UNIVERSITY OF TROMSØ UIT

Faculty of Science and Technology
Department of Computer Science

Practical Fault-Tolerance for Mobile Agents

Kjetil Jacobsen

A dissertation for the degree of
Philosophiae Doctor

October 2011

Abstract

The amount of computational resources available on the Internet is increasing. Effectively using these resources for distributed computations is challenging. An infrastructure called *computational grids* provides tools for structuring and deploying large-scale distributed computations on the Internet. One of the key problems in computational grids is managing the available computational resources; tools based on mobile agents are being advocated to solve this problem. However, to be widely adopted, such tools must be robust towards failures in the grid environment, and thus require effective mechanisms for mobile agent fault-tolerance.

To gain insight on how grid applications perform on the Internet, this dissertation investigates two master-worker algorithms, one based on group communication and one based on message flooding. Both algorithms are executed in simulations using Internet communication traces. The results from running and evaluating the algorithms are used to infer requirements for our mobile agent fault-tolerance approach. This dissertation then derives a fault-tolerant mobile agent protocol. The protocol is rooted in the primary-backup approach, where a set of backups monitor the progress of the mobile agent during the computation. The protocol allows the set of backups to be changed during the computation to adapt to the current network topology. The dissertation then describes an implementation of our protocol on top of a mobile agent platform, and evaluates the performance of the protocol. The results show that explicit management of backups can be beneficial to performance, and that our protocol is applicable outside the scope of mobile agent computations.

Acknowledgements

I would first like to thank my advisors Dag Johansen and Keith Marzullo. I am indebted to Dag for fuelling me with inspiration, motivation, and ideas, and I'm very happy with the results that have come from our collaboration so far. I am also thankful to Dag for the possibility of working with Keith Marzullo. The work on the NAP and WAMW protocols with Keith has been very rewarding and a great learning experience for me. I would also like to thank Fred Schneider and Robbert van Renesse at Cornell University for hosting numerous productive sessions on problems and solutions related to the TACOMA project, and NAP in particular.

Some of the former TACOMA members and affiliates deserve significant credit: Nils Sudmann for the fruitful discussions and inspiration during the entire course of the TACOMA project. Kåre Jørgen Lauvset for all the lessons in practical software design. Xianan Zhang for the collaboration on the WAMW paper. Krister Mikalsen and Sveinar Rasmussen for geeking me up whenever I am getting outdated, and for the great times both before and after April 2000. Håkon Brugård for the enjoyable collaboration on the Extensible File System, DHash and numerous other projects. Steffen Viken Valvåg for refining and implementing my half-baked ideas with POSH. Åge Kvalnes for all the crazy projects we have worked on, and for all the great food. Alexander Wilkens for all the interesting discussions throughout the years. Håvard Johansen for his valuable input and comments on this dissertation.

Thanks to all that in some way influenced the outcome of this dissertation without knowing — friends, family, and colleagues at the University of Tromsø. Also a big thank you to the Norwegian Research Council (NSF) who generously funded this work through grant no. 112578/431 (the DITS program). The Department of Computer Science at the University of Tromsø smoothly provided me with all the infrastructure required to do the work on this dissertation. A particular thanks to Jon Ivar Kristiansen for helping me with the test environment.

I would like to direct a special gratitude to my girlfriend Gøril and my sons Øystein and Sverre for their faithful love, patience, and support.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The number of computers connected to the Internet with a broadband connection is still growing rapidly [80]. Such computers provide a powerful foundation for new distributed applications. As an early example of such an application, *file-sharing* applications emerged with the infamous Napster in 1999 [70]. Napster is a client-server application where the server maintains a centralized index that maps media files to the user computers where the file is located. Users register the files they want to share in this central index, allowing other users to search and download media. In contrast to Napster, most file-sharing applications today act as both clients and servers in a so-called *peer-to-peer* fashion. The main advantage of the peer-to-peer file-sharing applications such as BitTorrent [75] and Gnutella [34], is that they operate without central servers. The decentralized nature increases the availability of the service.

The success of file-sharing applications shows that the idea of resource sharing among computers on the edge of the Internet has great potential for other kinds of distributed application paradigms. One of the paradigms for sharing other resources than files is applications for sharing CPU-cycles. In 1999, an alternative way of running massively distributed computations on Internet computers emerged with the SETI@home project [5]. The purpose of the SETI@home project is to detect intelligent life outside Earth by analyzing radio signals from outer space using data from a telescope. The analysis process requires a significant amount of digital processing, which can be performed as a distributed computation. The approach for distributing the computation exploits the fact that most computers connected to the Internet run a screen-saver program when the machine is idle. Here, rather than running a regular graphical screen-saver, users can share their CPU cycles for research by downloading and installing a specific SETI screen-saver that participates in the distributed computation when the screen-saver program normally runs. The concept of harvesting idle cycles this way has captured millions of contributing users and has motivated numerous similar distributed applications, for instance the protein folding research program called Folding@home [108], where PlayStation3 owners can contribute cycles from their game console to help cure diseases such as Alzheimer's and Parkinson's.

Although distributed computations like SETI@home and Folding@home have been proven successful, they have limitations. Since the resources these computations employ are centrally coordinated and maintained in research facilities or corporations [55], users other than the initiators of the projects are unable to harvest from the farm of contributed resources. Thus, this approach does not offer a general-purpose environment for users that wish to run their own distributed computation. However, for such requirements, an alternative resource sharing model exists in the more general approach of *grid computing* [58].

1.1 Grid Computing

Grid computing is a computation model where users and organizations, called *providers*, donate resources for the benefit of running distributed computations and applications submitted by *consumers*. The resources together constitute what is known as the *grid*. In some grids, providers cannot be consumers, but in most cases providers may freely use grid resources. Unlike the resource requirements prevailed for instance by file-sharing peer-to-peer networks, grid computation resources are not just related to the requirements of file transfer. Rather, direct access to computers, the software running them, storage, network and all other associated resources are assumed. The main purpose of the grid technology is to facilitate access to these resources through standardized protocols and interfaces.

The first grid computing applications emerged from scientific computing [72, 166], large-scale distributed computing [19, 56], and collaborative environments [169]. Such grid computing applications usually execute on computers located in several geographically dispersed organizations. Hence, grid computing applications are typically not restricted to a single administrative domain. To handle this, grid consumers are given the required credentials to access the grid resources through some sort of service level agreement granted by the grid providers. The grid providers and consumers which adhere to this agreement constitute so-called *virtual organizations* [58]. The implementation of a virtual organization requires a means of secure authentication to control which users are allowed to use the grid. Several approaches exist — some grid computing system rely on existing authentication methods such as *secure shells* [16], while others offer their own infrastructure [94]. As an example, consider the procedure for providing to or consuming the resources available on the PlanetLab grid [137]. First, the organization wanting to provide or consume resources appoints a person to join the PlanetLab Consortium. A principal investigator is then appointed for the organization, and is responsible for mediating PlanetLab resource access to the users within the organization through secure shell credentials.

A commonly used approach to execute computations on the grid is the *master-worker* computation [162]. In a master-worker computation, a master process splits a program into independent tasks, and distributes the tasks across a set of worker processes. The parallelization gained from a master-worker computation may cause the time a program takes to execute to be reduced significantly. However, since the same resources are shared between several consumers, the performance improvement of this parallelization depends on the ability of the grid to do resource management. In fact, global management of grid resources is one of the key issues for scheduling an effective grid computation [96, 97]. The challenge stems from resource variation: grid computers may come and go, large parts of the grid may become unavailable because of network failures, and the available resources on a grid computer may vary.

A distributed computation paradigm called *mobile agents*, which emerged more than a decade ago, has resurfaced in the context of grid computing [24, 57, 61, 62, 78]. In essence, mobile agents are computer programs with the ability to explicitly move with their state from one computer to another using the network as carrier. The ability to move allows the mobile agent to *adapt* according to the available resources in its environment. For instance, if the mobile agent requires more CPU than available on the computer it is currently executing on, the agent may choose to move to another computer. If the agent requires frequent data access across a long-haul network link, the agent may choose to move itself closer to the data to avoid crossing the long-haul link when accessing the data. The ability to adapt upon resource changes in the environment makes mobile agents an attractive alternative to client-server interaction for finding, aggregating and retrieving information on the Internet. With this in mind, there are several properties of mobile agents that can be leveraged in grid environments, for instance:

- Searching the grid for a particular type of resource during the computation or before the computation starts [97, 178].
- Monitoring the progress of grid applications [78].
- Moving the grid computation executing at one computer to another computer when there are lack of local resources [61].
- Upgrading and installing software required by the grid infrastructure or the grid computation itself [178].

