REPORT

Computer Science Technical Report: **95-23** **June 1995**

An Introduction to the TACOMA Distributed System

Version 1.0

Dag Johansen[1] Robbert van Renesse[2] Fred B. Schneider[2]

[1] University of Tromsø [2] Cornell University

INSTITUTE OF MATHEMATICAL AND PHYSICAL SCIENCES
Department of Computer Science
University of Tromsø, N-9037 Tromsø, Norway, Telephone +47 77 64 40 41, Telefax +47 77 64 45 80

An Introduction to the TACOMA Distributed System

Version 1.0

TROMSØ
TACOMA
CORNELL

Contents

Chapter 4 The TACOMA API 23

Preface

The TACOMA (Tromsø And COrnell Moving Agents) project is concerned with how to provide operating system support for agent based computing. We have developed a prototype execution platform with a flexible mechanism for agent computations to move and branch out in a networking environment.

Several TACOMA implementations have been completed. This document provides an introduction to TACOMA Version 1.0, a TACOMA version based on UNIX and Tcl-TCP. We present the TACOMA project, the computational model, how to get started, and the basic TACOMA abstractions. More elaborate system services and agent applications not described here are under construction. We plan to include these in later distributions of TACOMA.

User feedback (including error reports) should be reported to `<tacoma@cs.uit.no>`. *Those interested in obtaining more information about TACOMA patches and further distributions, should send an e-mail to the same address.*

TACOMA is a joint project between University of Tromsø and Cornell University. Principal investigators are Dag Johansen (Tromsø), Robbert van Renesse (Cornell) and Fred B. Schneider (Cornell). Students currently involved include Raymond Andreassen, Kjell Irgens, John-Einar Jakobsen, Pål Knudsen, Scott Stoller and Nils Peter Sudmann. Especially Nils Peter has contributed much to the current implementation and documentation of TACOMA Version 1.0.

Those interested in getting started immediately should read chapter 3 briefly. Additional manual pages and concrete examples bundled with the source distribution should be sufficient for writing and executing your first TACOMA agents. We assume the reader is familiar with the very basics of UNIX and Tcl.

Dag Johansen	Robbert van Renesse	Fred B. Schneider
Tromsø, Norway, 24th June 1995	Ithaca, USA 24th June 1995	Ithaca, USA 24th June 1995

Chapter 1

The TACOMA Project

Abstract

We are currently investigating networking agents for its use in large scale distributed computing. This is organized in the TACOMA (Tromsø And COrnell Moving Agents) project. This chapter presents the background of and the framework for the TACOMA project.

1 Introduction

The agent paradigm receives a lot of attention currently. An agent is most commonly a software being acting on behalf of a human user. People are interested in how to specify the behavior of agents, and how agents communicate with each other and with their human masters. Different types of agents exist [Riec 94]. Some are stationary in nature. Others use services found in a networking environment. Such agents can roam the internet to accomplish the tasks that the users are requesting, visiting site after site collecting information and possibly performing actions. Agents may also negotiate deals and/or services on behalf of their users. Some of these networking agents are stationary processes, working from their user's workstation by connecting to remote services (such as ftp or finger) without actually moving about. Other agents will actually travel into the internet, transfering their state from machine to machine. It is this second type of networking agent in which we are particularly interested. For short, we call them *agents* in the rest of this document.

In the TACOMA project[1], we focus on operating system issues in connection with agents. We are interested in identifying and providing the system services this type of agent computing requires.

We are also interested in how to structure agent applications in a networking environment. Our thesis is that an agent should have the ability to change locality during its execution. Nevertheless, general application issues, as found in [Riec 94], is a secondary issue in our work.

1. For more information: URL: **http://dslab3.cs.uit.no:1080/Tacoma/index.html**

2 Agent Motivations

We want to focus on four main motivations for the agent paradigm. First, potential *performance improvement* is an important motivation. To start with, consider a common situation where a client and a server are located on two different nodes. The server will typically manage some local data. To carry out the computation, several network messages will often have to be transferred back and forth between the two parties. This is illustrated in Figure 1.

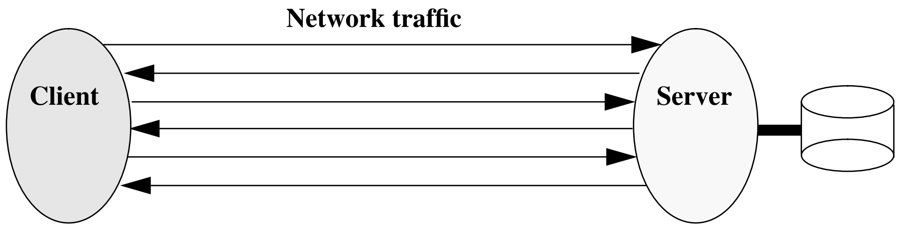

FIGURE 1. A client-server interaction.

We believe there can be performance advantages in moving a client computation to the server side of this connection. This is why cycles are cheap and bandwidth is still a scarse resource. This approach can be efficient for distributed applications that operate on large data structures normally residing at the server side. We also believe that the amount of network interactions can be reduced as a result of co-locating service requesters and service providers heavily engaged in communication. Figure 2 illustrates this, where the client computation moves to the server side of the connection.

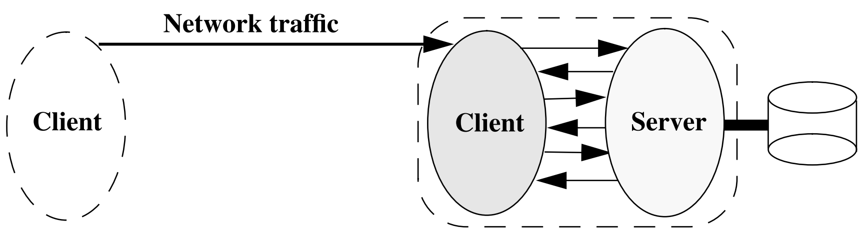

FIGURE 2. Moving the computation close to the data.

A second motivation for the agent model is that it is *intuitive*. Agents acting more or less independently on behalf of somebody is a well-known concept. Co-locating a service requester and a service provider is a real life metaphor well understood. This is an intuitive and compelling approach for negotiation and collaboration purposes between these parties. Figure 3 illustrates a similar situation as in Figure 2, but now the client itself does not move. It sends a representative to the server side to do the actual computation. We call this representative an *agent*. The agent will typically reduce the amount of data that has to be transferred over the network. In this example, the result is finally sent back to the client. This remote filtering of data also reduces the need for voluminous client caching. Also, ownership of data is better ensured by allowing controlled use of it. A service provider does not ship its data away, data that might be its source of income. Instead, controlled, accountable access is allowed.

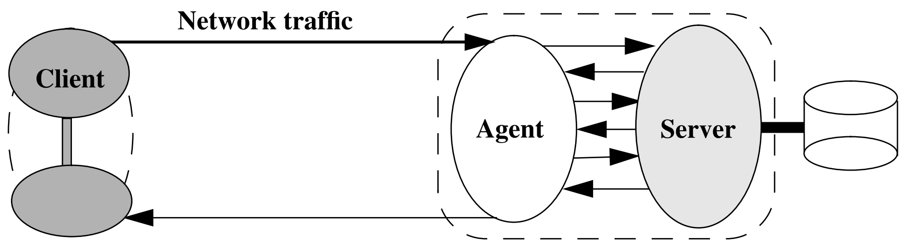

FIGURE 3. Agent sent to the server location.

A third motivation for the agent paradigm is that communication is reduced to a site local issue. The application programmer does not have to deal with complex networking syntax. All that is needed, is to move, or co-locate, the agents that want to communicate. *Locality* of agents are *not hidden*, but communication channels are. Contrast this with location transparency in a more traditional distributed system. As illustrated in Figure 4, an agent `A` that wants to communicate with agent `S`, must be moved to the same site as `S` (or the other way round). The (internet) operating system supports this moving. All that is needed is that agent `A` must ask to be moved.

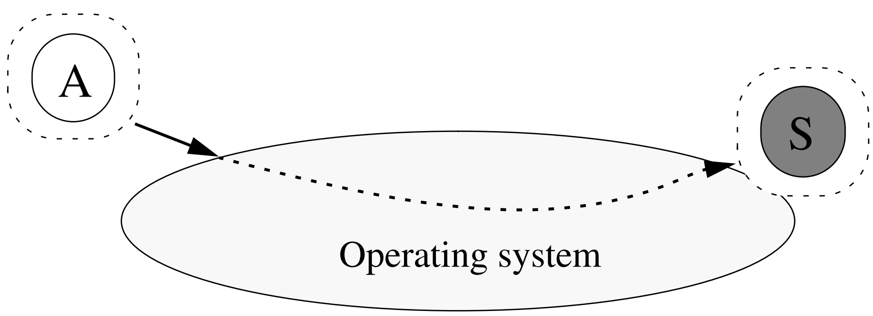

FIGURE 4. Co-locating communicating agents.

