue for building agent applications from existing components.

We presumed TACOMA would be available at all sites to be visited by our crawler agent but made no assumptions about the language used to implement the original web crawler or about how that web crawler worked.

We chose Webbot[5] from the W3C organization as the basis for our validation agent. Available as a binary executable for different common machine architectures, Webbot was never intended to serve as part of an agent. For our agent realization, we used TACOMA's VM_BIN virtual machine service agent to execute Webbot binaries on the various different kinds of hosts.

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

[5] `http://www.w3c.org/Robot`.

Since VM_BIN is unsafe, we configured it to run only those binaries accompanied with a certificate signed by some trusted principal—the wr_codeauth wrapper does this. And, finally, we designed a wrapper (called Webbot) to extend the functionality of the W3C stationary Webbot application for execution as an agent. This wrapper

- moves the Webbot binary to a specified set of web servers, one at a time, and
- restricts Webbot to checking only local links, storing in a folder any remote links encountered for checking when that remote site is later being visited.

The relative ease with which the crawler agent was built confirms that TACOMA's primitives can facilitate easy construction of agents from existing components.

To evaluate the performance of our crawler agent, we used a web server at the University of Tromsø with 3600 pages totaling 381 Mbytes of data. Webbot first crawled this server remotely from Cornell University. We then dispatched our crawler agent from Cornell to crawl the web locally in Tromsø. The UNIX program `traceroute` reported 12 hops between the webbot and the web server in the remote case and 2 hops in the local case. And we found that the crawling took 1941 seconds when the Webbot was run remotely from Cornell but took only 474 seconds when the Webbot-based crawler agent was run. Detailed experimental data can be found in [SJ00].

6 TACOMA Support for Thin Clients

The run-time footprint of TACOMA renders the system unsuitable for execution on small portable devices and thin clients, such as PDAs and cellular phones. Also, TACOMA does not provide adequate support for disconnected hosts. Since so many have claimed that agents would be ideal for structuring applications that run on these devices [Whi94, HCK95, GCKR00], we built TACOMA LITE, a version of TACOMA for devices hosting PalmOS and Windows CE [JJ97]. In doing so, we hoped to gain experience structuring distributed applications involving thin clients with mobile agents.

The TACOMA LITE programming model differs from TACOMA in its handling of disconnected hosts. With TACOMA, execution of a meet that names

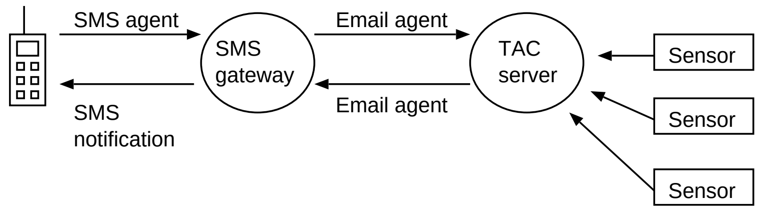

Figure 2: SMS Messages as Mobile Agents.

a disconnected host simply fails; Tacoma lite supports *hostels* for agents trying to migrate to a disconnected site. Agents in a hostel are queued until the destination host connects; they are then forwarded in a manner similar to the Docking System in the D'Agents system [KG99, KGN+97].

The run-time footprint problem is solved in Tacoma lite by using existing functionality on the portable device—email and HTTP support—so that full Tacoma functionality can be located on a larger server elsewhere in the system [JRS96]. The transport mechanism used for sending agents and receiving results on cellular phones is the GSM Short Message Service (SMS), a store-and-forward service like SMTP [RGT]. GSM base stations buffer messages for delivery if the target phone is disconnected. To bridge between the GSM and IP networks, we deployed an SMS to IP gateway available from Telenor.

- SMS messages from cellular phones are received on the gateway processor and converted to email messages. These are sent on the IP network for delivery to Tacoma, where they are converted into a briefcase. A meet is then issued to a service agent designated in the original SMS message.