There are problems with deploying mobile agents on the Internet. One of the main problems is how to establish a security model that satisfies both the requirements of the users that will be hosting a mobile agent computation and the computation itself. For instance, consider the security model offered by the Java Virtual Machine (JVM)[1]for executing foreign code in web-browsers — although appropriate for applications in web-clients, the default security model enforced for Java Applets is too restrictive for a mobile agent application that needs to aggregate information from computers on the grid. Nevertheless, from a security perspective, the deployment of mobile agents is less of a problem in virtual organizations than on the Internet, since service level agreements for resource access within the virtual organization already exist [137].

Security issues aside, another major challenge that faces the adoption of mobile agents in grid environments, is how to make mobile agent computations robust enough to sustain the life-cycle of potentially long-running grid applications [57].

[1]Java homepage: http://sun.java.com

1.2 Research Problems

Grid computing takes place on the Internet, and is thus subject to constant changes in the available resources. If mobile agents are deployed in grid environments, they will be subject to the same changes in available resources. Hence, to be *robust*, mobile agent systems must implement mechanisms which enable them to tolerate failures that are typical in grid environments, in particular:

- *Crash failures*, where a computer fails, or a program executing on a computer as part of the computation fails or is forcibly pre-empted from the computer.
- *Network failures*, where the network link between two or more computers fails because of problems in the network infrastructure.

Since mobile agents are subject to running within the contract of a virtual organization, we assume that failures are not intended by the program or *Byzantine* [105]. There are generally two approaches to implementing robustness towards failures in distributed systems [9]: i) *preventing* failures by ensuring the integrity of the program and the host environment where the program will be executing, and ii) *tolerating* failures by ensuring that a computation is able to progress despite program failures.

1.2.1 Robustness by Preventing Failures

Crash failures can happen when the computer or the runtime environment where the program executes fails. Mobile agents and grid computations are based on programs submitted into the environment by users — if the program has a bug that causes it to crash the host environment, the computation fails. Significant research efforts [129, 181] have been undertaken to ensure that programs such as mobile agents cannot compromise the host computer and the environment they roam — in other words ensure *host integrity* during execution of the program.

One approach to control whether a program will cause the runtime environment to fail is to execute the program in a confined environment. Within this environment, all resource access is governed by a set of security policies enforced by a reference monitor in the runtime environment [6]. By doing so, an intentionally malicious or bugged program cannot cause crashes that render the computer useless. One of the software based approaches to confined environments is sandboxes [181], a mechanism for trapping errors caused by illegal memory accesses. More recent examples of sandboxing are programming language runtime environments like the JVM or the .Net Common Language Runtime[2], that ensure memory safety for programs using the runtime. In addition to memory safety, these virtual machines allow high-level application-specific security policies to be enforced, for instance that no network connection can be made after a file has been read. Such policies are useful to restrict access to resources that can cause the computer to crash upon misuse. Although they do not offer specific interfaces for expressing security policies, virtualization of hardware through systems like

[2].Net homepage: http://www.microsoft.com/net

VMWare [172] and Xen [15] allows programs to be run without being able to compromise the host system.

A problem with too restrictive security mechanisms, like the ones used when executing Java Applets, is that the policy may conflict with the resource requirements of the programs running on the grid. Programs executing on the grid may require unrestricted access to all resources on the grid computer. A security policy that denies access to for instance stable storage may thus be in conflict with the requirements of the grid program. Here, using virtualization through hardware or a memory safe language environment would be more appropriate since it allows programs to execute in an environment similar to the host environment with respect to resources. Given appropriate vertical resource management across the virtualized computers running on the same host, several computations would be able to coexist on the same host with some notion of resource fairness.

An alternative to the use of sandboxing is to verify that the program is benign by using techniques such as proof-carrying code [129]. The idea is that the environment for executing a program has established a set of rules that guarantees safe execution of the program. The author of the program gives a formal proof along with the program that can be validated against the safety rules of the execution environment. The advantage of this approach is that the execution environment can be made simple and fast. The disadvantage is that expressing the formal proof is difficult and would in practice place an unwanted burden upon the author of the program.

Regardless of whether a host integrity mechanism could be employed on a grid environment, the mechanism would not be able to prevent network failures because of problems in the network infrastructure. Network failures can cause the operation of moving an agent from one computer to another to fail. In addition, although virtualization and sandboxing may provide safe environments for executing programs, they cannot prevent failures or security holes in the software or hardware that implements the virtualization. Hence, given the inability to prevent network failures and program crashes, we require a mechanism to *tolerate* the failure of a mobile agent.

1.2.2 Robustness by Tolerating Failures

If program failures cannot be prevented, we need to utilize mechanisms for tolerating failures. In the field of fault-tolerance, such mechanisms require some means of *replication*, which in the case of mobile agents means replicating the mobile agent program to mask the effect of a failure. Replicating a service in a distributed system requires consistency among the replicas. Consistency is ensured by a replication protocol [73]. There are two major approaches to replication that ensure consistency:

- *Replicated state machine approach*, where identical instances of the program execute in parallel [158].

- *Primary-backup approach*, where a single instance of the program is executed by the *primary* and one or more identical instances, or *backups*, are ready to recover the program if the primary fails [28].

In the context of this dissertation, a distributed computation is defined as *fault-tolerant* if it tolerates crash failures and network failures, either through the use of a replicated state machine approach or a primary-backup approach. We now outline how crash and network failures are masked by a fault-tolerant system.

1.2.3 Masking Failures

Primary-backup approaches typically use a *failure-detector* [33] that periodically sends probe messages to other programs to establish whether these have crashed. A program is detected as crashed if it does not reply to probe messages in a timely manner. Recovery of the crashed program is done by rolling the system back or forward to a consistent state.

A problem with failure detectors is that they cannot reliably detect failures when the environment they execute in is subject to asynchrony [33]. Asynchrony can occur when a network failure causes parts of the network to be isolated, or when a program executes too slowly to respond to probe messages from a failure detector in a timely manner. This means that backups in a primary-backup system can falsely detect a primary as crashed. A false detection of a crash can lead to a redundant invocation of the recovery procedure, resulting in two primaries.

The replicated state machine approach executes multiple instances of the program in parallel. As long as there is a majority of programs within a network of connected computers, the computation makes progress even when some computers cannot communicate with the rest. The fundamental problem with replicated state machine approaches, however, is that they waste resources in failure-free runs because of the execution of multiple program instances in parallel. In grid environments where computations may execute for a long time and resources are shared among consumers, replicated state machine approaches are undesirable and primary-backup approaches are likely to consume less resources. This is one of the reasons why many grid environments use primary-backup approaches with rollback recovery for fault-tolerance [42].

1.3 Thesis Statement

Mobile agent computations have several benefits that can be leveraged in grid environments, for instance the ability to adapt upon changes in the network topology. Operating in this environment, however, exposes mobile agent computations to failures that must be tolerated. Thus, in this dissertation we investigate different approaches to providing fault-tolerance for mobile agent computations executing in grid environments.

A number of mobile agent systems have addressed the issue of fault-tolerance in the presence of crash and network failures [41, 126, 128, 140, 151, 164, 165]. An important problem is striking a balance between blocking the computation to enforce *exactly-once* [141] execution of computation and at the same time ensuring the computation makes progress. Enforcing the exactly-once property in an environment where network failures occur may require that the computation blocks for an unbound amount of time. Some systems allow duplicate agents to ensure progress [132, 165], while other systems block until the network stabilize [140] to avoid duplicate agents. An alternative approach to enforce exactly-once behavior is to make all operations performed by the mobile agent take the form of transactions. Doing so allows the effects of the operations to be rolled back in case a network failure triggers the creation of a redundant agent [128, 151, 165]. A simpler technique to enforce exactly-once behavior in mobile agent computations is to statically set up the replication strategy according to the network topology before the computation starts [41, 61]. The purpose of this approach is to reduce dependencies on weak communication links.

One benefit of mobile agent computations compared to conventional distributed computations is the ability to explicitly adapt the computation based on knowledge of the network topology. By explicitly adapting to the network topology, a mobile agent computation is *network aware* and can minimize exposure to weak communication links. We conjecture that mobile agent fault-tolerance also benefits from being network aware instead of having replicas being managed transparently to the computation. More specifically, the thesis of this dissertation is that:

Network aware fault-tolerance provides similar benefits over transparent fault-tolerance that mobile agents provide over conventional distributed computing.

To evaluate our thesis, we devise a protocol for mobile agent fault-tolerance. This protocol provides network aware fault-tolerance. The strength of the thesis is determined by the following properties:

Performance. Supporting network aware fault-tolerance requires a protocol that allows the mobile agent to adapt fault-tolerance when required. However, the latency overhead of executing such a protocol may outweigh the performance benefits compared to a protocol that transparently manages fault-tolerance. Despite this, our hypothesis is that our *network aware fault-tolerance protocol allows adapting to changes in the topology faster than a transparent fault-tolerance protocol.* We will test the validity of this hypothesis by comparing the performance of our protocol for network aware fault-tolerance and a protocol for transparent fault-tolerance. Confirming this hypothesis will strengthen the thesis.