Finally, we believe this model gives cleaner and simpler *failure semantics*. In a regular distributed system, the state between the client and the server is normally distributed. A failure half-way through makes it tricky to ensure that both parties roll back their state so that consistency is maintained. This is a much simpler problem when the state is centralized. A co-located client and a server then implies failure atomicity where one does not have to deal with complex external (independent) failure modes.

3 The TACOMA Approach

We have started the TACOMA (Tromsø And COrnell Moving Agents) project to investigate how to support agents in the internet. We are particularily interested in the issues of fault-tolerance, scheduling and management, security, and accounting of agents.

We have constructed a series of TACOMA prototype platforms supporting agents. These implementations feature a flexible mechanism for computations to move and branch out. The first implementation was entirely based on UNIX and a C-based implementation for moving agents around. These agents were written in C and executed on a set of Sun workstations. The mechanism moving the agents around had functionality comparable to existing remote shell facilities as found in UNIX systems. Our approach far outweighted the standard UNIX approach in performance. Nevertheless, by already having a UNIX way of doing this, we decided to use the existing remote shell for the next implementation.

The second TACOMA prototype was more general in nature by using UNIX remote shell (`rsh` on the Sun, `remsh` on the HP workstations) as transport mechanism for moving agents about. Such agents were still written in C, but we also experienced the simplicity and power in using Bourne-shell syntax. Our first agents written were simple in functionality and could be expressed in shell code.

Bourne-shell syntax considered simple is an understatement. Nevertheless, we experienced that interpretive languages seemed to provide sufficient functionality for the type of agents we approached. A third prototype was now implemented in the interpretive language Tcl [Oust 94]. The agents were also implemented in Tcl. Still we used remote shell facilities provided by UNIX, a time-consuming operation for moving the agents about.

A fourth prototype got rid of the UNIX remote shell dependency again. Tcl-TCP, a Tcl extension supporting TCP communication, was used as platform for this TACOMA prototype. We can now transport an agent to another site more efficiently by using our own transfer mechanism based on TCP. This is the version of TACOMA that we are making public available, *TACOMA Version 1*.

Other prototypes of TACOMA are currently under construction. A version based on Tcl-DP has been implemented. Currently, HORUS [Rene 94] provides a UNIX independent platform for a new TACOMA implementation. Focus is on fault-tolerance. This ATM based version is still in a very experimental stage.

4 Scientific Problems

We are studying two main problem areas in the TACOMA project. First, we focus on operating system support for agent based computing. Central problems include:

- Identifying the appropriate *abstractions* a distributed system should provide for agent based computing.
- Providing an *efficient* implementation of these abstractions.
- Providing increased *fault-tolerance* of agents.

- *Management* (monitoring and control) of agents.
- *Security* issues, both securing the agent from the environment and securing the environment from the agent.
- *Accounting* issues, including applicability of electronic cash [Chau 92].

Our second problem area is concerned with applicability of the agent model. We need to identify applications lending itself naturally to this paradigm. As part of this, we need to focus on *how to structure* applications based on the agent paradigm. These applications can then be used to determine what is actually needed from the operating system.

5 Relevant Work

Agent computing is trendy these days. Especially, applicability of this paradigm receives a lot of attention [Riec 94]. A number of AI and World-Wide Web projects focus on agent based computing. Language approaches also receives a lot of attention. This includes:

- General Magic and Telescript [Whit 94].
- Tcl [Oust 94] and derivatives as Safe-Tcl.
- Obliq [Card 95].

6 Work in Progress

TACOMA Version 1 is kept to a bare minimum. It only contains core funtionality to quickly get started building the first internet agents. A more complete version of TACOMA containing more services is not complete for distribution yet. This work in progress (to be distributed upon completion) includes:

- schedulers/broker services.
- firewall agents focusing on security aspects of arriving agents.
- agent management tools.
- active multi-media document support services.
- agent construction support services.
- parallel processing services.
- different agent based distributed applications. This includes re-engineered StormCast services [Joha 93] [Joha 94].

So far, we have used a rear guard agent approach [Joha 95] combined with logging techniques for increased fault-tolerance of TACOMA agents. Fault-tolerance is our current main focus, and a TACOMA version based on HORUS [Rene 94] is under construction. It targets a highly *fault-tolerant* environment for internet agents.

7 Outline

The rest of this document is organized as follows:

- Chapter 2 presents the computational model for networking agents. We will make our model of agents more precise by introducing some well-known daily life concepts.
- Chapter 3 is a step-by-step explanation on how to install and get started with TACOMA.
- Chapter 4 summarizes the basic TACOMA system API. This includes the most vital TACOMA system agents. We present some concrete examples of TACOMA applications and discuss how to structure an agent based application.

Chapter 2

The TACOMA Model and Its Basic Implementation

Abstract

This chapter introduces the TACOMA computational model. The basic system support provided by TACOMA Version 1 is also presented. A mobile computation, an agent, can move around in a networking environment. We need concepts capturing the behavior of the agent itself, but also concepts for the necessary support infrastructure. We attempt to use daily life concepts in this model.

1 Introduction

The client-server approach to distributed computing requires servers to be assigned to nodes statically. A new paradigm is emerging where computations can move around freely. This distributed computing paradigm is based on the idea of computations that act on behalf of others.

This paradigm, which we call networking agents, or just *agents*, is currently being exploited for modernizing electronic mail, shopping, etc. It is also viewed as an appropriate paradigm for negotiations between service providers and service requesters.

The TACOMA project focusses on the agent paradigm for distributed computing. In this section, we will introduce the model of our agent system. Simultaneously, we will introduce our basic implementation of this model.

2 Agents - The Computational Unit

First, we need a computational unit in our system. We use the concept *agent* for any such computation. In chapter 1, we combined client-server concepts and the agent concept. This was done for simplicity reasons. From now on, we make no overall distintion between a client, a server and an agent. In our model, they would all be agents. Some agents never provide any service for others. Other agents are willing to provide services for others. Some agents act on behalf of other agents, while others do not. Some agents always reside at a node, while others can move about.

From a low-level perspective, an agent is just a process, consisting of code and state. A simple agent can be a short Tcl script, or a small C program. At the other extreme, complex AI based agents [Riec 94] can be constructed.

At the same time, we make a distinction between a process in Unix terminology and an agent. A process is a sequence of instructions with some initial state. An agent is also a sequence of instructions and some initial state, but all instructions do not have to be executed on the same site.

Figure 1 illustrates an agent application consisting of 3 separate parts, `A`, `B` and `C`. We will later detail how to structure an agent application. For now, we can view an agent application as a collection of smaller agents. As illustrated, the different parts of the application can be processed throughout a network of nodes. The colored circle indicates active processing, the white circle indicates presence of the agent, but it is not activated at that particular site. Notice that `A` remains back on the first site while the rest of the application (including a copy of `A`) moves about.

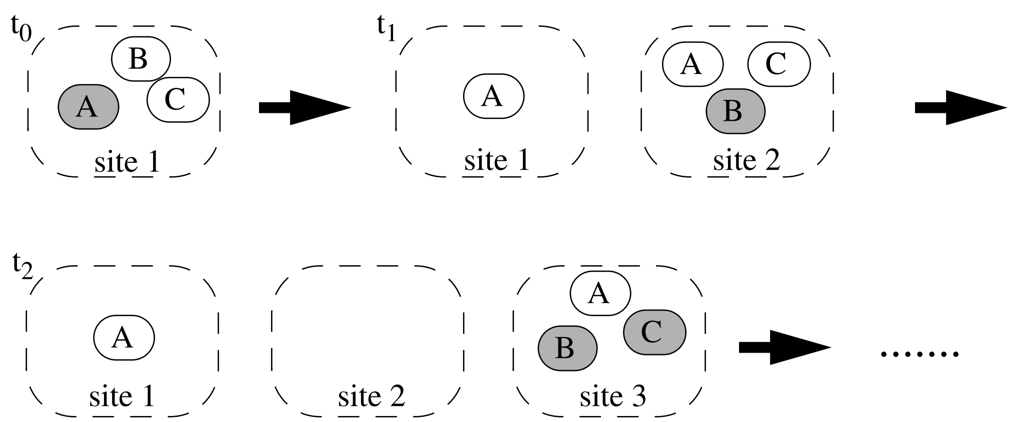

FIGURE 1. An agent application moving about.

This *mobility* is an important characteristic of an agent. As such, it must be possible to have agents that can move about. This sounds like process migration, but is different. The difference lies in *who* decides when to move, the system or the application itself. In process migration, this is normally forced upon a running process by the system. This can be an expensive, complex and, sometimes, even impossible operation.

