NAP: Practical Fault-Tolerance for Itinerant Computations

Dag Johansen* Keith Marzullo† Fred B. Schneider‡ Kjetil Jacobsen*
Dmitrii Zagorodnov†

Abstract

NAP is a protocol for supporting fault-tolerance in intinerant computations. It employs a form of failure detection and recovery, and it generalizes the primary-backup approach to a new compuational model. The guarantees offered by NAP as well as an implementation for NAP in TACOMAare discussed.

1 Introduction

One use of mobile agents is support for *itinerant computation* [5]. An itinerant computation is a program that moves from host to host in a network. Which hosts the program visits is determined by the program. The program can have a pre-defined itinerary or can dynamically compute the next host to visit as it visits each successive host; it can visit the same host repeatedly or it can even create multiple concurrent copies of itself on a single host.

Itinerant computations are susceptible to processor failures, communications failures, and crashes due to program bugs. Prior work in fault-tolerance for itinerant computations has focused on the use of replication and masking. For example, [14] discusses a technique for replicating (on independently failing processors) the environment—herein called a *landing pad*—in which an itinerant computation executes. Thus, failures are masked below the landing pad and the programmer of an itinerant computation need not be concerned with handling them.

Replication and masking, however, has limitations because replication requires redundant processing, which is expensive. Furthermore, preserving the necessary consistency between replicas can be done efficiently only within a local-area network. Replication and masking approaches are also unable to tolerate program bugs. Thus, a fault-tolerance method based on failure detection and recovery seems the better choice when itinerant compuations must operate beyond a local area network and must employ potentially buggy software.

We present such a fault-tolerance method in this paper. It has roots in the primary-backup approach [1, 4], only with the fixed backup processors being replaced by mobile agents called *rear guards* [9]. With our method, a rear guard performs some recovery action and continues the itinerant computation after a failure is detected.

The key differences between our approach 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 landing pads where the itinerant computation recently executed. Much of what is novel about NAP stems from the need to orchestrate rear guards as the itinerant computation moves from host to host.

We call our protocol *NAP*.[1] The idea for such a protocol was first discussed in [9]. This paper fleshes out the idea, describing the TACOMA [8] landing-pad support for NAP and the guarantees that NAP can provide to programmers. We also discuss an actual Python-based implementation of NAP.

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

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

‡Department of Computer Science, Cornell University, Ithaca 14853-7501, New York, USA. Supported in part by ARPA/RADC grant F30602-96-1-0317 and AFOSR grant F49620-94-1-0198. The views and conclusions contained herein are those of the author 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.

[1] *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.

2 Assumptions

In TACOMA, an itinerant computation is structured from mobile agents. Each host in the network is assumed to run a landing pad; a mobile agent is started on host H by giving the landing pad at H the program text and the initial state of the agent.

A program running on a host can crash, and a host or landing pad can crash thereby crashing all programs running on that host or landing pad. When a mobile agent terminates as a result of one of these crashes, we say that execution of the agent has experienced a *fault*. We assume that a fault is eventually detected by one of more of a small, well-defined set of landing pads. This is equivalent to assuming the *fail-stop* failure model of [13].

Replication of data and control is what enables an itinerant computation to recover from faults. We can characterize how much replication is needed in terms of a parameter, f. One simple characterizaton is given by:

> **Bounded Crash Rate.** For any integer $0 \leq i \leq f$, there can be no more than i crashes of hosts or landing pads during the maximum period of time it takes the agent to traverse i distinct hosts.

This characterization is convenient because f remains fixed during the entire itinerant computation. However, a more practical characterization would have f depending on the host currently being visited (how reliable is it?) and on the current state of the itinerant computation. We use the Bounded Crash Rate characterization in this paper for expository simplicity; extending our protocols to more realistic characterizations is straightforward.

Finally, each pair of hosts in the network is assumed to be connected by a FIFO communications link that masks communications failures. In Section 6, we revisit this assumption and discuss how to adapt NAP to networks that can partition.

3 Fault-Tolerant Itinerant Computations in TACOMA

A TACOMA mobile agent can

- move to another host using a **move** operation, or
- continue executing on the current host and create a new agent on another host using a **spawn** operation.

This means that execution of a TACOMA mobile agent defines a sequence of *actions*, where a mobile agent executing its i^{th} action is said to be *version* i of that mobile agent. For a mobile agent a, we denote version i of this agent as $a[i]$.

In TACOMA, a fault during the execution of an action terminates that agent in an undefined state, except an option to TACOMA's **move** can specify that the interrupted action be re-executed when (and if) the affected landing pad restarts [8]. To accommodate the guarantess that NAP implements, we therefore now extend the definition of a TACOMA action along lines first proposed in connection with *fault-tolerant actions* [13].