Applicability. The applicability of a system is hard to quantify. However, we have two requirements that must be satisfied for our thesis to be relevant. The first requirement is to ensure that our solution to network aware fault-tolerance is relevant. Doing so requires that we establish how communication behaves in grid environments. Second, the applicability of our mobile agent fault-tolerance beyond mobile agent computations indicates that our protocol is sufficiently general to be applied in distributed computations without mobility. Thus, we must show that our mobile agent fault-tolerance has applicability outside the field of mobile agent computations and that existing distributed systems can be amended for and benefit from applying our protocol.

Although this dissertation mainly focuses on the issues of adapting to the network topology, we conjecture that our results can be generalized to adapting to other resources exposed by the underlying system, such as CPUs, disks or software.

1.4 Methodology

While the debate whether or not computer science is a science has been long and controversial, computer science is known to satisfy all requirements for scientific research [44,53].

In scientific research, there are several methods for testing theories and hypotheses. One of the most applied methods in natural sciences is the hypothetical-deductive method. The hypothetical-deductive method is the process of executing four steps: i) identifying the hypothesis to test, ii) making predictions based on the hypothesis, iii) checking the validity of the hypothesis with experiments, and iv) rejecting, modifying or accepting the hypothesis.

Although not cited explicitly, the hypothetical-deductive method is fundamental to the work of the ACM Education Board which defines an intellectual framework for the discipline of computing [39]. This framework divides the methods of computer science in three categories:

1. *Theory*, which is rooted in mathematics.

2. *Abstraction*, which is rooted in the experimental scientific method.

3. *Design*, which is rooted in the field of engineering.

The paradigm of theory consists of executing four steps for the purpose of establishing a theory. First, the objects which are to be studied are identified, and form the basis of a definition. Next, the relationship between the objects are posted as hypotheses, forming the basis for theorems. Subsequent proofs then determine whether the relationships are true, and finally the results are interpreted.

The paradigm of abstraction consists of executing four steps for the purpose of investigating a phenomenon. First, a hypothesis is proposed. Next, a model is constructed and predictions are made based on this model. An experiment is then designed and data is collected based on the executing this experiment. Finally, the results are analyzed. As with the hypothetical-deduction method, the scientist is expected to iterate these steps if the experiments differ with the predictions of the model.

The paradigm of design consists of executing four steps for the purpose of solving specific problems during construction of a system. First, the requirements are stated along with specifications for the system. Next, the system is designed. An implementation is then created based on the design specification. Finally, the resulting system is tested. As with abstraction, the steps are iterated if the system does not match the specifications.

The following section elaborates on how we have applied the methodology in practice.

1.4.1 The Methodology in Practice

Theory, design and abstraction are intertwined and it is usually hard to separate them from one another [39]. This is also the case in this dissertation.

In the category of theory we devise algorithms for master-worker computations and mobile agent fault-tolerance. Both these algorithms must expose properties that are advantageous for computations in grid environments. Thus, the process of devising the algorithms involves logical reasoning based on satisfying properties that are realistic in grid environments. Through mathematical and logical analysis of the algorithms we evaluate that the requirements are satisfied.

In the category of abstraction we observe and analyze the behavior of executing two algorithms for master-worker computations in a simulated environment that is similar to the environment where the algorithms are assumed deployed. The purpose is to investigate how communication behaves in grid computing environments and how communication affects the performance of our algorithms. Simulations allow analysis of a complex environment — a particular advantage is that deterministic algorithms can be re-executed with completely reproducible results. Thus, we use simulations to establish how our algorithms behave, and to evaluate whether the algorithms satisfy predicted behavior.

In the category of design, we start by inferring initial requirements for our mobile agent fault-tolerance by analyzing the requirements and results from existing mobile agent systems in the literature. We then extend our initial requirements with the requirements and results from running simulations of master-worker computations in grid environments. The design step of the resulting system involves a number of iterations, where different properties of the systems are balanced against each other. We first devise and implement a protocol that provides mobile agent fault-tolerance, and evaluate the performance of the resulting system. We then evaluate the applicability of our fault-tolerance approach by extending a non-mobile application with our protocol for the benefit of gaining fault-tolerance. Analysis of the result will either confirm, reject or require the thesis of this dissertation to be modified.

1.4.2 Trace-based Simulations

Parts of the requirements for our mobile agent system are based on the results from analysis of simulations of algorithms. Unfortunately, there are no accurate existing models of network connectivity that would allow us to analyze network communication behavior for complex algorithms such as the Moshe group communication system [95]. This makes it difficult to analyze the behavior of the proposed algorithms based on implementing the algorithms and measuring their performance, as would be required to satisfy the *design* category of computer science. Hence, we have followed other similar work [4, 13] and resorted to using network traces.

We also considered using the traces from the All Pairs Pings project in the PlanetLab environment [137]. However, since our analysis requires ping messages to be sent with high frequency between all participants, the traces did not possess the required fidelity. We considered creating our own version of the All Pairs Pings project to avoid this problem, but discovered that more aggressive ping measurements than used in the All Pairs Pings projects are discouraged by the PlanetLab service level agreement.

1.5 Context

This dissertation has been written as part of the **T**romsø **A**nd **CO**rnell **M**oving **A**gents (TACOMA) project [88, 92] at the University of Tromsø. The purpose of the TACOMA project has been the continuous development and refinement of a mobile agent infrastructure to complement classic distributed communication paradigms such as Remote Procedure Calls (RPC) [27] . The focus has been on how to structure the agent and the agent support environment, providing security mechanisms to ensure the integrity of the host that executes the agent, and providing appropriate fault-tolerance mechanisms — in short, the system support required by an agent computing environment.

The TACOMA project conjectured the need for mobile agent fault-tolerance from the outset, and was one of the first mobile agent projects to publish algorithms for mobile agent fault-tolerance in the research community [89, 92]. The problem domain of this dissertation emerged directly from this focus.

1.6 Summary of Contributions

The following are the main contributions of this dissertation:

- We have evaluated a group communication system called Moshe [95] and a master-worker algorithm called AX [66] and found that their performance depend heavily on the amount of network instability.

- Based on our evaluation of Moshe and AX, we have devised WAMW, an alternative master-worker algorithm. Although WAMW has a much higher message complexity than AX, it completes the same computations significantly faster.

- We have designed, implemented and evaluated NAP, a mobile agent fault-tolerance algorithm. NAP is based on the primary-backup approach to fault-tolerance, but allows backups to be rearranged during the computation to adapt to resource changes in the execution environment.

- During development and evaluation of NAP, we refined and implemented a new version of our mobile agent platform that hosts NAP. We performed a number of scalability optimizations and present a performance evaluation of the resulting system.

- For the evaluation of the applicability of NAP, we have devised a version of WAMW that uses NAP to improve its fault-tolerance within a local area network.

The work of this dissertation has resulted in a novel mobile agent fault-tolerance protocol that provides mobile agents with the ability to adapt to changes in the network topology, and thus reduces exposure to weak communication links.

1.7 Dissertation Outline

This chapter has given the motivation for this dissertation, the research problems and the thesis statement. We then examined our methods and summarized the contributions. The rest of this dissertation is structured as follows:

- Chapter 2 gives the background for process migration, mobile agents, and fault-tolerance. We identify the general tradeoffs in providing mobile agent fault-tolerance.
- Chapter 3 investigates the execution environment which we assume for this dissertation. We do so by evaluating two master-worker algorithms. We identify what types of network failures happen on the Internet, why they occur, and their performance impact.
- Chapter 4 evaluates the performance of a master-worker algorithm called AX that is based on group communication.
- Chapter 5 presents an alternative master-worker algorithm called WAMW that is based on flooding and leases. We evaluate the performance of WAMW and compare the results with algorithm AX.
- Chapter 6 presents the design and implementation of our mobile agent fault-tolerance protocol, called NAP.
- Chapter 7 evaluates the performance of NAP, and also gives estimators that can be used to predict the performance of a mobile agent computation with NAP.
- Chapter 8 discusses our findings, where we start by describing alternative solutions to replication. We then discuss our assumption on perfect failure-detectors and end by discussing the applicability of our work.
- Chapter 9 concludes and suggests future work.

Chapter 2

Mobile Agent Fault-Tolerance

Investigation of prior work precedes the process of proposing new solutions. We divide this investigation in two parts. The first part is formed by this chapter and characterizes the concepts of mobile agents and mobile agent fault-tolerance. The second part is given in the subsequent three chapters, where we analyze the behavior of a common grid computation when deployed on the Internet. Together, these two parts provide the background for devising a mechanism for mobile agent fault-tolerance.

