Performance Issues in TACOMA

Dag Johansen[1]
Nils P. Sudmann[1]
Robbert van Renesse[2]

[1] Department of Computer Science, University of Tromsø, NORWAY***
[2] Department of Computer Science, Cornell University, Ithaca, NY, USA†

Abstract. Mobile code performance depends, in part, on the costs of transferring an agent from one host to another and of initiating execution of that agent on a target host. These costs are reported for TACOMA (Tromsø and COrnell Moving Agents) v1.3, a UNIX-based system that supports agents. The experiments suggest opportunities for performance enhancements, both by changing the underlying operating system and by changing the architecture of the TACOMA run-time system.

1 Introduction

One of the motivations for using mobile code in distributed applications is the potential for improved performance. Moving a program between hosts in a network may be cheaper than moving large amounts of data between those hosts. Of course, the performance improvement will depend on the relative cost of moving and installing code.

This paper describes detailed measurements of move and install operations for mobile code that is run using the TACOMA (Tromsø and COrnell Moving Agents) system[3]. In TACOMA, a piece of mobile code is called an *agent* and is accompanied by state information in a *briefcase* [JvRS95]. By executing a *meet* operation, an agent initiates the installation and execution of code at another host.

TACOMA v1.3, used in these experiments, is the latest in a series of implementations. The system runs under most flavors of UNIX, and it supports agents implemented in C, Perl, Python, Scheme, and Tcl. Other agent systems, like Telescript [Whi94], Agent-Tcl [Gra95] and Messengers [LFB96], have similar architectures to TACOMA. However, in contrast to these other systems, TACOMA does not depend on properties of a specific programming language for preserving the integrity of hosts that execute agents; TACOMA relies only on operating systems mechanisms (e.g., address spaces and file system protection) for host integrity.

*** This work was supported by NSF (Norway) grant No. 17543/410 and 111034/410.
† This work was supported by ARPA/ONR grant N00014-92-J-1866.
[3] For more information: URL: http://www.cs.uit.no/DOS/Tacoma/

2 The Experiment

In TACOMA, a meet operation is executed by an *initiating agent* to cause the execution of a *target agent* on some specified host. The syntax of the meet is:

meet $ag@h$ bc [sync|async]

Execution causes target agent ag at host h to be executed using briefcase bc. A briefcase is a collection of named folders, each containing an uninterpreted sequence of bits. When async is specified, the initiating agent continues executing in parallel with execution of ag; otherwise the initiating agent blocks. The experiments of this paper are based on blocking meets with a target agent that has a null body (it simply accepts and returns a briefcase), and varying sized briefcases.

Our experiments were conducted using 2 Hewlett-Packard C-160 workstations running HP-UX (version 10.20) executing in single-user mode and with TACOMA's logging and security features disabled. Each workstation was equipped as follows:

- 160 MHz (PA-8000) CPU.
- 128 Mbyte RAM.
- 4 Gbyte F/W differential disk.

The workstations were connected by a dedicated 10 Mbit/s Ethernet segment.

3 Implementation and Performance

The critical path of meet is depicted in the time-space diagram of Figure 1. There, the leftmost arrow represents the initiating agent; the rightmost arrow represents the target agent. The two other arrows correspond to TACOMA system processes that are involved in implementing the meet: tac_firewall and ch_firewall. Both of these processes run on the host executing the target agent. tac_firewall monitors a well-known network port and forks an instance of ch_firewall for each incoming meet request.

The 13 labeled steps in Figure 1 can be grouped as follows:

- Marshalling and sending a briefcase (steps 1 - 4).
- Receipt of briefcase by ch_firewall (steps 5 - 6).
- Creation of an execution environment for the meeting (steps 7 - 10).
- Sending a message back to the initiating agent (steps 11 - 13).

Each of these is explained in the subsections that follow. To measure these, we ran 100 meet operations and profiled each using calls to gettimeofday(). In some cases, we summarize our experimental results using a *sample distribution plot* (SDP), which gives the number of samples that fall within each of 10 consecutive time intervals of equal length.