A fault-tolerant action FTA

$$FTA\text{: } \textbf{action } A \textbf{ recovery } \overline{A}$$

where A is called a *regular action* and $\overline{A}$ is called the *recovery action* associated with A executes according to the following:

1. A executes at most once, either with or without failing.
2. If A fails, then $\overline{A}$ executes at least once and executes without failing exactly once.
3. $\overline{A}$ executes only if A fails.

An action *fails* if that action experiences a fault during its execution. A fault that occurs between the execution of two fault-tolerant actions is attributed to one or the other. So, it is possible for all of the user's code in A to execute, yet to have $\overline{A}$ also execute because a fault occurs just after A finishes. However, once a subsequent action A' starts executing, a fault will result in $\overline{A'}$ executing rather than $\overline{A}$ executing.

Fault-tolerant actions are general enough to program any kind of fault-tolerance scheme that is based on detection and recovery. For example, given an operation undo/redo mechanism [3], fault-tolerant actions can be used to implement atomic transactions.

The recovery action that an agent should take will most likely be changed when that agent moves or spawns a new agent. Hence, **move** and **spawn** both are defined as terminating an action.[2] For example, Figure 1 shows an itinerant computation originating with $a1[1]$. The second version of agent $a1$, $a1[2]$, starts when $a1[1]$ executes **move** naming host H2 and terminates by executing **spawn**. The **spawn** creates both the third version $a1[3]$ of $a1$, still on H2, and the third version $a2[3]$ of a new agent $a2$ (on H4). By convention, we define $a1[2]$ to be the second version of $a1$ and $a2$.

[2] A third operation, **checkpoint**, also terminates an action. This operation is described later in this section.

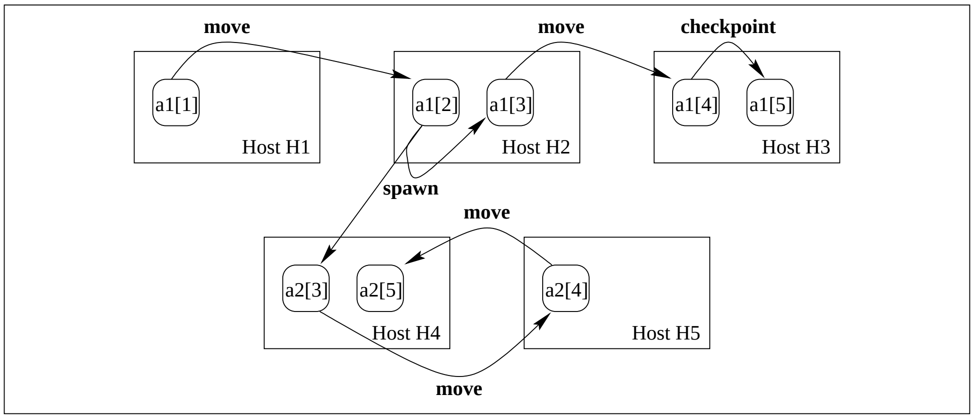

Figure 1: Versions of Mobile Agents

TACOMA agents can be written in many different languages, so fault-tolerant actions are encoded rather than being programmed using the syntax given above. For this encoding, the state of a TACOMA mobile agent is described in a data structure called a *briefcase*. A briefcase stores a named set of *folders*, $\langle name, value \rangle$ pairs; each of the names in the briefcase is unique. A TACOMAmobile agent's briefcase would have five folders associated with fault-tolerant actions and two additional folders associated with recovery actions. The purpose of these folders is summarized in Table 1.

The effect of **move** and **spawn** can be described operationally in terms of folders. For example, **move**(b) starts executing the program given as the head[3] of b.CODE at the landing pad named in the head of b.HOST[4]. This code starts executing as a regular action, and it is given a briefcase b' identical to b except that:

- b'.HOST is the tail of b.HOST.
- b'.CODE is the tail of b.CODE.
- b'.RECOVERY is the tail of b.RECOVERY.
- b'.VERSION is b.VERSION $+ 1$.

A fault during exection of a regular action invokes the associated recovery action, and a fault during execution of a recovery action causes that recovery action to be re-executed. With NAP, the recovery action executes on some landing pad that was recently visited by the itinerant computation. When a regular action executing with a briefcase b experiences a fault, the code for the recovery action is the head of b.RECOVERY. The briefcase b' this recovery action gets is identical to b except that two new folders are added:

- b''.RECOVERY_HOST is the identity of host upon which the recovery action is executing.
- b''.FAILURE_STATUS is information about the nature of the failure of the regular action.