In this chapter we start by presenting early works in process migration and continue by investigating mobile agent systems. We then present the two approaches to replica management in the field of fault-tolerance. Based on this, we continue with related work on mobile agent fault-tolerance. The chapter ends by establishing initial requirements for our mobile agent fault-tolerance mechanism based on the investigation of prior work.

The areas investigated in this chapter span a large amount of research, and several excellent survey papers provide more in-depth information than space allows in this dissertation. For process migration, consider [124], mobile agent fault-tolerance is surveyed in [141], while [60] gives a thorough taxonomy for code mobility.

2.1 Motivation

Mobile agents emerged almost two decades ago as a novel way of structuring distributed computations [182]. One of the key problems that motivated the introduction of mobile agents was the performance problem with RPC [27] over weak communication links. This problem was particularly strengthened by the introduction of ad-hoc communication infrastructures in industrial domains where communication links are fundamentally poor — for instance, fishing vessels in the Arctic sea where sea waves and the curvature of the earth weaken signals, or vehicles located far away in outer space where the distance itself makes the signal latency extremely long [86]. Such circumstances, coupled with the increasing Internet-wide communication, made communication-intensive distributed computations with RPC challenging.

The classic difference between the RPC and mobile agent approaches to communication is depicted in Figure 2.1. On the left side we have a client that communicates messages with a server as part of a distributed computation. The client issues a request

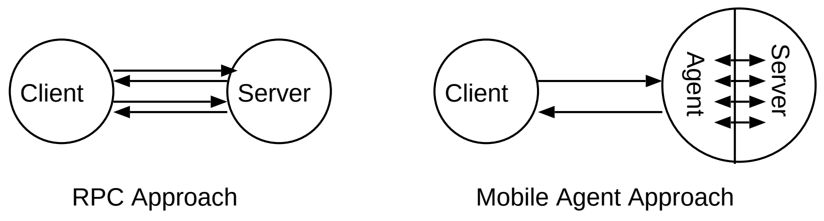


Figure 2.1: Comparison of RPC and mobile agents approaches to communication.

message, and the server sends a response message. There are two main performance problems with this approach. First, if network communication between the client and the server suffers from long message latency, then the interaction between the client and the server may become significant to the total execution time of the computation. Second, if network communications between the client and the server has low message bandwidth, a request or response with a large message size may cause an increase in the total execution time of the computation.

On the right side of Figure 2.1, we have a mobile agent that is sent from the client to the server. The mobile agent implements the parts of the client that need to interact with the server. This approach reduces the number of network interactions between the client and the server to two, or one if the client does not require a result from the computation. In addition, all subsequent interactions between the client and the server are through local procedure calls and thus not subject to the performance of network latency or available network bandwidth.

Besides reducing network load and overcoming network latency, mobile agents are known for the following advantages [36, 106]:

- **Protocol encapsulation**. A mobile agent can implement and encapsulate legacy communication protocols and interfaces between the server and the client, which better facilitates subsequent protocol or interface upgrades.

- **Asynchronous execution**. Computations can be embedded in the mobile agent, and then sent to a server (or set thereof) for execution. After being sent from the client, the agent no longer depends on the client and can execute autonomously and asynchronously, even if clients are intermittently disconnected. Later, the agent may communicate with the client to exchange the result of the computation.

- **Adaptivity**. Mobile agents can be programmed to adapt to changes in the execution environment if the environment is not suitable or lacks appropriate resources.

- **Heterogeneity**. Mobile agents do not depend on the hardware or the software platform where the mobile agent environment executes. Rather, this is abstracted away by the mobile agent environment, which makes mobile agents suitable for system integration tasks.

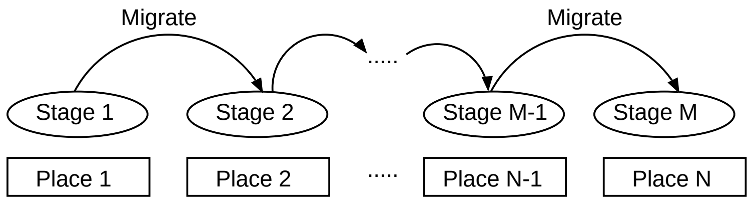


Figure 2.2: The model of mobile agent computations.

There are several factors that influence whether or not a mobile agent computation will have better performance than the corresponding RPC computation. Although the number of network interactions can be reduced, the cost of sending the agent from the client to the server can still be high if the agent and its associated state are large. The same applies for the result sent from the server to the client after the agent has completed its execution. A more fundamental overhead with mobile agents compared to RPC is the additional costs incurred by setting up and maintaining an execution environment for the agent at the server. In one mobile agent system, the cost of sending a minimal agent from the client to the server was found to be over 20 milliseconds [90], where a corresponding minimal RPC call would take just over 5 milliseconds.

Computational Model

The mechanisms for capturing the state of the agent, sending the state over the network, and resuming execution at the server (i.e., *migrating* the agent), is often the key differentiator among mobile agent systems. While there are many approaches to the implementation of migration in mobile agent systems, the migration itself is only one part of the computational model for mobile agents.

From a logical point of view, a mobile agent computation consists of a sequence of *stages* where each stage is executed by the mobile agent system, or *place*, running on a computer. A place can execute several stages during the same computation. The execution of each stage is *terminated* when the agent *migrates* to a place, which causes the next stage to be started. When there are no stages left to execute, the computation terminates. The sequence of places that the agent visits during the computation is called the *itinerary* of the agent, which is why this model of computations is commonly known as an *itinerant* computation. A mobile agent computation is depicted in Figure 2.2, where M stages are executed on N places before the agent terminates.

Before investigating how various mobile agent systems are designed and implemented with respect to this computational model, we give an overview on the mechanisms that influenced the design rationale for many mobile agent systems, namely *process migration* [124].

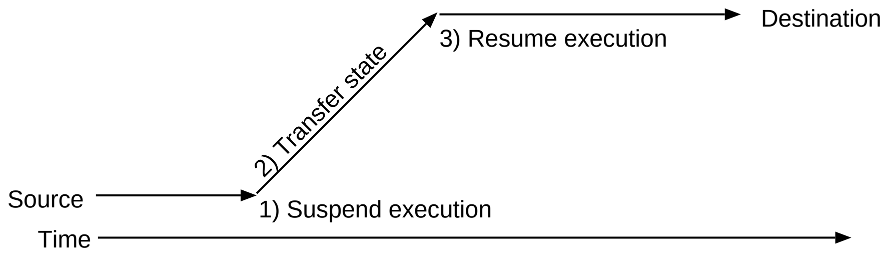


Figure 2.3: Execution flow of process migration.

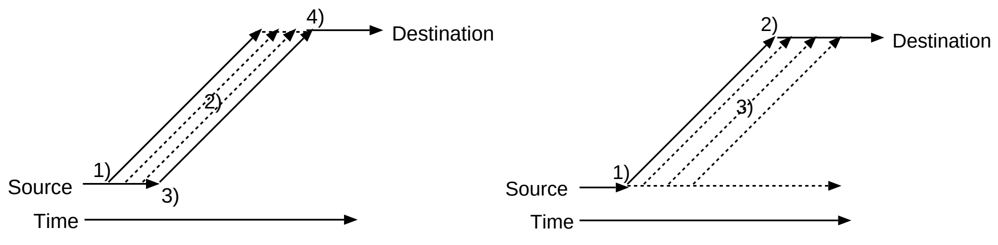
2.1.1 Early Process Migration

The *process* in process migration systems originates from the process definition in operating systems, and consists of data, stack, registers and other operating system specific state related to open files, network connections and memory management. At least one thread of control is associated with a process, where each thread has its own stack and registers. *Process migration* is the procedure of transferring the process from one computer to another computer and resuming execution without affecting the control flow of the process (i.e., as *transparently* as possible). The main policy issues in process migration include determining *when* to move, *where* to move, *what* to move, and *who* decides to move. In this overview we focus on the issues of what to move, and how to move.

Process migration emerged in the early eighties for some of the same reasons as mobile agents: if a process lacks access to either a local resource or some remote resource must be accessed locally, the process can be migrated to the computer where the resource can be accessed. Another advantage is that long running processes can, if notified in advance, be migrated to another computer if the current computer is to be shut down. Migrating a process from one computer to another is also attractive for load-balancing or resource sharing purposes. Figure 2.3 shows the execution flow of a process migration, which consists of three main steps:

- **Suspending execution**. A process is executing at the source computer when execution is suspended in step 1.
- **State transfer**. The state of the process is transferred to the destination computer as step 2. The state typically consists of process related information such as the memory image, program counters, and open descriptors (for files, sockets, etc.).
- **Resuming execution**. When the state has been transferred, execution is resumed at the destination computer in step 3.