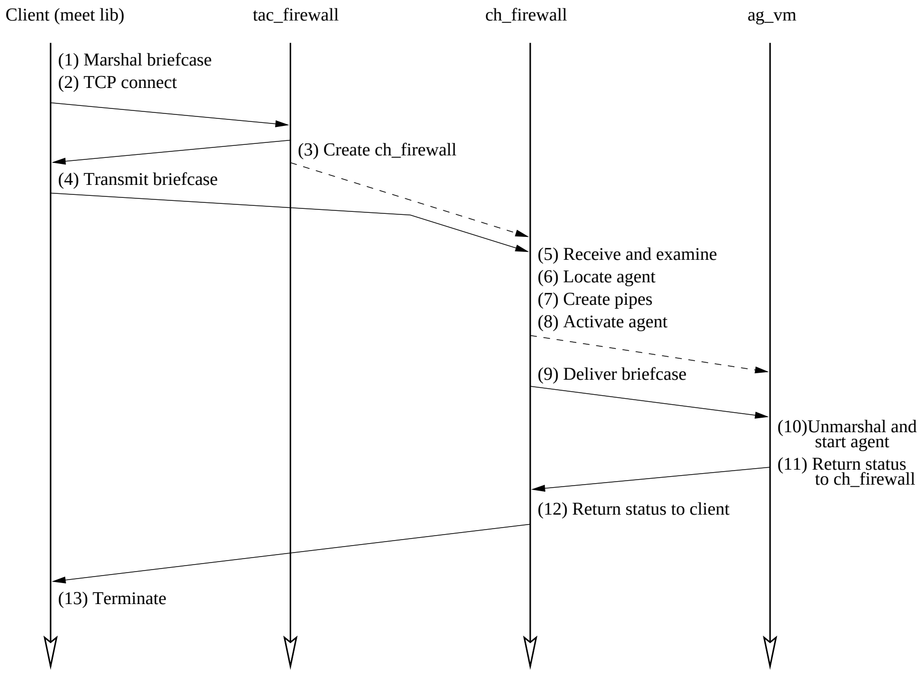


Fig. 1. Event diagram illustrating a meet operation.

3.1 Marshalling and Sending

Step 1 - Marshalling a briefcase: In this step, the briefcase is marshaled for transmission. The briefcase contains a collection of strings. For a minimal size briefcase (42 bytes), the median cost measured for this step is 31 microseconds. Figure 2 shows the median cost as a function of the briefcase size (which depends on the size of the strings).

Step 2 - TCP connect: In this step, a TCP connection is established with tac_firewall. The median cost measured for this step was 597 microseconds, and is independent of the briefcase size. Figure 3a is an SDP for this step.

Step 3 - Creating ch_firewall: Here there are two sub-steps. a) tac_firewall forks to create ch_firewall for processing the meet request. (After forking, tac_firewall is ready to receive another meet request.) The median cost measured for the fork was 1678 microseconds. b) An additional 97 microseconds is required for ch_firewall for initialization. Figure 3b is an SDP for this entire step. The elapsed time is largely independent of the briefcase size.

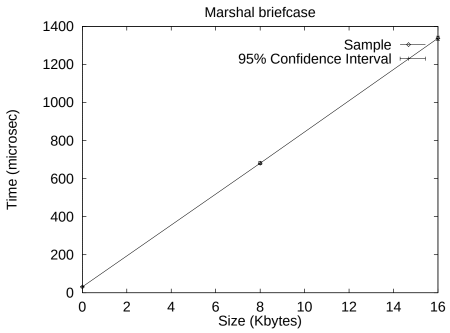
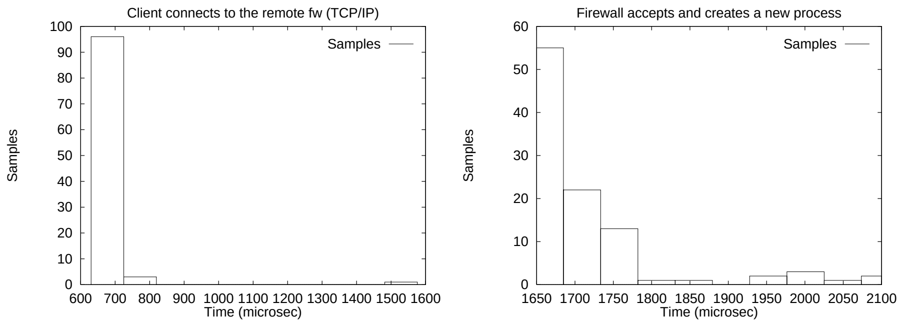


Fig. 2. Marshalling cost vs. briefcase size.



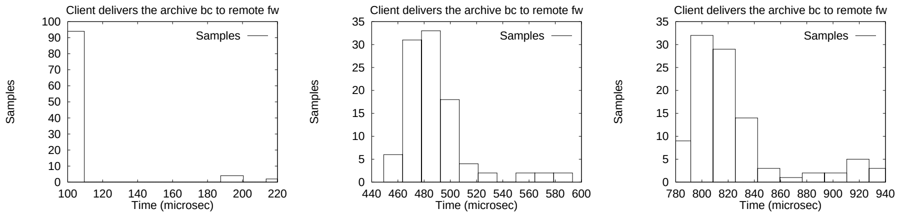
(a) (b)

Fig. 3. (a) Cost of connecting to tac_firewall; (b) Cost of tac_firewall forking to create ch_firewall.