A mobile agent can interact with its environment, and—at times—the mobile agent will need to change its recovery action for such interaction. For example, suppose a mobile agent finds some information on a host it is visiting, and because of this information the mobile agent decides to delete a local file. If this file should be deleted no matter how the local information changes, then the recovery action should change to ensure that the file is eventually deleted. This need to change recovery actions is a manifestation of the *output commit problem* [6]: before taking an irrevocable action, the mobile agent ensures that its current state is stable so that any recovery action will have the information that led to the irrevocable action and will be able to complete the action (even if the regular action was interrupted by a fault).

A third TACOMA operation, **checkpoint**, can be used to do ensure that saved state is stable, so that state is available to recovery actions. Figure 1, for example, shows version $a1[4]$ creating version $a1[5]$ by executing **checkpoint**. Operationally, **checkpoint**(b) is like **move**(b) but the new action *head*$(b$.CODE$)$ is executed at the current landing pad

[3]Given a list ℓ, the *head* of ℓ is the first element of the list and the *tail* of ℓ is the list with the head removed.

[4]The list of hosts b.HOST can be changed at any time, so the itinerary of a mobile agent changes under program control.

rather than at *head*(b.HOST) and, therefore, the implementation of **checkpoint** can be cheaper than implementing it directly with **move**.

Appendix A contains a TACOMA mobile agent that illustrates the implementation of fault-tolerant actions by use of the TACOMA**move**, **spawn**, and **checkpoint**operations.

4 Protocol

At a high level, implementing NAP is simple. Consider a regular action $a[i]$ executing at a landing pad L_i. When $a[i]$ terminates, the identity of the next landing pad L_{i+1} is the head of the HOST folder in current briefcase b. We can thus achieve the desired behavior for NAP if L_i uses a reliable broadcast protocol [7] to send b to a set $G(a[i])$ of landing pads, where the rear guards for $a[i]$ and the landing pad L_{i+1} are in $G(a[i])$. Reliable broadcast guarantees that all nonfaulty landing pads in $G(a[i])$ either deliver b or do not deliver b.

Three outcomes are possible from the reliable broadcast:

1. No landing pad delivers b. This implies that the landing pad L_i crashed. The recovery action $\overline{a[i]}$ should be executed by one of the rear guards in $G(a[i])$.

2. L_{i+1} delivers b. This implies that all nonfaulty landing pads in $G(a[i])$ have delivered b. The regular action $a[i+1]$ should thus begin to execute.

3. Some landing pad delivers b, but L_{i+1} does not. This implies that L_{i+1} crashed. A rear guard for $a[i+1]$ in $G(a[i])$ will determine this fact and execute the recovery action $\overline{a[i+1]}$.

4.1 Runtime Architecture

Each host has, in one process, a *landing pad* thread and a *failure detection* thread. The landing pad maintains a *NAP state* object that stores information about mobile agents the host is executing or for which the host serves as a rear guard. The landing pad thread informs the failure detection thread which landing pads to monitor. (See below.)

Each mobile agent at a host executes in its own process; that process is created by the host's landing pad and, therefore, the reliable broadcast is initiated when mobile agent process exits.

4.2 Reliable Broadcast

The reliable broadcast protocol we use for our implementation of NAP is a refinement of the one presented in [15], instantiated with a linear "broadcast strategy." Here is how that works.

Consider a process p_0 that broadcasts a value b to a group $G = \{p_0, p_1, \ldots, p_{n-1}\}$. For process p_0 to ensure that all nonfaulty processes in G either deliver b or do not deliver b, p_0 sends b to p_1 and waits for an acknowledgment from p_1. Process p_1, upon receipt of b from p_0, ensures that, assuming it does not fail, all nonfaulty processes in $G - \{p_0\}$ deliver b. In general, when p_i receives b it becomes responsible for ensuring that b is delivered by all nonfaulty processes in $G - \{p_0, p_1, \ldots, p_{i-1}\} = \{p_i, p_{i+1}, \ldots, p_{n-1}\}$. And when this obligation is discharged, p_i sends an acknowledgment to p_{i-1}. Thus, if there are no crashes, then message b will travel from p_0 to p_1 to p_2 and so on to p_{n-1}, and then the acknowledgment will travel back from p_{n-1} to p_{n-2} to p_{n-3} and so on back to p_0.