The execution flow depicted in Figure 2.3 shows the case where the process does not resume execution before being completely transferred to the destination computer. The benefit of the *direct-copy* approach in Figure 2.3 is that the resources occupied by the process at the source computer can be relieved when transfer has completed. If we,



(a) Pre-copy (b) Lazy-copy

Figure 2.4: Optimizations for process migration.

however, allow the process to start execution at the destination before the entire state has been transferred by using *lazy-copy* or *copy-on-reference* approaches, the performance of the migration can be improved. Rather than copying state on demand, performance can also be improved by copying state before the process is migrated, as in the *pre-copy* approach. One of the fundamental problems in migrating a process is that sometimes the complete process state cannot be migrated, leaving *residual dependencies* on the source computer. Residual dependencies imply that operations on files or resources that have not been migrated to the destination computer must be routed through a local proxy that transparently communicates with the source.

Process migration can be implemented in a variety of ways, from first approaches with message-passing in DEMOS/MP [143], with kernel support in the Sprite [130] and V [35], to user-level libraries in Condor [114]. We now explain how DEMOS/MP, Sprite, V and Condor solve the issues related to the migration procedure.

DEMOS/MP

DEMOS/MP [143] is a distributed system based on the use of message-passing, a characteristic inherited from the earlier DEMOS operating system for the CRAY-1 [17]. Messages in DEMOS/MP are sent on *links*, and a link is associated with a process. Processes that are willing to accept messages are responsible for creating the link. A link can also be sent in a message to allow other processes to use it, but the link still points back to the originating process.

The process migration mechanism in DEMOS/MP implements the *direct-copy* approach where the process at the source is frozen and incoming messages on the links associated with the process are buffered in the link queues. The destination is then requested to migrate the process and allocates the necessary process state. All process related state is then transferred to the destination. After this step has completed, control is resumed to the source and buffered messages are then sent from the source to the destination. All state at the source is cleaned up, and a *forwarding address* is created and points to the last known computer where the process was executing. Finally, execution of the process is resumed at the destination.

Since the use of message-passing is pervasive for both operating system services and the kernel in the DEMOS system, the actual implementation of the process migration mechanism is slightly simplified. The only extension of the original system to support migration is the addition of forwarding addresses and mechanisms for updating links to reflect the current location of the process. After a process has migrated, a message is sent on every link to update the forwarding address to the current location, which avoids long chains of forwarding addresses. The residual dependencies of a process in DEMOS/MP are bound by the number of links the process has when migrated. A major disadvantage is that this makes the migrated process induce load on the source after being migrated. The migrated process is also vulnerable to failures both at the source and the destination computer. Other systems like LOCUS [142] and Charlotte [8] use the same direct-copy approach as DEMOS/MP.

The V System

The V kernel project aimed at providing network transparency for interprocess communications while preserving high performance [35]. The motivation for introducing process migration in V was the desire to use idle workstations as a pool of computers that users could harvest upon demand. When users logged in on computers that host migrated programs, these programs would then be *pre-empted* and migrated to another computer as quickly as possible.

A requirement for V was that the migration procedure should cause minimal interference and blocking of the process being migrated. This means that the time that is spent on state transfer in Figure 2.3 should be made as small as possible. After migration to the destination, there should be no residual dependencies to the source. The argument here is that residual dependencies induce load on the source computer, which defeats the purpose of migrating a process for performance reasons.

All processes in the V system are identified by a globally unique identifier (pid). The pid is used in all interprocess communication calls as well as in the migration primitives and makes the V system network transparent. Doing so helps keeping the execution environment for processes transparent, since all references that are outside the V address space use pids. Some references that are tied to local resources such as frame-buffers and audio devices cannot be migrated, and applications using these resources can thus not be migrated. The address space in V is identical across all computers in the system, which aids keeping transparency within the address space of the process. The process migration mechanism in V was the first to use a technique called *pre-copy*. The mechanism involves the execution of four steps, as visualized in Figure 2.4a:

1. Initialize and prepare the destination for the incoming process.

2. Continuously copy the address space of the process at the source to the destination until the number of dirty pages is sufficiently small.

3. Freeze the source computer and finish the migration by sending pending dirty pages since step 2.

4. Resume the process at the destination, remove residual dependencies to the source by rebinding all references outside the address space.

A predefined number indicates the amount of dirty pages that is sufficiently small to terminate the algorithm. It has been shown that termination usually requires only two or three rounds before the dirty set is small enough to start the freeze in step 3 [35]. This means that the freeze time of the process can be made very small, since it only involves the transfer of a few dirty pages. All state that is not globally accessible in V (e.g., local files) is part of the state that is migrated from the source. This means that the actual pre-copy step could take considerable time if the process has opened large files.

When pre-copy has completed and the source is frozen, the process at the destination assumes the logical host id of the source computer to allow processes that communicated with the migrating process to send messages to the process at the destination while the source is frozen. After all state has been transferred, communication endpoints are removed at the source, and the interprocess communication primitives in V ensure that other processes re-establish communication endpoints at the destination.

Sprite

The Sprite [130] project is a network operating system whose main design goal is that computers should be made available as a time-shared computer, but still retain the performance of an individual computer. If a computer that hosts a migrated process is occupied by a user, the migrated processes should be pre-empted from the computer. The idea is based on the same principle as SETI@Home [5] and V: without local autonomy of the computer, users would not run foreign processes in the first place. In addition, a process should, regardless of migration, be unaware of its network location during execution.

Migration in Sprite is transparent, and based on the concept of a *home machine*. The home machine is the computer where a process originates from. If a process is migrated, the Sprite operating system ensures location transparency by forwarding most system calls to the home machine. A significant part of Sprite is a distributed file system that allows all file related calls to be performed locally. If a migrated process executes on a computer and a user returns to use that computer, the migrated process is evicted and migrated back to its home machine. The migration mechanism in Sprite, involves the execution of the following steps:

1. The source determines what constitutes the state of the local process, allocates a buffer that holds the state, and flushes its virtual memory to the file-server.

2. The source sends the buffer to the destination through RPC, and upon receipt the destination prepares the environment of the process, such as ensuring that open files are transferred.

3. The source sends an RPC to the destination, telling it to resume execution of the process, and cleans up the buffer allocated in step 3. The process at the destination demand-pages the process image from the file-server.

All system calls that are executed on the home machine cause a residual dependency in Sprite, which means that migrations still induces load on the home machine, as well as the migrated process depending on the home machine not crashing.

The Sprite process migration technique is known as a variation of the *copy-on-reference*, or *lazy-copy* approach introduced by Edward Zayas [185]. Lazy-copy was first used by the Accent system [145] and later in Mach [125]. In Accent, the migration procedure consists of the three steps visualized in Figure 2.4b: 1) all process state except memory is sent to the destination, 2) the process starts execution at the destination, and 3) process memory is demand-paged from the source to the destination. The main advantage of the approach in Sprite compared to Accent is that there is no dependency on the source keeping the process image after migration in Sprite. Sprite, however, requires more communication since the process image is first flushed to the file server and then paged in by the destination afterwards.

Condor

Condor [114] is a system that runs entirely in user space and supports checkpointing and process migration for distributed computations [113, 115]. Like many other process migration systems, the main design goal for Condor was motivated by the needs to harvest unused capacity for long-running computations.

When a process is migrated in Condor, a large part of the environment from the source is transferred and reproduced at the destination. The migration is performed by checkpointing the entire process state and is provided through a user level library that all processes that use checkpointing must link to. The library provides system call wrappers that allow Condor to track the state of the process, for instance open files. The Condor checkpoint mechanism is expensive and requires the entire process image to be written to disk (comparable to direct-copy), and is thus not suitable for short-lived processes.

By default, Condor assumes that files are located at the same paths on the source and destination computers. Condor also supports file systems that are not shared through the use of shadow processes that execute on the source where the checkpoint was performed. The Condor process at the destination communicates with the shadow process at the source through RPC (e.g., for file access), after migration. Condor is not as transparent as V and Sprite, and there are many limitations in addition to requiring processes to link with a special library. For instance, Inter-Process Communication (IPC) is not supported, nor are system calls like *fork* and *exec*.