Step 4 - Transmitting the briefcase: In this step, the marshaled briefcase is transferred to ch_firewall. The median cost for executing the send (for a minimal briefcase) was measured as 57 microseconds; this reflects only the elapsed time to copy the briefcase into the kernel at the sender. Figure 4 gives SDPs for three different briefcase sizes. Notice that step 4 overlaps the execution of step 3.

3.2 Receipt of the Briefcase

Step 5 - Receive and examine: Here there are two sub-steps. a) ch_firewall receives the incoming briefcase. The median cost measured for this step was 1510 microseconds. The receive (read) itself accounts for a surprising 1507 microseconds — we are currently attempting to isolate the reasons. b) The remaining



(a) 0k (b) 8k (c) 16k

Fig. 4. The cost of shipping a briefcase to the ch_firewall.

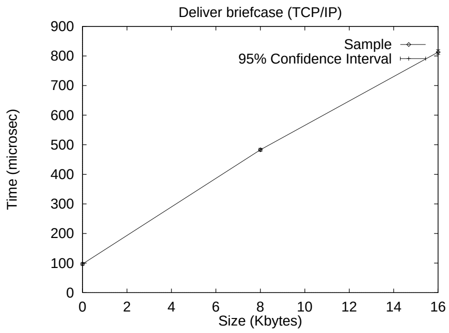


Fig. 5. The median cost of briefcase transmission; varying the size of the briefcase.

3 microseconds are spent checking the message contents for consistency with its header.

Step 6 - Locate local TACOMA agent: In this step ch_firewall determines the location of the executable for the target agent. This involves issuing a stat() kernel call. The median cost measured for this step was 152 microseconds.

3.3 Building an Execution Environment

Step 7 - Create pipes: In this step, ch_firewall creates two UNIX pipes for sending the briefcase to the target agent and receiving results. The median cost measured for this operation was 346 microseconds.

Step 8 - Activate the TACOMA agent: This step initiates execution of the target agent and consists of two sub-steps. a) A vfork() (509 microseconds) is invoked, followed by an b) execl() (which returns after 2147 microseconds)[5]. The median cost measured for this step was 2656 microseconds and is independent of the size of the briefcase. Figure 6 shows the SDP for this step. The samples grouped around 14 miliseconds is probably caused by occasional cache misses and, consequently, accessing the disk.

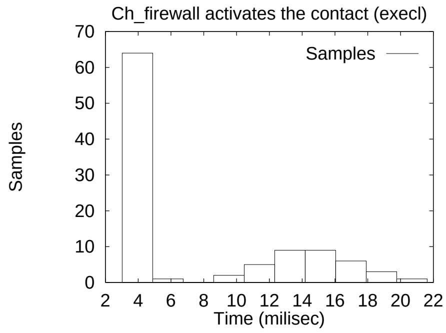


Fig. 6. The cost of activating the destination agent.

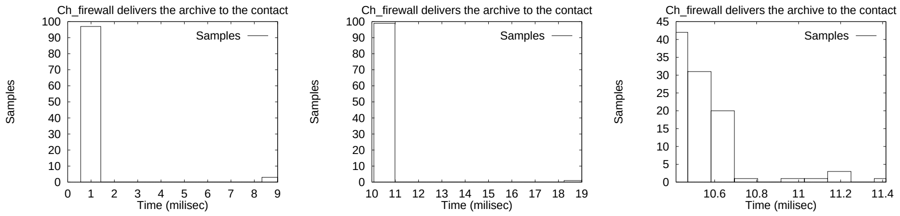
Step 9 - Transfer the briefcase: In this step, ch_firewall sends the marshaled briefcase to the target agent using one of the pipes created in step 7. This cost is independent of briefcase size as long as the briefcase is smaller than 8Kb and, therefore, fits in the pipe's kernel buffer. However, if the marshaled briefcase is larger than 8Kb, context switches are necessary between ch_firewall and the agent. Above 8Kb the overhead grows linearly.

For a minimal size briefcase, the median cost measured for this step is 133 microseconds. Figure 7 shows SDPs for three different sizes of briefcases, and Figure 8 gives the median cost for additional briefcase sizes.

Step 10 - Unmarshal and start agent: Here there are three sub-steps. a) First, completion of the step 8 target agent creation is awaited. If the target agent executable was not in the kernel file cache, the executable must be loaded from disk. When a cache hit occurs, which was the common case in our experiments, a delay of 8137 microseconds was measured.

b) The target agent receives the briefcase sent in step 9 from the pipe created in step 7. This seems to require 2535 microseconds, according to our measurements.

[5] Execution of the target agent starts at step 8, and occurs only after the execl() returns.



(a) 0k (b) 8k (c) 16k

Fig. 7. SDPs of pipe communication cost; three different briefcase sizes.