After p_i sends b to p_{i+1}, process p_i monitors p_{i+1} for a crash. If p_i detects p_{i+1}'s crash before receiving an acknowledgment from p_{i+1}, then p_i takes over the task of establishing that the nonfaulty processes in $\{p_{i+1}, p_{i+2}, \ldots, p_{n-1}\}$ deliver b. In particular, p_i sends b to p_{i+2} and waits for an acknowledgment from p_{i+2}. p_{i+2} sends the acknowledgment to p_i when it can. (For example, p_{i+2} can immediately send the acknowledgment if it had already sent an acknowledgment to p_{i-1}). If p_i detects p_{i+2}'s crash before receiving this acknowledgment, then p_i continues by sending b to p_{i+3}, and so on.

The reliable broadcast protocol in [15] also implements an election protocol: there is always eventually one process (initially p_0) that knows itself to be elected. A process remains elected until it fails. This is important when using arbitrary broadcast strategies, because if p_0 fails, then a process must take over to complete the broadcast. The election protocol used in NAP is as follows [3]:

1. Upon receiving b from p_{i-k}, process p_i monitors for the crash of p_{i-k}.

2. If while monitoring p_{i-k}, process p_i then detects the crash of p_{i-k} then p_i either monitors for the crash of p_{i-k-1} (if $k \neq i$) or it elects itself (if $k = i$).

4.3 NAP

NAP builds on the reliable broadcast protocol just given. Process p_ℓ in the reliable broadcast protocol is assigned to the landing pad $L_{i+1-\ell}$ that executed regular action $a[i+1-\ell]$. Two simple changes are:

1. By the Bounded Failure Rate assumption, once $f+1$ landing pads have b, then b cannot be lost due to crashes. Thus, once $f+1$ landing pads have b, it is safe for L_{i+1}. Therefore, once a landing

folder	*use*
HOST	list of hosts to be visited (head is the next host to visit)
CODE	list of regular actions (head is next to be executed)
RECOVERY	list of recovery actions (head is associated with this action)
VERSION	the version of the current action
NUM_GUARDS	minimum number of rear guards
RALLY_POINT	list of hosts to retreat to in case of disaster
RECOVERY_HOST	host on which recovery action is executing
FAILURE_STATUS	information regarding failure of regular action

Table 1: Folders relevant to Fault-Tolerant Actions

pad L determines that $f+1$ landing pads have received b (equivalently, that b.NUM_GUARDS rear guards have b), L sends a b *stable* message to L_{i+1}. L_{i+1} does not start executing $a[i+1]$ until it receives this message.

2. If a landing pad finds itself elected after having last received b, then it starts executing the recovery action $\overline{a[i]}$.

In the remainder of this section, we describe other changes. Appendix B gives the complete protocol in pseudocode.

Membership. One can think of NAP as a reliable broadcast protocol to a process group, where the group changes with each broadcast. The changes are determined by *membership rules*: $G(a[i])$ is defined to be $G(a[i-1])$ plus a set of landing pads that join $G(a[i])$ and minus a set of landing pads that leave $G(a[i])$. The only requirement on group membership that we require is that $G(a[i])$ include L_{i+1}.

Group $G(a[i])$ must contain at least $f+1$ members. Thus, any landing pad that receives b after $f+1$ landing pads have received b need not deliver b nor need be in $G(a[i+1])$. Since landing pads sequentially learn the number of landing pads that have b, we can use the following rule: when a landing pad receives b, if previously $f+1$ other landing pads have already delivered b, then it leaves $G(a[i])$. This rule is attractive, because it is simple to implement and has an intuitive appeal.

There are other plausible rules for choosing which landing pads leave $G(a[i])$. The *oldest* landing pads might be required to remain in $G(a[i])$, since they have not failed recently and thus appear to be stable. With this rule, the *latest* rear guard would drop out of $G(a[i])$ once it receives the acknowledgment that the broadcast of b is complete. More generally, landing pads could piggyback information with their NAP acknowledgments. The information, for example, might include performance measurements provided by the failure detection thread. L_{i+1} could use this information to determine which rear guard is introducing the most latency and therefore should leave $G(a[i+2])$. This rear guard's identity could be included in the broadcast of b_{i+1}.

One additional membership rule is required for when a mobile agent revisits a landing pad. That landing pad may find itself twice in the broadcast strategy. For example, consider agent $a2$ in Figure 1. If $f=3$, then $G(a2[5]) = \{H1, H2, H4, H5\}$ where $H4$ both precedes and follows $H5$ in the broadcast strategy. When this happens, the second entry is dropped from the broadcast strategy. For example, the broadcast to $G(a2[5])$ uses the broadcast strategy $H4; H5; H2; H1$.