2.1.2 Worms

The early process migration systems focused on providing support for *transparent* process migration. Contemporarily, another approach, called “worms” [163], emerged where the migration was cooperative. Instead of hiding migration from the process, the process was instead programmed with migration in mind. The idea of a worm originates from the nature of the reptile, where a computation moves through the network in a crawling fashion. The computation being processed by the worm is distributed to several *segments*, and each segment occupies at least one computer. If a segment fails, other segments try locating an appropriate computer where the failed segment can be re-instantiated. The code in the worm consists of three main parts. First, bootstrap code that is executed on the first computer. Second, bootstrap code that is executed on subsequent computers. Third, the computation the worm should do, including the code for the mechanism that keeps the worm computation running. The mechanism that maintains the lifecycle of the worm can be split into the following steps:

- Locate computers. This forms the basis for creating segments and is executed by the bootstrap code by the first computer.
- Boot computers with the bootstrap code embedded in the worm. The environment hosting the worm normally bootstraps computers from file servers, which is the procedure the worm replaces.
- Maintain the worm through continuous communication between segments to detect failed segments. Failed segments are re-instantiated on another computer.
- Terminating the computation by rebooting the worm computers.

The Worm is appropriate for distributed applications that require robustness, such as distributed alarms or diagnostics tools.

Internet Worm

The classic Internet Worm [167] was written by Cornell student Robert Morris and launched in 1988. The purpose of the worm was not intentionally malicious, but rather an attempt at measuring the number of computers connected to the Internet. However, the worm code had a problem that caused multiple instances of the worm program to occupy each computer, ultimately slowing the computer down. The Internet Worm migrated by exploiting weaknesses in Unix passwords, the `sendmail` program, `rsh`, and `finger`. The number of computers that where affected by the Internet Worm is reported to around 6000 [46], approximately 10% of the computers connected to the Internet at the time.

The Internet Worm and the Worm [163] both share the same weakness since an error in the worm program can cause it to execute without bounds and thus become difficult to terminate.

2.1.3 Virtual Computers

Recently, the interest in computer virtualization as implemented by VMWare [172] and Xen [15] has produced systems that support migration of virtual computers [38, 155]. The idea is that rather than supporting migration within the operation system, the entire virtualized operating system and its state are migrated to the destination. This is possible by piggybacking on the virtual monitor layer of the virtualization software.

The project described in [155] is built on top of the VMWare GSX virtualization software for x86 processors. When running a virtualized operating system, the image representing the virtualized system may occupy hundreds of megabytes. The motivation for the project is how to migrate this system image, say from the work environment of the user to the home environment, as fast as possible. The abstraction used for migration is called a *capsule*, and represents the state of the virtualized system. The project uses several optimizations for transferring capsules, such as tracking modified disk pages for the capsule with a copy-on-write mechanism, demand-paging capsule pages at the destination, using checksums on data to avoid redundant copies to be sent, and tracking which memory pages are actually used by the virtualized operating system. These optimizations allow a migration of a virtualized environment with 256 MB of memory and 4 GB of disk in as little as 20 minutes on a 384 kB/s link.

The project described in [38] takes the idea of migrating a virtualized operating system one step further by allowing migration while the system is running by using the pre-copy technique from V. They run several experiments where the virtualized system executes a web server and a game server for Quake 3, where the measure of success is the amount of downtime due to migration. By carefully tuning when to freeze the pre-copy stage, these servers both achieve downtime well under 1 second, which is close to imperceptible in practice.

The problem with both these approaches is that the cost of migration is still high and requires significant network resources, which makes them unattractive for migration across communication links with limited bandwidth.

2.1.4 Observations

Transparent process migration, or *strong mobility* [60], as implemented in Sprite and V, requires significant support from the operating system kernel. Most computers today do not run a distributed operating system, and process migration is not supported in operating system kernels like Linux[1]or Microsoft Windows[2] Without kernel support, determining the state that is to be migrated and reincarnating the state of the process at the destination computer becomes difficult, which is evident from the complexity of adding the limited checkpoint transparency in Condor [113, 115].

Another major problem was the focus on homogeneous environments within local area networks. The work on virtual computers also share some of the problems of the early process migration systems, and require homogeneous environments and significant bandwidth to perform well. The Worm from Schoch and Hupp has a very specific purpose, which is to use idle workstations for a parallel computation. While robust in execution, the programming model is too constrained to be generally applied.

2.1.5 Mobile Agent Systems

When mobile agent systems emerged with Telescript in 1994, the deployment of the World Wide Web was growing fast. Since agents were assumed to execute in the same environment as the Web, mobile agent systems had to be built with hardware and

[1]Linux homepage: http://www.linux.org

[2]Windows homepage: http://www.microsoft.com/windows/

software heterogeneity in mind. Thus, rather than depending on direct kernel support for migration, mobile agent systems are normally implemented as middleware between the mobile agents and the operating system.

Another emerging technology at the time was the Java programming language and the Java Runtime Environment. Java offers support for safe execution of programs, suitable mechanisms for serializing program state and objects, as well as providing a runtime environment running on most available operating systems today. These features of Java make it attractive for implementing mobile agent systems, and caused numerous systems to be based on it [18, 67, 107].

The complexity of supporting strong mobility is one of the main reasons why many mobile agent systems only support *weak mobility* [60], where the state that is part of the migration is explicitly managed by the agent itself. While requiring less from the execution environment during migration, explicitly managing the state in the agent also keeps the amount of state to be transferred during migration to a minimum. This is a desirable feature when optimizing a computation for network performance.

We now investigate how the first mobile agent systems, Telescript [182], TACOMA [91], Agent Tcl [69], and Mole [18] work. The Agent Tcl evolved from supporting one language to supporting multiple programming languages with strong mobility. TACOMA offers weak mobility and supports multiple programming languages. Mole was an early approach to mobile agents running on the JVM.

Telescript

In 1994, the Telescript mobile agent system emerged from General Magic [182]. Coming from industry, the emphasis in Telescript was on how mobile agents could be leveraged in e-commerce settings. Although Telescript did not become commercially successful, it founded the abstractions that most of the subsequent mobile agent systems were based on.

There are two types of agents in Telescript. First, mobile agent programs that can *travel* (i.e., migrate) from one place to another using the `go` language abstraction. Second, stationary agents that implement the services offered by places to mobile agents. When a mobile agent wants to utilize the service offered by a stationary agent, the mobile agent uses the `meet` abstraction to exchange information.

Agents in Telescript are written in the Telescript programming language. The reason for devising this language is the difficulty of supporting migration in programming languages such as C and C++. The programming language is similar to Java, and shares many features such as being object-oriented, allowing safe execution of programs, providing orthogonal persistence through serialization, and abstracting the operating system from the programmer. As a consequence of having such a complete infrastructure in the programming language environment, the `go` abstraction in Telescript implements strong mobility.

Telescript was assumed to be applied in large-scale networks such as the Internet. This implies mechanisms to discern which agents are allowed to do what. Each agent is represented by an *authority*, which is either an individual or an organization. Places and stationary agents may choose to offer services only to agents of a certain authority.

Authorities provide the mechanism on which to base monetary transactions without the need for human intervention. Once an agent is authorized by another agent or place, a *permit* is given to the agent. Through granting capabilities to the agent, the permit controls what actions the agent can perform within a place — for instance whether the agent is allowed to create another agent or the amount of CPU-time granted to the agent in the place. By the Principle of Least Privilege [154], permits help places guard against buggy or intentionally malicious agents and are enforced by the runtime environment for Telescript. Without permits, Telescript would easily be susceptible to problems like the one caused by the Internet Worm [167] and render the places useless.

One of the fundamental problems with Telescript is the reliance on a proprietary language and language environment, thereby limiting Telescript to only supporting agents written in the Telescript language.

Agent Tcl

Agent Tcl was also one of the first mobile agent systems along with Telescript and TACOMA [69]. The first version of Agent Tcl used a modified version of the Tcl[3]language interpreter with added support for strong migration. The agent model is similar to that of Telescript, with mobile agents migrating from place to place and interacting with stationary agents that provide services to the mobile agents. The real difference between Agent Tcl and systems like Mole and Telescript is that it eventually evolved to supporting more than one programming language.

The requirement of strong migration forces interpreters to be altered with support for state capturing during execution. However, Agent Tcl does not allow residual dependencies, which means that open files or other shared resources outside the address space of the interpreters are not part of the state that is migrated from a place. As an example, migration in the Tcl interpreter is performed with the `agent_jump` statement. Upon invocation, the transport protocol for transferring the agent is determined. Next, the state of the entire Tcl interpreter is captured and sent to the destination place, where the agent is re-instantiated again and execution resumes to the statement following `agent_jump`.

Agent Tcl supports communication between agents with the `agent_meet` and `agent_accept` statements. When an agent A wishes to communicate with agent B, agent A invokes `agent_meet`. Agent B then decides whether to accept the request by issuing `agent_accept` or reject the request. The security model in Agent Tcl depends on a public key infrastructure for authenticating agents and providing encrypted communication. Security is enforced by the server layer in the architecture. Similar to permits in Telescript, a resource manager in the server layer checks all agent resource accesses against a set of access permissions for the agent.