In the agent model, the agent itself is in charge of deciding *when* to move. Still, as with process migration, once a decision to move has been taken, the operating system must support this.

3 Folders, Briefcases and Cabinets - Maintaining State

Agents must be able to manipulate on data. There is two aspects to this. First, an agent must be able to leave data at a certain site. Such site-local data is typically used the same way that a traditional file system is used by a regular computer program. Second, an agent must be able to carry data along when it moves about. We experienced that an agent could do little unless it could carry state around. Otherwise, its subsequent actions could not be based on its past actions. This is exemplified by an agent sequentially visiting multiple sites. On each site a part of an overall computation is done, and the subresults have to be carried along when the agent leaves the site. We introduced *folders* for this purpose.

We have organized the state into folders, which are units of data accessible by agents. Each folder has an ASCII name. An agent can store or fetch items from this folder. We need folders of a general nature, the same way we need a common, well-defined interface between a client and a basic server.

We have identified several folders of general interest. For instance, a `CODE` folder that contains the source code of an agent, is vital for transferring agents around. Another example is a `DATA` folder containing data that can be associated with the agent in the `CODE` folder. We will later come back with more details about these and other folders.

We found it convenient to group associated folders together. We introduce two new concepts for this. First, we call the collection of folders associated with an agent a *briefcase*. Typically, an agent will carry a briefcase along while moving about. At a certain site, the content of, for instance, the `CODE` folder can be extracted and executed.

Second, stationary folders are needed for permanent data repository purposes. We introduce *file cabinets* for this purpose. The main difference is that file cabinets are stationary while briefcases can be moved. An agent can typically leave data in a file cabinet, or read or remove data from it. Figure 2 illustrates these concepts, where a briefcase is sent between two sites. Site 1 has one local file cabinet, site 2 has two. There are 3 folders in each of the file cabinets.

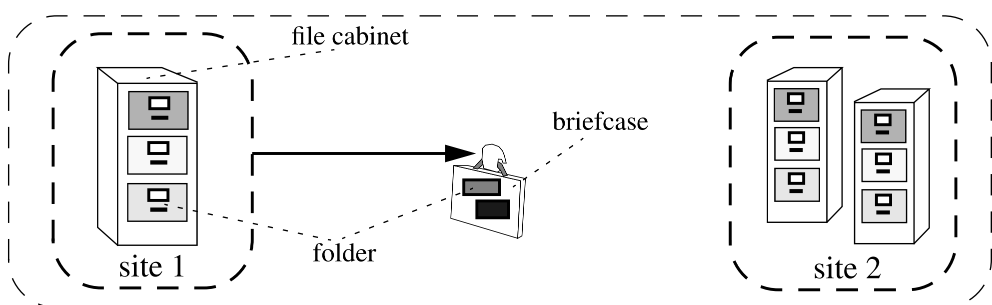

FIGURE 2. Briefcase (with folders) and file cabinets (with folders).

4 Meet - The Basic Abstraction

Agents can *meet*. This is a vital concept used for communication and synchronization between agents. To our surprise, the only abstraction needed to implement the agent model was:

`meet another_agent briefcase`

Agents do not have to communicate by exchanging mail, they simply meet at the same location and exchange a briefcase. For instance, a client agent may request a service from a system agent by passing it a briefcase with a folder containing a service specification. The system agent can return a result folder to the client if necessary.

Currently, we name the agent to be met using an ASCII name. Figure 3 illustrates the meeting between two agents at the same site. Agent `ag_A` now meets with agent `ag_B` by delivering a folder to it. Notice that this meeting actually activates `ag_B`.

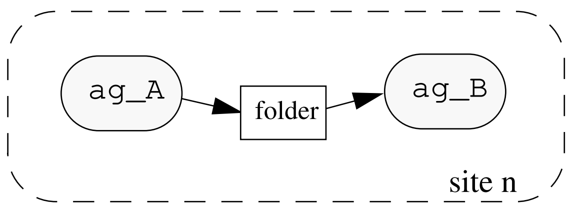

FIGURE 3. Meeting between two agents.

Any communication *channel* is hidden for the programmer. Programming the internet is now done by meeting between the communicating parties. This distinguishes this concept from message based distributed systems. We presume this makes life easier for distributed application programmers.

5 Where to Meet

Agents must be able to *move*. Otherwise, agents located on different machines can not meet. Our notion of agents moving across the internet is agressive. We mean that they physically move from machine to machine.

In earlier TACOMA versions, we had a special transport agent for remote meetings, the `rexec` agent [Joha 95]. An agent that wanted to move first specified where to move in its `HOST` folder. Then, it had to meet locally with its `rexec` agent, which was like a taxi service for it. The `rexec` carried a briefcase. The code of an agent `ag_A` to be activated at the destination site now had to be represented in this briefcase. This is illustrated in Figure 4, where agent `ag_A` already has met with `rexec` and has handed over a briefcase.

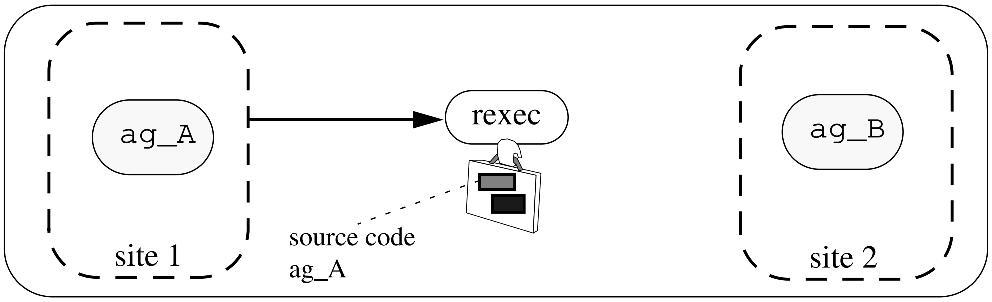

FIGURE 4. Moving an agent as part of a briefcase.

We wanted to make this moving more transparent for the application programmer. First, we have got rid of the `rexec` agent. Its functionality is now implemented by the `meet` abstraction. A remote meeting used to be like this:

`meet rexec briefcase`

The briefcase had a `CONTACT` folder specifying who to meet with at the site specified in the `HOST` folder. This `CONTACT` agent could be, for instance, `ag_tcl` extracting Tcl code from the `CODE` folder and running it. A meet is now directly with this particular agent:

`meet ag_tcl briefcase`

Second, we are attempting to get rid of the manual `HOST` folder operations. A scheduler is currently under construction manipulating on this particular folder. An agent programmer can then leave (some) locality decisions to the TACOMA system. In the current TACOMA implementation, however, the `HOST` folder must still be used as described.

Notice that an agent has to be moved as part of a briefcase. All the agent does, is to wrap itself up in it's briefcase and do a `meet`. Alternatively, the agent is wrapped up by another agent or a human being.

6 Arriving at a New Place - The Bridge Head for the Arriving Agent

Agents travel from site to site. We can view an agent visiting a remote site as guest software. We will now take a closer look at how to provide a convenient environment for a guest agent touching base at a remote site.

The `meet` must have an entry point, a *bridge head*, at the destination site. This single entry point for guest agents lends itself naturally to authentication, access control, accounting, and provision of fault-tolerance. We use a particular firewall agent, the `tac_firewall` agent, for this purpose.

In the current implementation of TACOMA, `tac_firewall` basically logs the briefcase to disk. Then the agent must be extracted from the briefcase and executed. Activating a new agent is a time consuming task, especially if the guest agent has to be recompiled. To be able to quickly accept new incoming agents, the `tac_firewall` agent leaves this task to another agent, the `tac_exec` agent. The `tac_firewall` agent simply hands over the briefcase to the `tac_exec` agent and is then ready for a new `meet`. This extra level of indirection is used for performance reasons since `tac_firewall` is not replicated. A number of `tac_exec` agent replicas can exist in parallel, only one `tac_firewall` is active.

The `tac_exec` agent sets up the environment for the guest agent, the *place* for execution. As such, it can also restrict the execution environment at this particular site. In Figure 5, the agent `ag_local` is not a part of the place environment, but is still running on site 2. A guest agent activated in the place at site 2 can not meet with `ag_local` since it is outside the place.