- An agent can communicate with cellular phone by doing a meet with a service agent. That service agent sends the briefcase to the gateway processor, which then generates a suitable message for display on the cellular phone.

Figure 2 summarizes this infrastructure for handling communication between cellular phones and Tacoma.

SMS messages cannot exceed 160 characters. This means agents constructed by cellular phone users must be terse, and that precludes use of a general-purpose programming language here. Application-specific languages appear to be a viable solution for the time being—at least until the capacity of these small devices grows. The first application we built for a cellular phone was a weather alarm system [JJ99]. For that system, one would write

```
ws gt 20 & t lt 10
```

to request notification if ever the windspeed (ws) is greater than (gt) 20 meters per second and (&) the temperature (t) is less than (lt) 10 degrees. Whenever the predicate specified by the agent evaluates to true, a short notification is sent back to the cellular phone. Obviously, use of application-specific languages limits the class of "agents" that can be written by the user of a cellular phone.

7 Factored Agents: Beyond Wrappers

Over the course of the Tacoma project, it has become clear that developing an agent as if it were a single monolithic object is a bad idea, because it forces the agent programmer to deal unnecessarily with complexity. Adopting wrappers for agents was a start at developing an infrastructure to allow better agent structuring. Recently, we have been exploring a new approach that leverages characteristics intrinsic to all agents and separates three concerns:

Function which deals with data and its transformation.

Mobility which deals with determining sites the agent will visit and mechanisms involved in transferring data and control.

Management which deals with the glue that controls the agent's function and mobility.

This new structuring approach is embodied in the latest Tacoma prototype TOS, which defines a language for programming *carriers*, the management parts of agents.[6]

The design of TOS was an outgrowth of our work in developing a set of applications for distributed management: a generic software management

[6] The name "carrier" was chosen to reflect the intended usage as a carrier of information and software.

platform, a distributed resumé-database search engine [Joh98], a peer-to-peer network computing platform, and a distributed intrusion-detection system [LJM01]. It became clear that these application had much in common but we lacked the structuring tools to make that commonality apparent in the agent's code. So, we are now developing a library of carriers for constructing classes of agent applications.

The largest of these efforts is *OpenGrid*, a platform based on TOS—hence "open" to diverse function, mobility, and management regimes—for running highly parallel computations [FK98]. Highly parallel computations are often structured according to the controller/worker paradigm. With TOS, different carriers can be invoked in OpenGrid to facilitate computations for a variety of network topology and computer configurations. Carriers from the TOS standard library are used to install and configure legacy software on the grid's computers; other carriers, written specifically for OpenGrid, provide the communication infrastructure used by the controllers and workers, as well as providing different levels of fault-tolerance. So programmers of the parallel applications on OpenGrid often are not required to write carriers; they need only implement algorithms for the workers and, if necessary, the controller.

8 Conclusions

Methods for structuring distributed computations seem to fall in and out of fashion: client-server distributed computing, middleware and groupware, mobile agents, and now peer-to-peer computing. No one of these has proved a solution to all distributed application structuring problems. Nor should we expect to find such a magic bullet. Each hides some details at the expense of others. And the same computation could be written using any of them. What differs from one to the other is the ease with which a computation can be expressed and understood. What details are brought to the fore. What details are hidden. What is easy to say and what is hard to say. One size is never going to fit all, and having expectations to the contrary is naive.

Much research in mobile agents has focused on issues concerning mechanisms and abstractions. In the TACOMA project, we took a strong stand on some of these issues and were agnostic on others. We religiously avoided designing a language or offering language-based guarantees, because we wanted agents to serve as a glue for building distributed applications. By choosing a language or a class of languages or a particular representation for data we would have artificially limited the applicability of our experimental prototypes. The generality of our folder and meet mechanisms decouples TACOMA from the choice of language used in writing individual agents. A program in any language can be stored in a folder and moved from host to host. And, any language that a given host supports can be used to program the portion of an agent executed on that host. This generality is particularly useful in using agents for system integration. Existing applications do not have to be rewritten; COTS components can be accommodated.