Catastrophic Failure. Although not admitted by our failure model, in practice there will be situations (such as programming bugs) in which recovery action $\overline{a[i]}$ will fail repeatedly. All rear guards thus fail. A reasonable response for this case is to pass the briefcase b of the failing agent to a well-known host; we call this host the *rally point*. The identity of the rally point is specified in the RALLY_POINT folder.

One implementation of would have rally point p_{rp} be a member of the group $G(a[i])$ for each version i, and to have p_{rp} take over should it detect all of the other members of the group as having crashed. A more efficient implementation is to have at least $f+1$ rather than f rear guards. If a rear guard finds that all other rear guards have failed, it passes the briefcase to p_{rp}.

Termination. When a mobile agent terminates, the NAP for this agent must also terminate. Surprisingly, even though the reliable broadcast protocol that NAP is based on cannot terminate [15], orchestrating termination of NAP is straightforward. The TACOMA operation **exit** is a command that instructs a landing pad to terminate support for the corresponding mobile agent. Suppose the last user-defined action of some mobile agent is:

FTA_ω: **action** A_ω **recovery** $\overline{A_\omega}$

To orchestrate termination of NAP, FTA_ω can be then replaced by two actions:

action { A_ω; **checkpoint**} **recovery** $\overline{A_\omega}$;
action exit recovery exit;

When the last landing pad executes **exit**, it will appear to have crashed, resulting in a failure detection[5] The election protocol in NAP will then choose a rear guard to execute the recovery action. The agent that executes the recovery action will then terminate executing NAP, causing another failure detection and another rear guard executing the recovery action. This will continue until all rear guards have terminated executing NAP for this program.

When a rally point is defined, this termination protocol will pass the final briefcase b_ω to b_ω.RALLY_POINT. Hence, all executions end up at the rally point at termination. The reason for termination (abnormal or regular) can be recorded in the final briefcase b_ω.

Reducing Latency. Using a linear broadcast strategy yields a simple protocol, but has the worst latency of all broadcast strategies. With a linear broadcast strategy, before a version of a mobile agent can start executing, a chain of $f+1$ messages must be sent and received. As we show in Section 5, for a **move** operation and for reasonably small values of f, the latency of the reliable broadcast is subsumed by the latency of initializing the new agent version, but for **spawn** and **checkpoint** the latency can be significant.

For **spawn** and **checkpoint**, optimistic execution can mask some of the latency imposed by the reliable broadcast. Instead of blocking the execution of a new mobile agent version $a[i+1]$ until a "*b stable*" message is delivered locally, $a[i+1]$ starts executing as soon as possible. This creates the danger that crashes may cause $\overline{a[i]}$ to be executed after user code associated with $a[i+1]$ starts executing. If this does pose a problem, then $a[i+1]$ can use the TACOMA **wait_stable** operation to block until b has been delivered by at least $f+1$ landing pads. If $a[i+1]$ does not explicitly execute **wait_stable**, then **wait_stable**is implicitly executed at the end of $a[i+1]$.

[5]Tthe failure detection latency can be reduced by sending an explicit message indicating that the landing pad is terminating.

An illustration of this optimization appears in Appendix A.

5 Implementation

We have implemented NAP in a Python-based[6] version of TACOMA. We chose Python because it is a convenient language for prototyping. Of primary concern was deciding how we would integrate NAP into the existing TACOMA architecture. The performance of this first version of NAP in TACOMAwas of less importance

The cost of doing a **move** with NAP are given in Table 2. These values were obtained on a system comprising Pentium Pro processors with 200 MHz clocks. Each machine had 128MB of RAM and 100MB Ethernet. Each was running FreeBSD 2.2.7. To compute each value in Table 2, 100 measurements were made; the standard deviation was within 5 percent of the averages.

A least-squares fit to these values gives the cost of a **move** given g rear guards as $51.6 + 87.5g$ msec. We expect to be able to reduce this cost significantly.

number of rear guards	0	1	2	3	4
time (msec)	54	138	235	311	405

Table 2: Cost of NAP as a function of number of rear guards

6 Conclusions

NAP provides fault-tolerance for itinerant compuations at low cost. The replication needed for fault-tolerance is obtained by leaving some code running at landing pads 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. Thus, when a low cost method of recovery is possible, the programmer can use that method (rather than, for example, active replication [14] or primary-backup [12]). We believe that this flexibility is especially important when partitioning is possible.