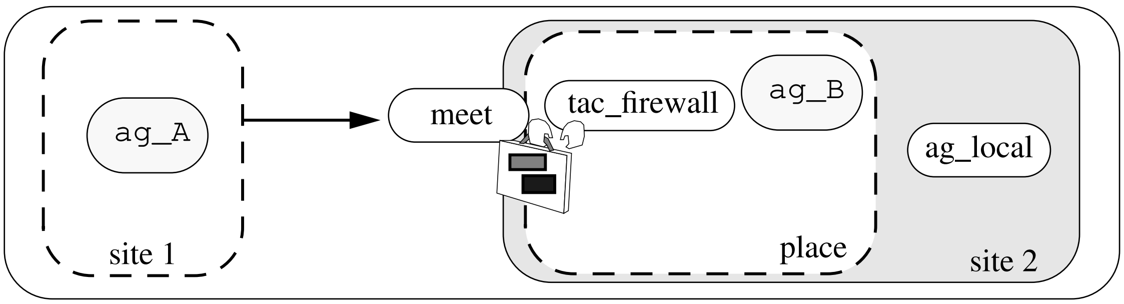

FIGURE 5. Transferring of a briefcase to site 2.

Figure 6 illustrates how `tac_exec` has taken over the responsibility for the arrived briefcase. The `tac_exec` agent now activates the original agent specified in the `meet`. In this example, this is `ag_B`. This agent can be a script interpreter that extracts the source code from the `CODE` folder and runs the guest agent. Notice now that `tac_firewall` is ready again to handle arrival of a new agent.

.

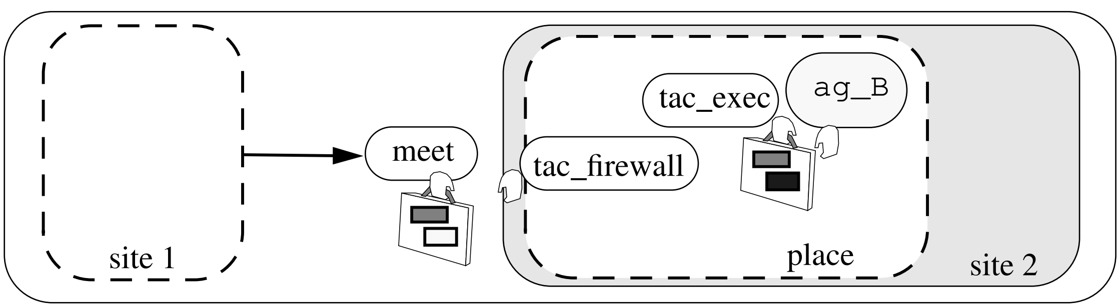

FIGURE 6. `Meet` with the specified agent `ag_B`.

7 Optimize for RPC

The agent model is fundamentally different from RPC [Birr 84] in that the clients and servers may travel in the system to a convenient place where they can meet. The meeting between the client and server then proceeds much like an RPC, but has much simpler failure semantics as there can be no message omission failures.

Agent applications can be structured as a sequence of agents, each one activated upon termination of its predecessor. This data driven model does not fit well with the client - server model, a model more familiar to distributed application programmers. As such, we want to optimize for RPC style of communication between agents.

The idea is that the client agent sends out an agent doing the remote computation. Then, this client agent does not terminate, but is suspended for a while. Upon completion of the remote agent, it travels back to the site of the client agent. There it activates the client agent again with the result from the remote computation. This is supported in TACOMA by having a `rpc` option as:

`meet agent briefcase rpc`

The system agent `tac_rpc` is used in this `meet`. The `tac_firewall` agent can not block on a RPC like session. At the same time, the communication channel between the two sites must be open for performance reasons. Therefore, the `tac_firewall` agent activates one of the local `tac_rpc` agents. The idea is that this `tac_rpc` agent creates a *new* connection directly with the client agent. From now on, the `tac_firewall` is not involved any more. The `tac_rpc` now proceeds just like the `tac_exec` does. Upon completion of the guest agent, however, the briefcase is sent back using the open connection. The `meet` now terminates, and the client agent can use the result from the remote agent computation.

Yet an alternative, is to send the continuation part of the calling agent *with* the remote agent. The client agent can now be resumed at any site in the system.

8 Summary

This section has introduced our model for agent based distributed computing. At the level we are studying, an agent is just a process with its state. This process can move about and execute at multiple sites in the network.

An agent can carry state along in a briefcase, or leave state behind in a file cabinet. Agents `meet`, they do not communicate by sending messages. If the agents are located at different places, they must co-locate by moving. The server interface may be kept primitive since it does not need to provide complex interfaces to optimize for low bandwidth connections.

Chapter 3

Getting Started

Abstract

This chapter gives a detailed description on how to get started with TACOMA Version 1. This TACOMA version is based on Tcl-TCP, an extension to Tcl that uses TCP for communication. We introduce how to write new TACOMA agents written in Tcl/Tk.

1 Introduction

We have designed and implemented several versions of the TACOMA distributed system where computations move around and execute in a heterogeneous network environment. This chapter provides a practical introduction to TACOMA Version 1. This TACOMA version is based on Tcl-TCP, an extension to Tcl that allows use of TCP for communication. We can use Tcl to write simple agents. More advanced graphical user interface agents can be created through the Tcl extension Tk. Tk is a toolkit for the X Window System.

We describe how to install TACOMA and relevant software. The reader should then be able to run some very simple TACOMA agents. The reminder of this document provides necessary details for construction of more complex agents.

Throughout, we use the following notation. A `Courier` font is used for any input given to the computer, such as UNIX, Tcl and TACOMA commands. The result (feedback from the computer) appears in *`Courier italic`* style. We use the prefix "**=>** " to indicate normal response. For instance, the syntax of activating the TACOMA Tcl interpreter (from `tclsh`) is as follows:

```
meet ag_tcl <briefcase>
```

This means that the named briefcase `<briefcase>` is given to the `ag_tcl` agent at a site specified in the `HOST` folder in the `<briefcase>`. If this is a remote `ag_tcl` agent, the `meet` abstraction ensures that the `<briefcase>` is moved there.

The name and content of `<briefcase>` is entirely user specific. The receiver of the briefcase expects to find certain standard folders in it. For instance, an agent interpreting Tcl code expects to find this code in the `TCLCODE` folder of the briefcase it receives.

2 A Simple Agent Example

We shall now look at a very simple example to illustrate the main features of TACOMA. To introduce the TACOMA abstractions, we start by sending a simple ASCII string to a remote place. This string is given to an echo agent at the remote place. The echo agent now writes the string to a file (in the common case, a UNIX device). Later in this chapter we will show that there is little differences between sending and displaying a string and sending and executing an agent.

The problem is as follows. We want the message "`lunch now - DJ`" to be displayed at the *remote* computer (`odin`). The message must be displayed on the console `/dev/console` by a local echo agent `ag_echo`.

Recall that agents (or plain messages) are transferred to a remote agent using the briefcase abstraction. In this example, we have to fill the briefcase manually, but this can also be done by an agent (for instance, replicating itself by putting its own source code into the briefcase). Yet an alternative is to pipe this from a file of predefined values.

The following commands typed to TACOMA will now create a briefcase `out_bc` and use the folders `HOST`, `DATA`, and `OUTPUT` for application specific data. Then, this agent wants to `meet` with the `ag_echo` agent at the remote site `odin`. This `meet` will move `out_bc` to `odin`, the site specified in the `HOST` folder. The `meet` abstraction uses a `CONTACT` folder to keep track of which agent to meet with. This is a pure system specific folder.

```
bc_create out_bc
folder_store out_bc HOST odin
folder_store out_bc DATA "lunch now - DJ"
folder_store out_bc OUTPUT /dev/console
meet ag_echo out_bc
```

Next, the message "`lunch now - DJ`" is displayed at the remote site `odin` at `/dev/console`. The `meet` does not block on remote events more than necessary. The `meet` terminates once the transported briefcase is successfully delivered at the remote place (`odin`). In reality, yet another agent, `tac_exec`, is activated to handle multiple arriving agents. Figure 1 illustrates this example.