TACOMA

The **T**romsø **A**nd **CO**rnell **M**oving **A**gents (TACOMA) system [91] was started in 1993 as an approach to overcome the performance problems of RPC in the StormCast

[3]Tcl homepage: http://www.tcl.tk

weather system [86]. The first version was not a mobile agent system, but rather a remote evaluation system inspired by the remote evaluation approach from Stamos et al. [168]. This version was implemented through the remote shell facilities on Unix systems, where procedures were shipped among computers for remote execution. At the same time, mobile agent systems started emerging for many of the same reasons that motivated the initial TACOMA system. Hence, in 1994 the TACOMA project was formally started as a mobile agent activity, and the next version that supported the Tcl programming language with weak mobility was released. By the subsequent release of TACOMA version 1.2 [93], multiple programming languages were supported.

TACOMA relies on agents explicitly managing their state through the use of *folders* and *briefcases*. A folder is an ordered list of strings, and a briefcase contains a mapping from folder names to the folders contained in the briefcase. Some folders have a special meaning and are maintained by the TACOMA runtime system. For instance, the *host* folder always contains the name of the place that executes the agent.

The decision to use folders and weak mobility is driven by the goal of having the runtime cost of migration controlled by the programmer, since the agent programmer is aware of the state that is needed for the future actions of the agent as well as how this state should best be stored. For instance, this allows the agent to drop folders that are no longer needed. In addition, using weak mobility and folders simplifies adding support for multiple programming languages. Abstractions similar to folders and briefcases are also used in the Knowbot [84] and Mobile Ambients [30] systems.

The fundamental communication primitive in the TACOMA system is called `meet`. When issued, it allows the local agent program to start a remote agent and passes a briefcase as argument to it. The agent issuing the `meet` operation can choose to block or execute in parallel with the remote agent. If blocking `meet` is chosen, the briefcase is returned when the remote agent completes, similar to RPC-style communication. Like other agent systems, TACOMA offers stationary service agents that implement services offered to mobile agents. One of these services are programming language specific agents that implement the execution environment for agents within TACOMA. As an example, an agent would migrate to a place and execute a Java program there by storing the program in the *javacode* folder and executing a `meet` operation naming the service agent for Java. The advantage with folders and briefcases is that implementing support for a new language only requires adapting folders and briefcases to the new language. The disadvantage is that folders and briefcases are still a fairly low-level abstraction compared to the offerings of the other mobile agent systems, but this can usually be overcome by adding language-specific layers on top of the folder operations.

When an agent invokes the `meet` operation, the briefcase is sent from the local agent to the destination place through a component in the TACOMA runtime called the *firewall*. In addition to orchestrating `meet`, the firewall manages the agents within its environment, including starting service agents when needed and checking whether the incoming briefcases are allowed to enter the place. TACOMA supports access control for agents through the use of digital certificates that are interpreted by the service agents. Once authenticated, agents have unrestricted access to all local resources, which provides less fine-grained access control than offered by permits in Telescript.

Later versions of TACOMA focused on alternative ways of structuring the agents [110], the use of agents for remote system administration [110], runtime service deployment, agent applicability in asymmetric network environments [85], in addition to exploring non-functional aspects like fault-tolerance [89] and security [159]. Most of these aspects are elaborated later in this dissertation.

Mole

Mole is one of the earliest mobile agent systems implemented in Java, where the first version was released in 1995 with several subsequent versions following this release [18]. Agents in Mole can move from place to place, where places offer services to visiting agents. Mole has service agents that implement the services offered to agents visiting a place.

A major difference between Mole and Telescript is that Mole does not support strong migration, since the JVM does not support capturing the state of a running thread [1]. Thus, to be able to run on unmodified JVMs, Mole instead implements weak mobility. Migration works by having the agent call a method `migrateTo(place)` when it decides to move to a particular place. The `migrateTo` method takes a transitive closure on the objects referenced by the agent instance (except for threads, which cannot be serialized), and serializes these into a byte sequence. The byte sequence is then submitted to the destination place, where the agent is recreated. All agents implement a `start` method that is the entry point for execution when the agent is started. If, however, an error occurs during execution of the `migrateTo` method, control is resumed to the statement after the `migrateTo` where error handling can be implemented. A problem with the weak migration approach in Mole is that the `start` method is the entry point every time the agent migrates to a new place, which implies that this method need to manually dispatch the code to run unless the same code is supposed to be executed on each place. In contrast, TACOMA uses the first element of the appropriate code folder as the entry point, which avoids the need for dispatching within the agent code itself.

Mole offers extensive support for different approaches to communication between agents through the use of *sessions*. Agents wishing to communicate create a session, where message passing can be used for subsequent interaction. Sessions allow agents to interact similar to the *meet* primitive in Telescript. Mole also supports anonymous group communication between agents through the use of tuplespaces. Security is implemented with a sandbox. Mobile agents have no access to the resources exposed by the underlying system, but have to go through service agents. Service agents are immobile and provide secure abstractions for resource access to mobile agents. Places can discern which mobile agents are allowed to visit, similar to the authorities in Telescript.

One of the weaknesses in Mole, like in Telescript, is the assumption that all agent code has to target a single language, which makes it harder for agent programs to leverage utilities and libraries written in other languages.

2.2 Mobile Agent Computations and Failures

When the computational model in Figure 2.2 is deployed in an asynchronous environment such as the Internet, failures are likely to occur. The failures we consider in this dissertation were identified in Section 1.2 as *crash failures* and *network failures*. For mobile agent computations this means that at any place during the computation of a stage, the place may fail because of the mobile agent crashing at the place, the place crashing, or the network link connecting the place to other places has failed and cause messages to be lost. Crash and network failures can stop the progress of the mobile agent computation. Compared to RPC, masking such failures is harder since the agent may be executing asynchronously at a place where the client is unable to recover the agent.

Consider the weather alarm system described in [85] that allows users to submit tiny agents written in a domain specific language with cellular phone text messages to a TACOMA installation running on the Internet. The motivation for building this system was the emerging ubiquity of cellular phones, which at the time had no applications for communicating with Internet services. While the weather alarm system enabled users to communicate with Internet services, the extremely low bandwidth offered by GSM text messages [79] presented other problems. For instance, it was very difficult for users to verify whether their weather alarms were still running or if the server on which they executed had failed. Rather than having users periodically check the status of their alarms with GSM text messages, providing the users with a fault-tolerant environment for running their weather alarms seemed like a better approach.

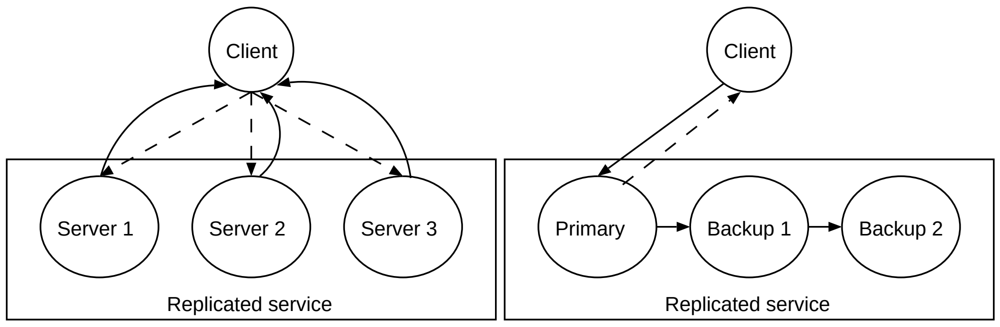
The issues of tolerating failures in mobile agent systems have spawned numerous systems based on replicating the mobile agent computation in some way. Before we investigate how these systems work, we explain the two major fault-tolerance approaches for replicating distributed services, the *replicated state machine approach* [158], and the *primary-backup approach* [28], and how these approaches influence the computational model of mobile agents.

2.3 Replica Management

In traditional client-server models, the replicated state machine and primary-backup approaches are modeled through clients making requests to servers and servers responding to client requests. A system consisting of n distinct servers is said to be f fault-tolerant if it withstands f failures within a bounded time interval. Here, $n > f$, and the minimum value of n/f depends on the protocol and the failure model. As specified in Section 1.2, clients and servers can fail by crashing, and network links can fail and cause request and response messages to be lost but not corrupted.

2.3.1 Replicated State Machine Approach

A state machine consists of variables that encode the state, and commands that transform the state and possibly generate responses. Commands are atomic with respect



(a) Replicated state machine (b) Primary-backup

Figure 2.5: Replica management.

to other commands and are executed in a deterministic way: the outcome is solely a function of the current state and the command. A client makes a request to a state machine to execute a command. The state machine executes the command, and the output of the command, if any, is returned to the client as a response. To simplify the presentation, we assume that a response is always sent to the client.