NAP is based on a reliable broadcast that uses a linear broadcast strategy. A linear broadcast strategy results in a simple rule for determining when a landing pad should be dropped from the rear guards. For small values of f, the latency of NAP is subsumed by the cost of a **move**, the most common method of terminating a regular action. The latency is not subsumed

[6]A reference manual for Python can be found at `http://www.python.org/doc/ref/`.

by the cost of a **spawn**, though. The latency could be reduced by using a broadcast strategy with a larger fanout than our linear broadcast strategy. We are examining versions of NAP built using such broadcast strategies for itinerant computations that frequently use **spawn** and **checkpoint**.

NAP, as presented here, cannot be implemented in a system that can experience partitions, because no crash failure-detector can be implemented in such a system. However, in systems that can partition, processes within the same partition can agree on which processes are unreachable (even though they cannot distinguish between the case of the unreachable process being crashed or being partitioned away [16]). With such a failure detector, a network partitioning into two connected components may lead to a regular action and its recovery action both executing without failing.

We are currently designing a version of NAP that provides better support for partitioned operation. The failure detection thread for this version is as described above: it implements consistent detection within a set of connected landing pads of the unreachability of the other landing pads. This version also has a set of tools that aid the TACOMA programmer in writing a TACOMAmobile agent that executes in a partitionable environment. For example, TACOMA already provides a mechanism for the transactional update of collections of folders on stable storage. We plan to use this mechanism to allow applications to have the same measure of fault-tolerance that, for example, the protocol of [12] gives. It will also allow for applications more demanding than those supported by [12], such as those for which a transaction spans many landing pads. For mobile agents that do not require such strict semantics, we will have tools that provide information on the network's topology and current performance. Such tools allow one to write "partition-aware" [2] mobile agents. The mobile agent described in Appendix A is one that we believe would fit well into this second class of applications.

Acknowledgements We would like to thank the other members of the TACOMA researh group and the anonymous referees for insightful comments on earlier versions of the paper.

References

[1] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the Second International Conference on Software Engineering*, San Francisco, California, USA, 13-15 October 1976, pp. 627–644.

[2] O. Babaoglu, R. Davoli, A. Montresor, and R. Segala. System support for partition-aware network applications. In *Proceedings of the Eighteenth International Conference on Distributed Computing Systems*, Amsterdam, The Netherlands, 26-29 May 1998, pages 184-191.

[3] Philip A. Bernstein, Nathan Goodman, and Vassos Hadzilacos. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley 1987.

[4] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Primary-backup protocols: lower bounds and optimal implementations. in *Proceedings of the Third IFIP Working Conference on Dependable Computing for Critical Applications (DCCA-3)*, Mondello, Italy, 14-16 September 1992, pp. 321–343.

[5] David Chess, Benjamin Grosof, Colin Harrison, David Levine, Colin Parris, and Gene Tsudik. Itinerant Agents for Mobile Computing. *IEEE Personal Communications* 2(5):34-49, October 1995.

[6] E. N. Elnozahy and W. Zwaenepoel. On the use and implementation of message logging. In *Digest of Papers, The Twenty-Fourth International Symposium on Fault-Tolerant Computing*, Austin, TX, USA, 15-17 June 1994, pp. 298-307.

[7] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In *Distributed Systems*, Second Edition, Sape Mullender editor, ACM Press Frontier Series, Addison-Welsey 1993.

[8] Dag Johansen, Robbert van Renesse and Fred B. Schneider. An Introduction to TACOMA distributed system version 1.0. University of Tromsø Department of Computer Science Technical Report 95-23, June 1995.

[9] Dag Johansen, Robbert van Renesse and F. B. Schneider. Operating systems support for mobile agents. In *Proceedings of the Fifth IEEE Workshop on Hot Topics in Operating Systems*, Orcas Island, Wahsington, USA, 4–5 May 1995, pp. 42–45.

[10] Friedmann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms* (M. Cosnard et. al. editor), Elsevir Science Publishers B. V. 1989, pp. 215–226.

[11] G. van Rossum and J. de Boer. Linking a stub generator (AIL) to a prototyping language (Python). In *Proceedings of the Spring 1991 EurOpen Conference*, Tromsø, Norway, 20-24 May 1991, pp. 229-247.

[12] Kurt Rothermel and Markus Straßer. A fault-tolerant protocol for providing the exactly-once property of mobile agents. In *Proceedings of the Seventeenth IEEE Symposium on Reliable Distributed Systems*, 20-23 October 1998, pp. 100-108.

[13] R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems* 1(3):222-238, August 1983.

[14] F. B. Schneider. Towards fault-tolerant and secure agentry. In *Proceedings of the Eleventh Workshop on Distributed Algorithms*, Saarbrucken, Germany, 24–26 September 1997, pp. 1-14.