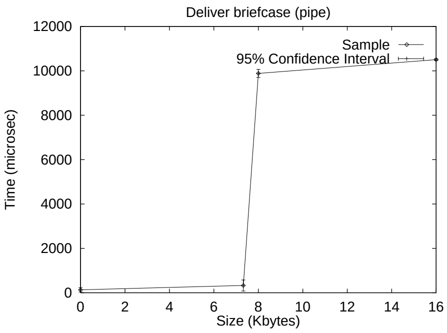


Fig. 8. The cost of delivering the marshaled briefcase to the agent.

c) The briefcase is unmarshalled and the target agent is invoked. We measured 606 microseconds for this step when there is a small briefcase. Figure 9 shows a graph of this cost as a function of briefcase sizes.

The median cost for this entire step is 11,278 microseconds.

3.4 Termination

Step 11 - Return status to ch_firewall: In this step, the target agent sends a status message through the pipe created in step 7 to ch_firewall. The median cost measured for this step was 2374 microseconds, but with a high variance (values ranging from 310 to 2458 microseconds were measured). We cannot account for the high variance and are investigating further.

Step 12 - Return status to client: In this step, ch_firewall returns the target agent's status message back to the initiating agent. The delay of this step was estimated by measuring the TCP round-trip time and halving it, obtaining 554 microseconds.

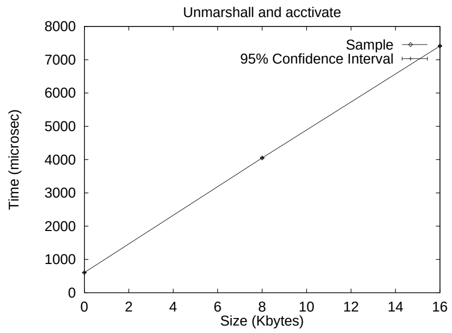


Fig. 9. Graph of unmarshalling cost.

Step 13 - Terminate: In this step, the TCP connection is closed at both ends. The median cost for this final operation was measured as 212 microseconds.

3.5 Total Cost

By summing the measurements given above, we calculate a median value of 21,675 microseconds (see Table 1). As a point of reference, a Sun RPC for the same data is 5.05 miliseconds. However, for such an RPC, the server's execution environment is static and set up in advance.

Table 1. Cost of steps in a meet (in microseconds).

Step	1	2	3a	3b	(4)	5a	5b	6	7	8a	8b	9	10a	10b	10c	11	12	13
Median	31	597	1678	97	57	1507	3	152	346	509	2147	133	8137	2535	606	2374	554	212

4 Discussion

This paper gives the delay associated with executing a meet in TACOMA. The delay results from setting up an environment for executing the code that was included by the source host. Our measurements suggest places where contemplating performance improvements is worthwhile. We now turn attention to those.

Steps 2, 3, and 7, which account for 2718 microseconds (approximately 12% of the total cost of a meet), can be improved by re-using resources rather than repeatedly allocating and freeing them. By keeping a cache of TCP connections to cached child firewall processes, the delay for steps 2, 3, and 7 is eliminated.

Step 9 and the second part of step 10 account for 2668 microseconds (approximately 12% of the total cost of a meet) also can be eliminated by (i) sending the briefcase directly from the initiating agent to the target agent and (ii) sending the status message directly back to the initiating agent.

Finally, the vfork() operation in step 8 that ch_firewall uses to set up a process can be removed from the critical path, saving another 509 microseconds. A real fork() would then have to be substituted, since now the child of ch_firewall will have to locate the agent executable, but since it will be out of the critical path this should not add to extra overhead.

The high cost of steps 5, 10, and 11 could be reduced if HP-UX provided better performance for transferring data between the kernel and user memory. We are investigating the current source of this poor performance.

An higher-level technique for improving performance of meet operations is to copy parts of a briefcase to the remote host on demand (i.e. lazily). Typically not all data in the briefcase will be needed at every host that an agent visits. This can be exploited to speed the activation of a target agent, sending data to that agent only when it is needed. After all, reducing data transfer is the *raison d'etre* for the mobile agent paradigm.

Acknowledgments

We would like to thank Fred B. Schneider for many discussions and comments on this paper.

References

[Gra95] R. S. Gray. Agent Tcl: A transportable agent system. Technical report, Dartmouth College, Hanover, New Hampshire 03755, November 1995.

[JvRS95] D. Johansen, R. van Renesse, and F. B. Schneider. Operating System Support for Mobile Agents. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HOTOS-V)*, pages 42–45. IEEE Press, May 1995.

[LFB96] M. B. Dillencourt L. F. Bic, M. Fukuda. Distributed Computing using Autonomous Objects. *IEEE Computer*, 29(8), August 1996.

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