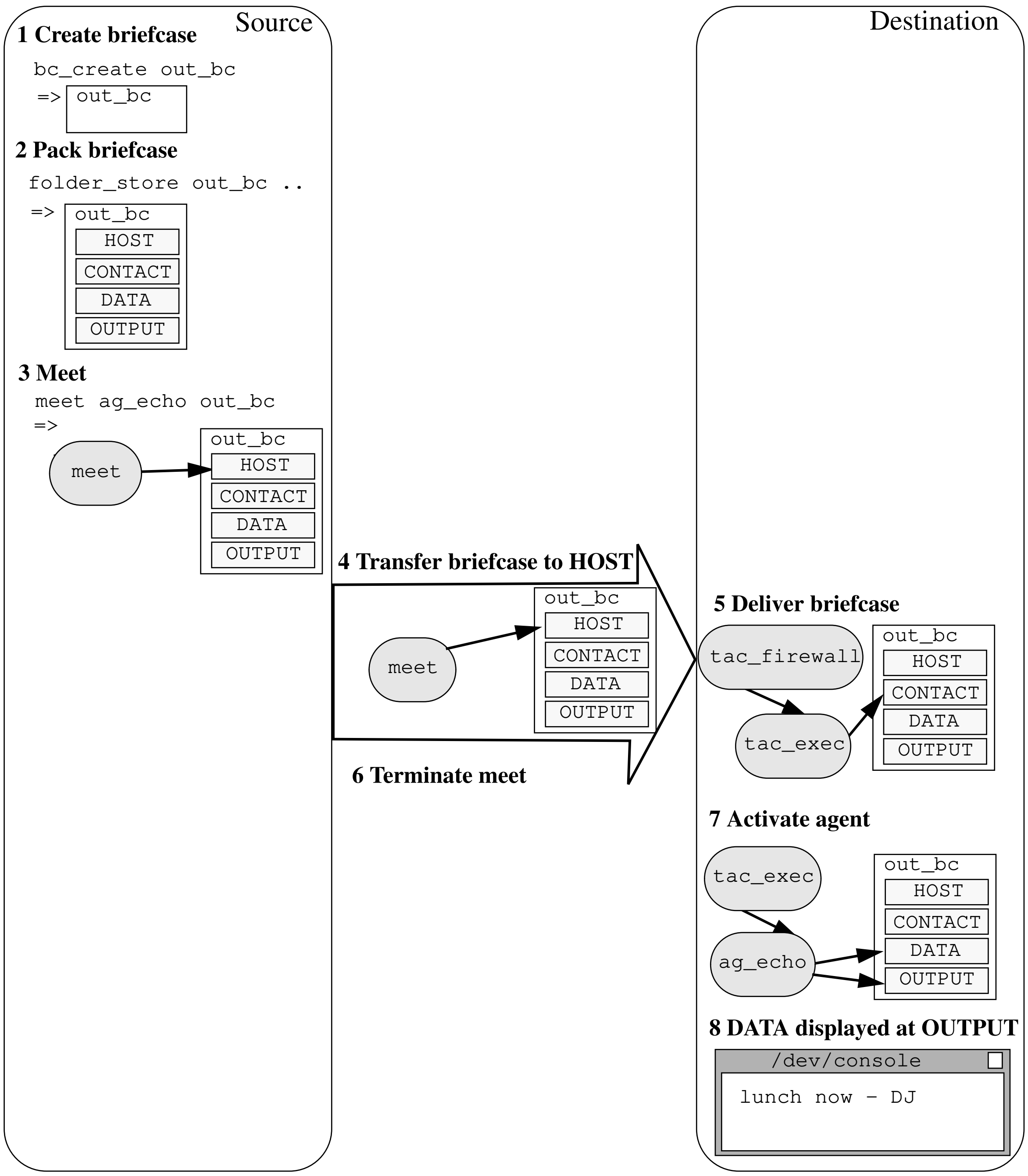

FIGURE 1. Remote echoing example.

3 Installing TACOMA and Relevant Software

TACOMA Version 1 is based on Tcl-TCP and UNIX (HP-UX). This section will describe step by step how to obtain and install TACOMA. We presume that Tcl-TCP has been installed[1].

3.1 Installing TACOMA

The TACOMA source can be obtained from the TACOMA www page at:

```
URL: http://www.cs.uit.no/DOS/Tacoma/
```

There, you will find the file:

```
tacoma_v1.0.dist.tar.gz
```

You can place this file in your home directory or any other convenient place. In this example, the current directory where we install TACOMA is in:

```
/users/dag/TACOMA/src/
```

To extract the different libraries and agents, execute the following UNIX commands:

```
gunzip tacoma_v1.0.dist.tar.gz
tar xf tacoma_v1.0.dist.tar
```

A new directory will be created: `/users/dag/TACOMA/src/tacoma`

Move to this directory. This directory contains six subdirectories:

- `bin/` which contains tcltcp and wishtcp. These files contains compiled versions of Tcl-TCP and Tk-TCP. If your workstation is not running HP-UX, you can install Tcl-TCP and Tk-TCP yourself in this directory.
- `lib/` which contains the `meet` abstraction and support agents like `ag_echo` and `ag_tcl`. This directory also contains `tacoma.tcl`. This file contains Tcl procedures implementing folder abstractions as `folder_store`, `bc_create`, and `archive`. The file `extensions.tcl` contains similar abstractions.
- `man/` containing manual pages. You can either add this directory to your man path and rename the `mann` entries, or you can use this command to display them directly:

```
nroff -man <filename> | more
```

A postscript version (`Manuals.ps`) of the man pages is also included in the distribution.

1. Tcl-TCP can be obtained from: **URL:ftp://ftp.aud.alcatel.com/tcl/extensions/**
File: tclTCP2.1.tar.gz

- `cabinets/`, which contains local cabinets.
- `examples/` which contains TACOMA agent examples.
- `sysagents/` which contains `ag_wish`. This is for execution of agents that use the Tk extension.

The `tacoma` directory also contains the most vital TACOMA system agents:

- `tac_firewall`.
- `tac_exec`.
- `tac_rpc`.

The next step is to start TACOMA at *your* workstation. Notice that your agents can only be executed locally (unless TACOMA is running somewhere else).

3.2 Running TACOMA

The following steps must be done to start TACOMA. First, the `HOST` environment variable must be set. This should already have been set under normal conditions, but in case of problems do:

```
setenv HOST hugin
```

Your host name in this example is `hugin`. The following TACOMA environment variables must also be set:

```
setenv TACOMAPATH /users/dag/TACOMA/src/tacoma
setenv TACOMAPORT 13147
```

It is important that *all* TACOMA sites that want to communicate specify the *same* port number (as long as we do not have a name server). Now the local firewall agent can be started as a *background* process:

```
./tac_firewall &
```

A successful start is indicated by:

```
=>  [1] 27301
    TACOMA v1.0

    Agent tac_firewall started at hugin (on port 13147)
    Date: Wed Mar  1 11:27:22 MET 1995
    Path: '/users/dag/TACOMA/src/tacoma'
    Starting tac_rpc ... done
```

The `tac_firewall` has now started three `tac_rpc` agents as part of the start up procedure.

3.3 Run the First TACOMA Agent

TACOMA is now ready to be used. The next step is to try an interactive example. This works as follows. First, you must start Tcl-TCP by the following command:

```
./bin/tcltcp
```

Then, you must specify where the TACOMA library routines are located:

```
set auto_path [linsert $auto_path 0 $env(TACOMAPATH)/lib]
```

=> *`/users/dag/TACOMA/src/tacoma/lib /usr/local/lib/tcl`*

Notice that the Tcl interpreter always feedbacks the result.

All agents must have a briefcase, so you must create a briefcase. In this example, we name the briefcase `bc`:

```
bc_create bc
```

=> *`227694011324451`*

The next steps use standard folders as `HOST` and `DATA`. Notice that `bc` is empty (has no `HOST` and `DATA`). The agent given briefcase `bc`, however, expects to find some predefined (standard) folders in `bc`. Consequently, we must specify where to execute the agent. We put the destination site into the `HOST` folder in the briefcase `bc`:

```
folder_store bc HOST hugin
```

=> *`hugin`*

Notice that, in this example, the destination site is the same as the local machine. In this example, the `CONTACT` agent is the echo agent that accepts a briefcase as input and displays the content of the `DATA` folder on the display specified in the `OUTPUT` folder. Now, we must put the ASCII string to be echoed by `ag_echo` into the `DATA` folder.

```
folder_store bc DATA "lunch again, NS"
```

=> *`lunch again, NS`*

Finally, we must specify where this data should be displayed:

```
folder_store bc OUTPUT "/dev/console"
```

=> *`/dev/console`*

The `bc` briefcase is now ready to be sent to the destination site. This is done by the `meet` abstraction:

```
meet ag_echo bc
```

=> *0*

The `meet` will now travel to the destination site `hugin`. The briefcase `bc` will now be delivered to the local `tac_firewall`. The `meet` will terminate once this delivery is successful. The `tac_firewall` now hands over the briefcase `bc` to a `tac_exec` agent. This `tac_exec` will now activate the agent specified in the original `meet`. This agent, `ag_echo`, is found in the `CONTACT` folder. Next, `ag_echo` is activated. It expects to find an `OUTPUT` folder and a `DATA` folder in the given briefcase. Finally, on `console` at site `hugin`, the following will be displayed:

=> *lunch again, NS*

Now, you can leave the Tcl shell (ctrl-d). To check that the agent left the source node and arrived at the destination node, check the logs:

```
ll cabinets/inlog
```

=> *total 2*

```
-rw-r-----   1 dag      users          87 Mar  1 11:59
hugin.cs.UiT.No.1
```

```
ll cabinets/outlog
```

=> *total 2*

```
-rw-r-----   1 dag      users          87 Mar  1 11:59
hugin.27307
```

3.4 Included Examples

Now, you should try to run some of the included TACOMA examples. Currently, this is the following agents:

- `ag_prompt.tcl hostname text`. This agent outputs user specified `text` at the `console` of a remote site (`hostname`).
- `ag_motd.tcl hostnames codefile`. This example agent travels to the first `hostname` and executes the code found in `codefile` at that site. You might use `ag.motd.codefile`. It may travel to a second site (if two host names are given) and output the contents of a `DATA` folder in the console window of that site. If not, it returns to home.
- `ag_query.tcl hostname question`. This example agent travels to `hostname` and asks the user at that site the `question`. It will then return with a y/n answer.
- `ag_query2.tcl question user [user ...]`. This example agent first maps the user names to host names (using UNIX `rwho`). Then, it travels to the host(s) (given by the `rwho` mapping) and asks the user(s) the `question`. Finally, it will return with the answer(s). This agent times out after three minutes on each question. If the user is on a machine not supporting TACOMA, the agent will register this in the `ANSWER` folder.