Similarly, we took the view that agents themselves should be responsible for packaging and transferring their state from site to site—adopting what became known as weak mobility. Awkward as this might seem, the problem of automatically performing state capture is now understood to be quite complex. And by our choice of mechanisms, we managed to avoid confronting that problem. However, in higher-level programming models where state is invisible to the programmer, automatic state capture becomes a necessity. The cost of moving an agent from one processor then cannot be predicted, and designing applications to meet performance goals becomes difficult.

Not all of the work performed under the auspices of the TACOMA project was concerned with new mechanisms or abstractions. Our work in fault-tolerance and, later, in security for systems of agents are examples. A test that we applied here as new approaches were developed was to ask: "What about mobile agents enables this solution?". The answer was often surprising. From the work in fault-tolerance, we learned that if replicas can move, then their votes must be authenticated—a problem that can be ignored in traditional TMR (Triple Modular Redundancy) replication where point-to-point communication channels provide hardware-implemented authentication. Our work in security led to a new enforcement mechanism (called *inlined reference monitors*) for fine-grained access control, but we quickly realized that this mechanism had no real dependence on what was unique about mobile agents. So that investigation changed course and concentrated on implementing the Principle of Least Privilege in broader settings. Not being agent-based, this security work is not discussed in this paper—in a sense, it has outgrown the world of mobile agents.

The lack of benchmark applications has clearly hampered research in the area of mobile agents. The existence of such applications would allow researchers to evaluate choices they make and might settle some of the debates. Applications would also give insight into what problems are important to solve.

Some have taken the dearth of benchmarks as a symptom that mobile agents are a solution in search of a problem. That is one interpretation. But other interpretations are also plausible. It could be that mobile code affords an expressive power that we are not used to exploiting, and a new generation of applications will emerge as we become comfortable with such expressive power. It also could be that applications that simultaneously exploit all of the flexibility of mobile agents are rare, but applications that take advantage of a few dimensions are not rare (but are never considered paradigmatic).

Acknowledgements

The authors would like to express their gratitude to Keith Marzullo and Dmitrii Zagorodnov at the University of San Diego, Yaron Minsky at Cornell University, and many students who have been involved in the TACOMA project at the University of Tromsø over the years. Anonymous reviewers provided many detailed and helpful suggestions on an earlier version of this paper.

References

[ARS97] A. Acharya, M. Ranganathan, and J. Saltz. Sumatra: A Language for Resource-Aware Mobile Programs. In C. Tschudin J. Vitek, editor, *Mobile Object Systems*, Lecture Notes in Computer Science. Springer Verlag, Berlin Heidelberg, Germany, 1997.

[BHRS98] J. Baumann, F. Hohl, K. Rothermel, and M. Straßer. Mole — Concepts of a Mobile Agent System. *World Wide Web*, 1(3):123–137, 1998.

[FFMM94] T. Finin, R. Fritzson, D. McKay, and R. McEntire. Kqml as an agent communication language. In *The Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94)*. ACM Press, November 1994.

[FK98] I. F. and C. Kesselman. *The Grid : Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, 1998.

[GCKR00] R. S. Gray, G. Cybenko, D. Kotz, and D. Rus. Mobile agents: Motivations and state of the art. In Jeffrey Bradshaw, editor, *Handbook of Agent Technology*. AAAI/MIT Press, 2000. Draft available as Technical Report TR2000-365, Department of Computer Science, Dartmouth College.

[Gra95] 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.

[HCK95] C. G. Harrison, D. M. Chess, and A. Kershenbaum. Mobile Agents: Are they a good idea? Technical report, IBM Research Division, IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, March 1995.

[JJ97] K. Jacobsen and D. Johansen. Mobile Software on Mobile Hardware – Experiences with TACOMA on PDAs. Technical Report 97-32, University of Tromsø, Norway, 1997.