[15] F. B. Schneider, D. Gries, and R. D. Schlichting. Fault-tolerant broadcasts. *Science of Computer Programming* 4(1):1–15, April 1984.

[16] J. Sussman and K. Marzullo. The *Bancomat* problem: AN example of resource allocation in a partitionable asynchronous system. In *Proceedings of DISC'98: Twelfth International Symposium on Distributed Computing*, 23-25 September 1998, Andros, Greece, pp 363–377.

A Example: License Checker

The following description of a TACOMA mobile agent illustrates the programming and use of fault-tolerant actions. The mobile agent visits a set of hosts, specified as a parameter. For each host visited, the mobile agent creates a folder that describes the action the agent took there or whether it found the host to be unavailable. This folder is returned to the originating host.

The mobile agent takes the following actions for each host it visits:

- If file `license` exists and contains the word "customer", then the mobile agent renames the file `program` to `old_program` and writes a new file `program`.
- If file `license` exists and contains the word "demo", then the mobile agent takes no action.
- Otherwise, the mobile agent deletes the file `program`.

We wish the agent to update the host with some care, however. In the unlikely (or perhaps maliciously orchestrated) event that the host crashes while the changes are taking place, we would like to notify the user who launched the agent.

The agent executes consists of the five fault-tolerant actions: **launch**, **visit**, **update**, **alert**, and **report**. It is started by executing **launch** on a host that we call the *originating host*. We assume that the originating host does not crash (but it is easy to rewrite this program to use a set of backup hosts should one wish to tolerate failures of the originating host).

The five actions are:

1. **launch** This action executes **move** of action **visit** to the first host, if there is such a host.

   There is no recovery action for this first action; there is no rear guard yet defined that will execute it.

2. **visit** This action determines the action to take based on the license file found on the host being visited. A folder is created having the name of the host; the folder records the action to be taken on this host. The action terminates with a **checkpoint**, causing action **update** to execute.

   The recovery action creates a folder with the name of the host and records the fact that this host was not available. The recovery action terminates by executing **move** for the action **visit** naming the next host, if there is another host to visit. Otherwise, the recovery action terminatesby executing **move** for the action **report** back to the originating host.

3. **update** This action updates the files in accordance with the contents of the host's folder. It records this fact in the host's folder. The action terminates with a **move** of the action **visit** to the next host if there is another host to visit. Otherwise, it terminates with a **move** of the action **report** to the originating host.

   The recovery action records in the host's folder that the host failed before the action completed. The action terminates by executing **move** for action **visit**, naming the next host, and by executing **spawn** for action **alert** naming the originating host, if there is a next host to visit. Otherwise, it terminates with a **move** of the action **report** to the originating host.

4. **alert** This action writes a message indicating that a host crashed while its file system was being updated. The action terminates with **exit**. The recovery action is **exit**.

5. **report** This action writes the current contents of the briefcase to a well-known place. The recovery action does the same thing. Both actions terminate with **exit**.

To use the optimistic method for reducing latency as described in Section 4.3, all but **visit** can be executed without using the **wait_stable** operation. If **wait_stable** were not used at the beginning of the action **visit**, then it would be possible that, due to a set of failures, both the file system of the host would be updated and the recovery action of the preceding **visit** action would record that the host was not visited because it was crashed.

B NAP

We now specify NAP as an automaton executed by each landing pad.

Each briefcase `BC` has a unique identifier `BC.ID` that is set when the briefcase is created. `BC.ID` does not change value when the briefcase is passed to another landing pad.

A **spawn** operation is initiated by having the exiting mobile agent give its landing pad two briefcases: one for the newly-spawning agent and one for the continuing agent. A **move** operation is initiated by having the exiting application mobile agent give only one non-NULL briefcase. A **spawn** results in two concurrent reliable broadcasts, while a **move** results in only one reliable broadcast. Although not described above, a **spawn** can have the continuing agent and the newly-spawned agent each execute on different hosts.

A landing pad maintains a table `NAPstate` that maps a briefcase identifier to the following information:

- The version of the agent `active` that the landing pad believes is being executed;
- The version numbef for the agent `me` that was last executed at this landing pad;
- The landing pad's vector clock `VC` that is associated with this agent.