To illustrate, the first example can be executed by issuing:

```
./examples/ag_prompt.tcl hugin "meeting NOW, boss"
```

The output in `/dev/console` at `hugin` is:

`=>` *`meeting NOW, boss`*

4 Summary

This chapter has detailed how to get started with TACOMA Version 1. This is our current TACOMA implementation based on UNIX and Tcl-TCP. The next step is to create your own TACOMA agents, a subject addressed in the rest of this document.

Chapter 4

The TACOMA API

Abstract

This chapter presents more details on the TACOMA API. The API includes folder, briefcase and file cabinet abstractions, and abstractions for meeting and executing agents. We will also discuss how to structure distributed agent applications based on these abstractions.

1 Introduction

The previous chapter might have left the reader with the impression that even simple agents require a complex infrastructure. This would be a wrong impression. A single `meet` abstraction, some folder abstractions, and a few TACOMA system agents are sufficient for writing agents that can roam the internet.

The good approach to TACOMA is learning by doing. To experience the simplicity in the TACOMA system interface, you should implement some realistic agent applications. This chapter provides more details to be used in this process. We will also present some simple agent structuring techniques and a concrete agent example.

2 Programming the Internet - Folder Abstractions

We can use the basic agent abstraction `meet` to transfer control and state around in the internet. We do not restrict this to agents written in a proprietary language or to binaries. Source code in whatever language can be moved about by TACOMA. At the receiver side, however, there must be compiler or interpreter support for this particular language. The basic system support for this *openness* is not visible for the agent programmer. The programmer just uses the basic abstraction:

```
meet agent briefcase
```

This will result in activation of `agent` with `briefcase` as argument. The use of the system agents `tac_firewall` and `tac_exec` are transparently hidden for the application programmer. These agents use information found in the supplied `briefcase` to set up the environment for the guest agent (which is represented in the `briefcase` as well). Some basic folder abstractions are necessary for manipulating on folders found in both briefcases and file cabinets. We will briefly present their interfaces in the following.

First, we need to be able to create a briefcase. The following abstraction creates a local briefcase called `name`:

```
bc_create name
```

Similarily, we can delete a local briefcase `bc` and list the content of `bc` by the following abstractions:

```
bc_discard bc
```

```
bc_list bc
```

Now, we need abstractions for manipulating on specific folders in a briefcase. The next abstractions manipulate on `folder` in `bc`, the folder being pure, *unstructured* data:

```
folder_store bc folder data
```

```
folder_append bc folder data
```

```
folder_fetch bc folder
```

```
folder_delete bc folder
```

The `folder_store` stores some `data` in `folder` of briefcase `bc`. The `folder_append` appends data to `folder`. The abstraction `folder_fetch` fetches the content in `folder` of briefcase `bc`. The abstraction `folder_delete` deletes the specific `folder` in `bc`.

We found it convenient to be able to structure folder data, and we introduced folders with data elements ordered in *lists*. TACOMA supports folder abstractions that manipulate on list structures in specific folders. General structures as, for instance, a stack can now be made. The following TACOMA abstractions operating on list structures are currently supported:

```
folder_shift bc folder direction amount data
```

(shifts the elements of folder in specific direction)

```
folder_rotate bc folder direction amount
```

(rotates the elements of folder in specific direction)

The `folder_shift` abstraction can be used to implement well-known stack and queue operations. However, the TACOMA system also supports these more intuitive abstractions directly. This is:

```
folder_pop bc folder
```

(fetches (removes) the <u>first</u> element from folder)

```
folder_push bc folder data
```

(adds an element to <u>start</u> of folder)

```
folder_top bc folder
```

(reads (not removes) the first element of folder)

```
folder_add bc folder data
```

(adds an element to end of folder)

The TACOMA system also supports packing and unpacking of briefcases. This is required functionality because Tcl *arrays* can not be moved about. Such a structure can not be treated as a single object when passed around. The following abstractions packs and unpacks a briefcase `bc`:

```
archive bc folder
```

(packs bc into folder)

```
unarchive bc archive
```

(unpacks archive into bc)

TACOMA supports *file cabinet* abstractions similar to those manipulating on briefcases. Archive and rotate operations are not implemented, but the following abstractions are:

```
cabinet_create name
```

(creates a new local cabinet)

```
cabinet_close cabinet
```

(closes a cabinet)

```
cabinet_delete cabinet folder
```

(deletes a specific folder from cabinet)

```
cabinet_discard cabinet
```

(deletes a cabinet and all its folder)

```
cabinet_fetch cabinet folder
```

(retrives content of cabinet folder)

```
cabinet_list cabinet
```

(list cabinet folders)

```
cabinet_open cabinet
```

(opens a cabinet)

```
cabinet_store cabinet folder data
```

(stores data in specific folder of cabinet)

3 Programming the Internet - Structuring Techniques

Agent applications can be structured differently. Fortunately, TACOMA agent applications are not limited to a single paradigm like, for instance, the client-server paradigm. At the same time, the agent paradigm can be used to attack problems already solved today by other paradigms. In the following, we will present some general structuring techniques to illustrate this.

3.1 Jumping Around - A Sequence of Agents

An agent application can be structured as a series of code sequences (agents) executed on a series of sites, one at a time. A typical agent application can be structured as a *troop* of agents moving about. A very simple example is to have just two agents in an agent application. The first agent executes for a while and terminates. As part of this termination (through a `meet` operation), the second agent is started and runs until termination. Also, this second agent can be a copy of the first agent.

A more complex example has a large number of agents to be executed at different sites, but they all have the same structure. The agents are run one after the other, and all but the last one terminates by a `meet`. This is illustrated in the event diagram in Figure 1, where the code pieces of an agent application are structured as a series of three agents `A`, `B` and `C`. They are executed, one after the other, on three different sites. Any remote operation is through the `meet` operation, *not* through any communication mechanism.

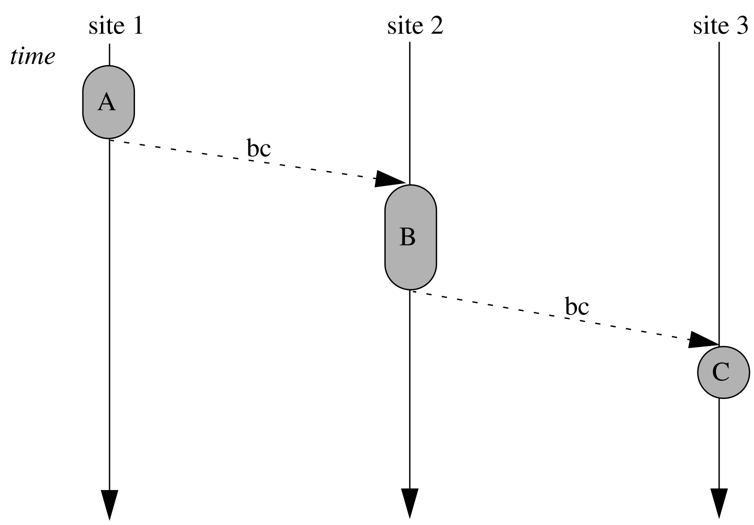

FIGURE 1. Jump execution.

The source code of each of these agents are packed into the `CODE` folder of the briefcase `bc`. This can be Tcl code that has to be given to the Tcl-interpreter, `ag_tcl`, at the new site. This TACOMA agent will extract the code of the agent, set up a virtual environment for it, and run it.