[JJ99] K. Jacobsen and D. Johansen. Ubiquitous Devices United: Enabling Distributed Computing Through Mobile Code. In *Proceedings of the 14th ACM Symposium on Applied Computing (ACM SAC'99)*. ACM press, 1999.

[JMS+99] D. Johansen, K. Marzullo, F. B. Schneider, K. Jacobsen, and D. Zagorodnov. NAP: Practical fault-tolerance for itinerant computations. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS'99)*, pages 180–189, Austin, TX, June 1999. IEEE Computer Society.

[Joh98] D. Johansen. Mobile Agent Applicability. In *Proceedings of the Mobile Agents*, Springer-Verlag LNCS series, pages 80–98, Stuttgart, Germany, September 1998.

[JRS96] D. Johansen, R. Van Renesse, and F.B. Schneider. Supporting broad internet access to TACOMA. In *Proceedings of the 7th ACM SIGOPS European Workshop*, pages 55–58, Ireland, September 1996.

[KG99] D. Kotz and R. S. Gray. Mobile code: The future of the Internet. In *Proceedings of the Workshop "Mobile Agents in the Context of Competition and Cooperation (MAC3)" at Autonomous Agents '99*, pages 6–12, May 1999.

[KGN+97] D. Kotz, R. Gray, S. Nog, D. Rus, S. Chawla, and G. Cybenko. Agent Tcl: Targeting the needs of mobile computers. *IEEE Internet Computing*, 1(4):58–67, July/August 1997.

[LJM01] K. J. Lauvset, D. Johansen, and K. Marzullo. TOS: Kernel Support for Distributed Systems Management. In *Proceedings of the 16th Symposium on Applied Computing (ACM SAC '01)*, Las Vegas, Nevada, USA, March 2001. ACM Press.

[LO98] D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets.* Addison-Wesley, 1998. ISBN 0-201-32582-9.

[Pet92] H. Petroski. *To Engineer Is Human: The Role of Failure in Successful Design.* Vintage Books, 1992.

[PM83] M. Powell and B. Miller. Process Migration in DEMOS/MP. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 110–119, October 1983.

[PS97] H. Peine and T. Stolpmann. The Architecture of the Ara Platform for Mobile Agents. In K. Rothermel and R. Popescu-Zeletin, editors, *Proceedings of the First International Workshop on Mobile Agents MA'97*, number 1219 in Lecture Notes in Computer Science. Springer Verlag, April 1997.

[RGT] Gsm 07.05: Short message service (sms) and cell broadcasting service cbs. Available from `http://www.etsi.org/`.

[Sch97] F. B. Schneider. Towards fault-tolerant and secure agentry. In *Proc. 11th International Workshop WDAG '97*, pages 1–14, Saarbrucken, Germany, September 1997.

[Sha79] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, November 1979.

[SJ00] N. P. Sudmann and D. Johansen. Adding Mobility to Non-mobile Web Robots. In *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS'00) Workshops*, pages F73–F79, Taiwan, April 2000. IEEE Computer Society.

[TLC85] M. M. Theimer, K. A. Lantz, and D. R. Cheriton. Preemtable remote execution facilities for the V-system. *Proceedings of the 10th ACM Symposium on Operating System Priniples*, December 1985.

[vRBH+98] R. van Renesse, K. P. Birman, M. Hayden, A. Vaysburd, and D. A. Karr. Building Adaptive Systems Using Ensemble. *Software - Practice and Experience*, 28(9):963–979, July 1998.

[Whi94] J. E. White. Telescript technology: The foundation for the electronic marketplace. General Magic white paper, General Magic Inc., CA, 1994.

[Zay87] E. R. Zayas. Attacking the process migration bottleneck. In *Proceedings of the 11th ACM Symposium on Operating System Principles*, pages 13–24, 1987.