The vector clock is a table that maps a version `i` of the agent to the host on which it executed `VC[i].host` and the version of the briefcase that this landing pad believes is stored there `VC[i].vers`. The host can either be a host identifier or the value `UNKNOWN`. The version can be either a number, the value `NONE` indicating that the agent has not executed at this landing pad, or the value `DOWN` indicating that the landing pad has either crashed or otherwise garbage collected information concerning this briefcase. The value `VC.vers` can be thought of as a vector clock of unbounded length where the values of `VC[i].vers` for versions that have not yet executed are set to `NONE`, and the values for versions that have been garbage collected are set to `DOWN`. Hence, only a bounded set of values need to be maintained in `VC[i].vers`. The `vers` component of `VC` is treated like any vector clock [10] where `NONE` less than any integer and `DOWN` is greater than any integer.

To keep the pseudocode for the protocol as short as possible without losing its essential structure, we do not describe initial agent startup, agent termination or keeping additional rear guards for when the number of rear guards drops too low then moving the agent to one of the hosts specified in `BC.rally_point`.

```
catch agent_termination(sBC, mBC):
  open e: NAPstate[mBC.ID] {
  Host mh = head(mBC.host);
  wait until (stable(e));
  active = me+1;
  VC[active].host = head(mBC.host);
  VC[active].vers = NONE;
  mBC.host = tail(mBC.host);
  send <"move", VC, mBC, active> to mh;
   if (sBC != NULL)
   open es: NAPstate[sBC.ID] {
     Host sh = head(sBC.host);
     es.active = active;
     es.me = me;
     es.VC = VC;
     es.VC[active].host =
       head(sBC.host);
       sBC.host = tail(sBC.host);
     send <"move", es.VC, es.sBC, es.active>
     to sh);
     }
  }

catch failure_detect(host):
  for each entry e in NAPstate {
    if (host == MyChild(e)) {
      e.VC[next(e, e.me)].vers = DEAD;
      DoUpdate(e.BC.ID, e.me); }
    if (host == MyParent(e)) {
      e.VC[prev(e)].host = DEAD;
      if (host == e.VC[active].host) {
        wait until (stable(e));
        fork recovery agent (e.BC);
      }
      else DoAck(e.BC.ID);
    }
  }
```

```
receive move(newVC, newBC, newActive):
  open NAPstate[newBC.ID] {
  updateVC(newBC.ID, newVC);
  me = active = newActive;
  BC = newBC;
  fork new agent (BC);
  DoUpdate(BC.ID, me);
  }

receive BC_stable(BC_ID):
  open NAPstate[BC_ID] {
    note briefcase stable;
  }

receive update(newVC, newBC, vers, i):
  open NAPstate[newBC.ID] {
  if (active < vers) {
    UpdateVC(newBC.ID, newVC);
    if (VC[i].host == VC[me].host)
      VC[i].vers = DOWN;
    else {
      BC = newBC;
      active = VC[me].vers = vers;
    }
  DoUpdate(BC.ID, i);
  }
  else DoAck(BC.ID);
}

receive ack(BC_ID, newVC):
  open NAPstate(BC_ID) {
  UpdateVC(BC_ID, newVC);
  DoAck(BC.ID);
  }

void UpdateVC(BC_ID, newVC):
  open NAPstate[BC_ID] {
    for all entries i of newVC:
      newVC[i].vers > VC[i].vers {
      VC[i].vers = newVC[i].vers;
      VC[i].host = newVC[i].host;
    }
  }

void DoUpdate(BC_ID, i):
  open e: NAPstate[BC_ID] {
  if (overstable(e)) VC[i].vers = DOWN;
  else if (stable(e)) {
    send <"BC_stable", BC.ID>
      to VC[active].host;
    }
  if (next(e, i) == i) DoAck(BC_ID);
  else if (VC[next(e, e.me)].vers
          < active)
  send <"Update", VC, BC, active,
    next(e, i)>
      to VC[next(e, i)].host;
  }

void DoAck(BC_ID):
  open e: NAPstate[BC_ID] {
  if (prev(e) != me &&
    VC[prev(e)].vers < active)
    send <"Ack", BC_ID, VC>
      to VC[prev(e)].host;
  }

index next(e, j) {
 return largest index i < j;
   e.VC[i].vers is a number
 else return j;
 }

index prev(e) {
 return smallest index i > e.me:
   e.VC[i].vers is a number;
 else if (e.VC[i].host != UNKNOWN)
   return e.active;
 else return e.me;
 }

host MyChild(e) {
  return e.VC[next(e, e.me)].host; }

host MyParent(e) {
  return e.VC[prev(e)].host; }

boolean stable(e) {
 return (number of entries in e.VC[*].vers
  that equal e.active >=
    e.BC.num_guards
  || next(e, e.me) == me);
  }

boolean overstable(e) {
 return (number of entries in e.VC[*].vers
  that equal e.active >
    e.BC.num_guards);
  }
```