The following pseudo code illustrates this structuring technique:

```
# agent_A (at site1)

......
pack bc:
  HOST = site2
  CODE = agent_B
  ......
meet ag_tcl bc
# move bc to site2 and give bc to ag_tcl
# exit (terminate this agent)


# agent_B (at site2)

use data in supplied bc
do someting
......
pack bc:
  HOST = site3
  CODE = agent_C
  ......
meet ag_tcl bc
# move bc to site3 and give bc to ag_tcl
# exit (terminate this agent)


# agent_C (at site3)

use data in supplied bc
......
......
# exit (terminate whole application)
```

3.2 Remote Activation

The agent paradigm can be used to activate another, remote agent. Once activated, the two agents can proceed totally independent of each other. Hence, we can regard them as two different applications `A` and `B`. This structuring technique is illustrated in Figure 2. Agent `A` packs a briefcase `bc` with agent `B`, then it does a `meet` with the code interpreter at the destination site. This system agent will extract the source code for agent `B` from `bc` and execute it. Once the `meet` operation terminates, the next instruction of agent `A` can be executed.

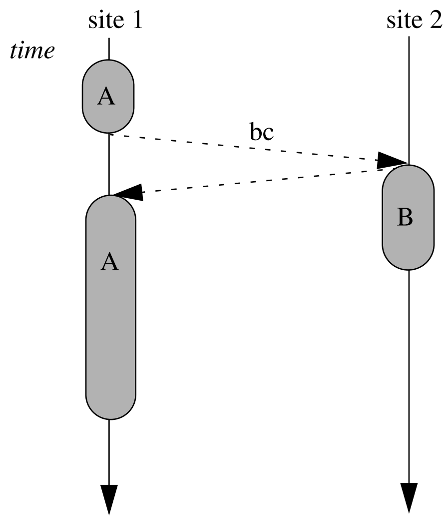

FIGURE 2. Forking off and activating remote agent.

The following pseudo code part of agent `A` and agent `B` illustrates this structuring technique:

```
#agent_A

......
pack bc:
  HOST = site2
  CODE = agent_B
  ......
meet ag_tcl bc
# meeting done

if delivery OK then
  next task to do
  ......
  ......

# exit (agent_A)


# agent_B

use data in supplied bc
......
......

# exit (agent_B)
```

3.3 Client - Server Style of Computing

The client-server model is widely adopted in distributed systems. In short, a client issues a request for some (remote) service and is blocked. A remote server implementing the requested service will accept the request, carry out the operations to service this request, and return the final reply to the client. Now, the client is activated and can use the reply.

TACOMA provides a flexible way to dispatch some remote agent computation. The flexibility now, is that the agent that is to recieve the result, can be *sent along* with the request. We call this agent the *continuation* part of the client agent. The advantage of this structuring technique is that no dependencies are left on the source node. In the client - server model, the blocked client is left back on the initial site. Figure 1 illustrates this technique, where agent `C` can be seen as the continuation part of agent `A`. Figure 3 shows that this continuation part can be executed at the same site as where the original request was issued.

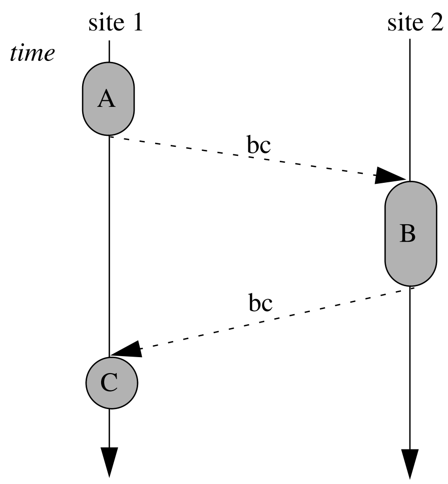

FIGURE 3. Client -server style of computing.

The following pseudo code part of agent `A` implements the structure of Figure 3, where the result is sent back to the initial site:

```
# agent_A

......
pack bc:
  HOST = site2, site1
  CODE = agent_B, agent_C
  ......

meet ag_tcl bc
# exit (terminate this agent)
```

```
# agent_B

start execution,
use data in supplied bc
carry out service and generate reply
pack bc:
  HOST = site2, site1
  CODE = agent_B, agent_C
  RESULT = reply

meet ag_tcl bc
# exit (terminate this agent (remote procedure))

# agent_C

start execution
use data (RESULT) in supplied bc
......
# exit (terminate application)
```

3.4 Optimize for RPC

TACOMA is optimized for RPC interactions. The calling agent `A` can now be blocked while the remote agent `B` is moving about carrying out the requested service. This server agent `B` will travel back to the initial site with the final result in its briefcase `bc`. There, agent `A` will be re-activated and can use the content of `bc`. This is illustrated in Figure 4.

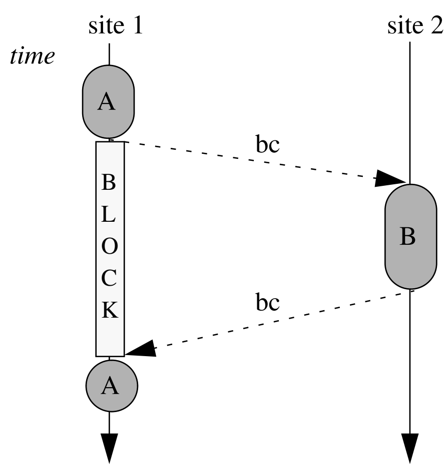

FIGURE 4. RPC style of computing.

We introduce the following option in the `meet` abstraction to support this optimization:

```
meet ag_tcl bc rpc
```

The following pseudo code part of agent `A` an `B` illustrates this structuring technique:

```
# agent_A

......
pack bc:
  HOST = site2, site1
  CODE = agent_B
  ......

meet ag_tcl bc rpc
# reply now received

use data (RESULT) from agent_B computation
......

# exit (terminate application)

# agent_B

start execution
use data in supplied bc
carry out service and generate reply
pack bc:
  (HOST = site2, site1)
  RESULT = reply

   # exit (terminate this agent (remote procedure))
```

3.5 Event Synchronization

The agent paradigm is suitable for event-driven programming. That is, when a a specific event occurs, a certain piece of code (an agent) must be executed. Examples of events can be, for instance, that temperature is exceeding a threshold value, a user is logging into a machine, or a computation has terminated.

We use the same structuring technique as previously described. At least three different agents are needed; one agent that creates and sends out another monitoring agent, the event monitoring agent itself, and the agent to be activated when the event occurs.

A distributed application can now be structured with an event monitoring agent `agent_B`, who continuously monitors its environment. When the specific event occurs, it will activate another agent `agent_C`. This is illustrated in Figure 5.

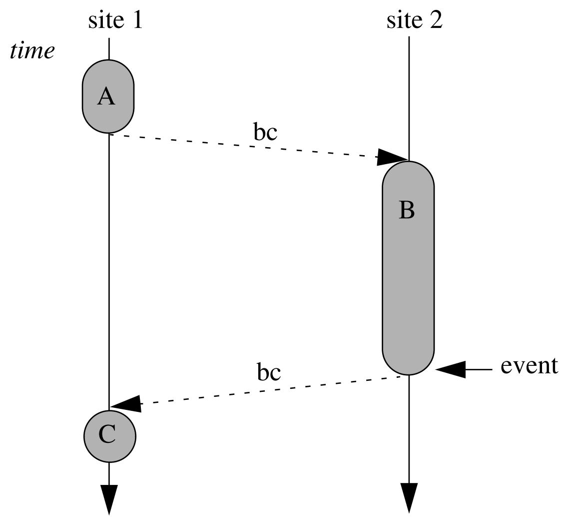

FIGURE 5. Event based agent activation.

One concrete `agent_C` example is simply to notify a (remote) user that the event has occurred. This can be done by, for instance, a new window popping up on the user's screen with monitoring information.

This pseudo code illustrates how to structure an application doing some remote monitoring:

```
# agent_A

......
pack bc:
  HOST = site2, site1
  CODE = agent_B, agent_C
  ......

meet ag_tcl bc
# exit (terminate creating agent)
```

```
# agent_B

loop:
  monitor event
  upon event exit loop
end loop
pack bc:
  Update STATUS folder in bc
  HOST = ~~site2~~, site1

meet ag_tcl bc
# exit (terminate this agent (remote procedure))

# agent_C

start execution
pop up Tk display window
display monitoring information
......
# exit (notification agent)
```

3.6 Sharing Common State

Applications must be able to share common state. One way is simply to use agents as couriers moving this shared state about. Another approach is to use file cabinets and shared folders for this purpose.

Figure 6 illustrates this concept, where `agent_A` and `agent_B` both access a shared folder through `agent_C`. This `agent_C` can be site local guarding this file cabinet, or it can be shipped in for this purpose.

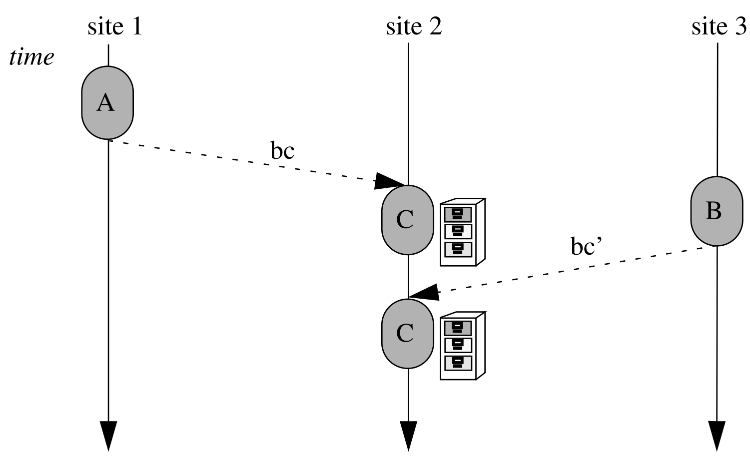

FIGURE 6. Sharing common state.

The following pseudo code illustrates this structuring technique:

```
# agent_A and agent_B

......
pack bc:
  HOST = site2
  CODE = agent_C
  DATA = shared state
  ......
meet ag_tcl bc
# exit (terminate) or proceed execution

# agent_C

wait for access:
    open file cabinet
    access shared folder
.......
# exit (terminate this agent)
```

3.7 Parallel Processing

A final structuring example illustrates how to do parallel processing. A number of agents can be activated and executed in parallel. Upon completion, each agent might `meet` with a controller assembling the final result. This is illustrated in Figure 7, where the agent replicas of `agent_B` execute in parallel at two different sites. Upon completion, they both activate the controller agent `agent_C` located at another site.

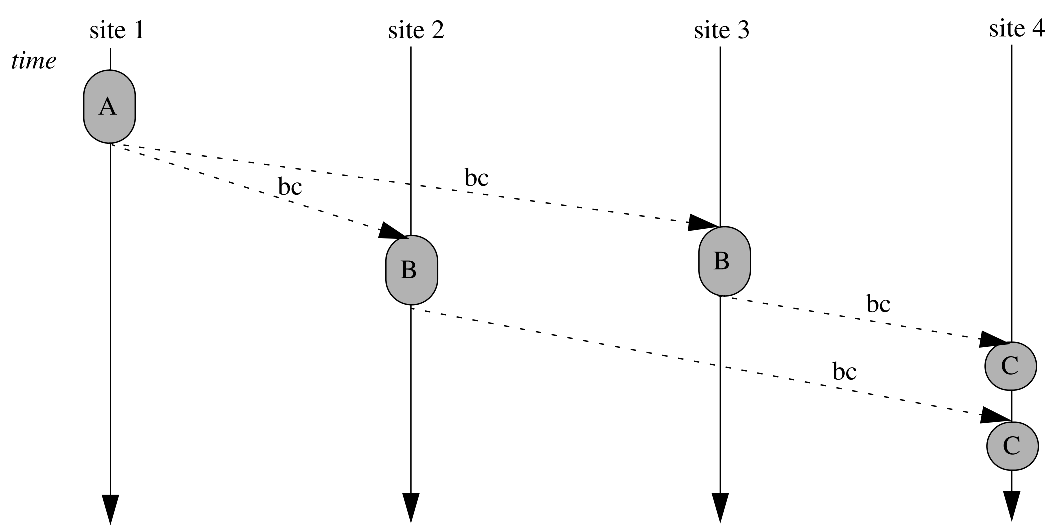

FIGURE 7. Parallel processing with agents.

Notice that this scheme also applies for fault-tolerant programming of agents. The agent replicas of agent_B provide redundancy, agent_C is the voting agent needed to mask the result of the redundant computations. This agent_C can also be sent along with the briefcase bc. File cabinets must then be used at the rendezvous site for voting purposes.

This pseudo code illustrates how it is possible to build a more fault-tolerant agent application through redundancy:

```
# agent_A

......
pack bc:
  HOST = site2, site4
  CODE = agent_B, agent_C
    ......
meet ag_tcl bc
# send first replica out

re-pack bc:
  HOST = site3, site4
meet ag_tcl bc
# send second replica out
# exit (terminate)

# agent_B

start execution
generate (this version of the) result
pack bc:
  HOST = site_i, site4
  RESULT = result

meet ag_tcl bc
# exit (terminate this replicated agent)

# agent_C

open file cabinet
access shared (voting) folder
if sufficient input:
                     vote
else
      deposit voting contribution

# exit (terminate or notify another agent)
```

4 Writing an Agent - An Example

A concrete Tcl agent example will now be presented. In functionality, this agent travels to a site specified by the user. There, the agent found in the `CODE` folder of the briefcase `my_bc` is executed by the local `ag_tcl`. Finally, it travels to a third site (default is the initial site) and outputs the contents of the `DATA` folder in the console window.

This is the Tcl code of the agent creating and sending another agent out in the network:

```
#!./bin/tcltcp

set auto_path [linsert $auto_path 0 ./lib]

# get the site to execute the agent and the name
# of the file with the agent source code:

if {$argc != 2} {
    puts "Usage: $argv0 \[hostnames\] \[codefile\]\n"
    exit
}

# create and pack briefcase:

bc_create my_bc
  folder_store my_bc HOST [lindex $argv 0] ""
  folder_store my_bc RETURNHOST $env(HOST)
  set file [open [lindex $argv 1] r]
  folder_store my_bc MYCODE [read -nonewline $file]
  close $file
  folder_store my_bc TCLCODE "MYCODE" ""
  folder_store my_bc OUTPUT "/dev/console"

# meet with and deliver the briefcase to ag_tcl
# at the remote site:

meet ag_tcl my_bc

# optional to check status of meeting:

puts [folder_fetch my_bc STATUS]
exit
```

This is the agent code to execute remotely (found in file specified by the user):

```
proc main {bc} {
    upvar $bc loc_bc
    set error [catch {open "/etc/motd" r} file]
    if { $error == 0 } {
        folder_store loc_bc DATA [read -nonewline $file]
        close $file
    } else {
      folder_store loc_bc DATA "No msg of the day at host"
    }
    meet ag_echo loc_bc
    return 0
}
```

5 Summary

This chapter has detailed the TACOMA API. It basically consists of a single `meet` abstraction with optional RPC semantics. In addition, a variety of folder, file cabinet and briefcase abstractions are supported.

Distributed applications can be structured as a troop of agents moving about. We have sketched how to solve some of the more common structuring problems in distributed environments. A concrete agent example shows how to pack an agent into a briefcase and how to transfer this to a remote site for execution. This is done without any networking syntax involved.

References

[Birr 84] Birrell, A.D., and B.J. Nelson. Implementing Remote Procedure Calls, *ACM Trans. Comp. Sys*., 2, (Feb. 1984), pp. 39-59.

[Card 95] Cardelli, L., A Language with Distributed Scope, *Computing Systems*, 8 (1), Winter 1995, pp. 27-59.

[Chau 92] Chaum, D., Achieving Electronic Privacy, *Scientific American*, 267 (2), August 1992, pp. 96-101.

[Joha 93] Johansen, D. "StormCast: Yet Another Exercise in Distributed Computing", in *Distributed Open Systems*, (eds. Brazier, F., Johansen, D.), IEEE Computer Society Press, USA, Oct. 93, pp. 152-174.

[Joha 94] Johansen, D., Hartvigsen, G. Convienient Abstractions in StormCast Applications. *Proceedings of the 6th. ACM SIGOPS European Workshop: "Matching Operating Systems to Application Needs"* (Sept. 12-14, 1994, Dagstuhl, Germany), pp. 11-16.

[Joha 95] Johansen, D., Renesse, R. van, and Schneider, F.B., Operating System Support for Mobile Agents, *IEEE 5th Workshop on Hot Topics in Operating Systems (HOTOS-V),* Washington, USA, May 1995, pp. 42-45.

[Oust 94] Ousterhout, J.K., Tcl and the Tk Toolkit, *Addison-Wesley*, ISBN 0-201-63337-X.

[Rene 94] Renesse, R. van, Hickey, T.M., and Birman, K.P., *Design and Performance of Horus: A Lightweight Group Communications System*, Cornell Tech. Report, TR 94-1442, August 1994.

[Riec 94] Riecken, D. (guest editor), Intelligent Agents, *Commun. of the ACM*, 37 (7), July 1994, pp. 19-21.

[Whit 94] White, J.E., Telescript Technology: The Foundation for the Electronic Marketplace, *General Magic White Paper*, General Magic Inc., 1994.