Making a state machine f fault-tolerant involves replicating the state machine on a set of servers. Assuming that each state machine replica starts in the same initial state and all client requests are executed by the non-faulty replicas in the same order, each replica will produce the same output. An illustration of state machine replication is shown in Figure 2.5a. In this illustration, a client issues the same request to all the servers, as depicted by the dashed arrows. Each replica executes the command referred to by the request and the output produced is sent back to the client in the response, as depicted by the solid arrows.

The client is not required to act as the sender of the request to all replicas. Instead, the client can send the request to a single replica and have this replica send it to a sufficient number of other replicas. However, this requires the client to ensure that the request is not lost before the request has been sent to a sufficient number of non-failed replicas. The protocol that ensures this property is called *consensus* [135].

A challenge with state machine replication is ensuring that replicas are kept in the same state. This requires that a sufficient number of replicas agree on the ordering of each client request and the output of each client request. An approach to ordering client requests is to consider the sequencing of client requests as a sequence of consensus problems [135]. It has been proved that consensus cannot be reached with asynchronous communication [52]. However, by using an unreliable failure detector, $2f+1$ replicas are sufficient to reach consensus when f servers can fail even with very weak assumptions on synchrony [33]. Crash failures and synchronous communication lead to a simpler model for consensus (e.g., using *atomic broadcast* [158,160]), where $f+1$ replicas are sufficient to tolerate f failures.

Agreement on the output of each client request is typically performed by voting, either in the client itself or by use of a protocol between the servers before the output is returned to the client. For *fail-stop* [157] crash failures, voting is not necessary since a failed server will by definition not generate any output.

2.3.2 Primary-Backup Approach

Similar to the replicated state machine approach, the primary-backup approach implements a fault-tolerant service through the use of multiple servers. One of the servers is designated as the *primary*, where the others are designated as *backups*.

An illustration of a simple primary-backup protocol is shown in Figure 2.5b. The client makes a request to the server that is designated as the primary, as depicted by the solid arrow. The primary processes the command referred to by the request and updates its state. The primary then sends information about the update to all the servers designated as backups, as depicted by the solid arrows starting from the primary. Finally, the primary sends the output of processing the command back to the client, as depicted by the dashed arrow from the primary to the client. The state information sent from the primary to the backups can be encoded in many ways: i) a snapshot of the state of the primary after the client request was executed, ii) the state changes caused by executing the client request, and iii) the client request itself and, if applicable, a trace of the non-deterministic events that occurred at the primary during the invocation of the command.

Backups monitor the primary to detect failures. If the primary fails, then one of the backups takes over the role as the primary. Which backup takes over as the primary can be determined statically or can be determined dynamically by a *leader election* protocol (see [116] for examples). After the new primary has been chosen, clients are notified and subsequent client requests sent to the new primary. Thus, in contrast to replicated state machines, the failure of a server is not transparent to the client.

When servers (i.e., the primary and the backups) in a primary-backup protocol are subject to network failures, $f + 2$ servers are required to survive f failures [28]. This assumes that there are at least one non-faulty path between two non-failed servers, where the path may include any number of intermediate servers. For fail-stop crash failures, $f + 1$ servers are required to survive f failures [28]. This means that the number of required servers are the same as for the replicated state machine approach for such failures. The significant advantage of primary-backup compared to the replicated state machine approach, however, is that it enables replication without redundant request processing.

2.4 Mobile Agent Fault-Tolerance

There are mobile agent systems that implement fault-tolerance with either primary-backup or the replicated state machine approach. One of the first systems to offer support for mobile agent fault-tolerance was based on the replicated state machine approach [126]. To implement a *1 fault-tolerant system* that tolerates fail-stop crash

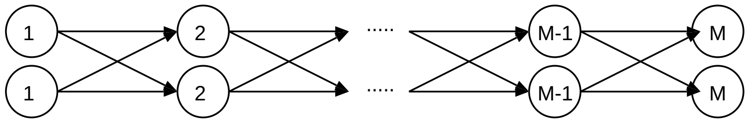


Figure 2.6: Mobile agent fault-tolerance with the replicated state machine approach.

failures, each stage is executed by 2 places.[4] When a stage has completed, the agent code and the state resulting from executing at that stage are sent to the 2 places that execute the next stage. Doing so allows masking a single crash per stage. The computational model when executing M stages given a 2 fault-tolerant system tolerating crash failures is shown in Figure 2.6. Circles depict agents, the number in the circle is the stage number, and the arrows the agent state going from one place to the next.

Since only a single agent code and state result is required to progress to the next stage, a single slow place for a stage will not delay the computation. If all places for a stage are slow, adding a replica per stage may help if that replica is faster. However, going from R to $R + 1$ replicas per stage increases the number of messages exchanged between each stage from R^2 to $(R + 1)^2$, which may cause an increase in bandwidth utilization.

The main drawback with this approach, and the replicated state machine approach in general, is that each replica is executing the stage (i.e., state machine command). In a grid environment where computational resources are shared among grid consumers, redundant processing is not desirable. For this reason, we focus on approaches to mobile agent fault-tolerance that use some variation of primary-backup.

2.4.1 Exactly-Once Properties and Blocking

The first mobile agent computations that were proposed involved electronic commerce, where agents autonomously do hotel reservations or order plane tickets. A requirement for such mobile agent computations is to satisfy *exactly-once* execution [141]. Exactly-once execution implies that all stages in the mobile agent computation are executed exactly once, regardless of stage failures. If exactly-once is not enforced, a failure can cause the agent to execute a stage action more than once (e.g., resulting in reserving a hotel room twice). Thus, in mobile agent computations involving electronic commerce, enforcing exactly-once becomes a correctness concern.

Grid computations are typically composed of several smaller tasks where the execution of a task is *idempotent* (i.e., executing a task multiple times produces the same result for each execution). Mobile agent computations in grid environments, including resource discovery [97, 178], computation surveillance [78], and installation of software required by the grid computation [178], are dominated by idempotent computations. This means that enforcing exactly-once is not a correctness concern for the mobile agent computation. However, executing a mobile agent stage more than once is wasteful when

[4]While the paper describes the protocol in terms of Byzantine failures, the message flow is the same for fail-stop crash failures.

resources are shared between grid consumers. Consequently, in this dissertation, the exactly-once property is considered an efficiency concern and not a correctness concern.

Primary-backup protocols depend on the ability to detect failures of the primary to implement failover. Failure detectors are defined by two properties: accuracy and completeness [33]. Accuracy limits the number of false suspicions of failures while completeness requires that a failure is eventually suspected. A perfect failure detector is one that offers strong completeness and strong accuracy, and thus does not make false suspicions [33]. In a grid environment where places are subject to variable computing resources and periods of unstable network communication, perfect failure detectors are impossible to implement.

An approach to overcome the problems of false suspicions caused by network communication failures is to block the mobile agent computation until a period of synchrony occurs or the suspected place recovers. Blocking, however, increases the time the computations take to complete, and poses a liveness issue if the state of the suspected place cannot be established. Thus, in practice, incorrect failure detections can occur to avoid blocking, causing the primary failover protocol to be invoked when the primary has not failed. As the next sections will elaborate, a primary failover resulting from an incorrect failure detection can cause duplicate mobile agents.

2.4.2 Fault-Tolerant Mobile Agent Systems

We now investigate how mobile agent systems support fault-tolerance with primary-backup approaches. Of particular interest is how various systems handle the tradeoff between blocking the computation and enforcing the exactly-once property when there are network failures.

Fatomas

The Fault-Tolerant Mobile Agent System (Fatomas) [140] has a strong focus on providing exactly-once semantics. Like the approach in [126], Fatomas uses several places to execute each stage, and is thus similar to state machine replication. However, instead of all places actually executing each stage, Fatomas views the execution of a mobile agent computation as a sequence of agreement problems. A consensus algorithm called Deferred Initial Value (or DIV) [43] is used to choose a single place to execute a stage. The flow of a Fatomas execution is given in Figure 2.7. Circles depict places and the number in the circle the stage number. The asterix indicates which place is actually executing the agent. The result of executing an agent at a place is sent using a *reliable broadcast protocol* [76] (depicted by bc) to the places comprising the next stage.

A consensus algorithm starts with each place having an initial value. For a mobile agent computation, the initial value for a stage is obtained by executing the agent. However, having all places in a stage execute the agent breaks exactly-once properties and wastes resources. DIV consensus allows deferring the execution of an initial value until it is required. The advantage of the DIV consensus algorithm is that if the place proposing the initial value does not fail, then the value will be chosen and no other place chooses another value (i.e., executes the stage). If a place fails, the DIV consensus

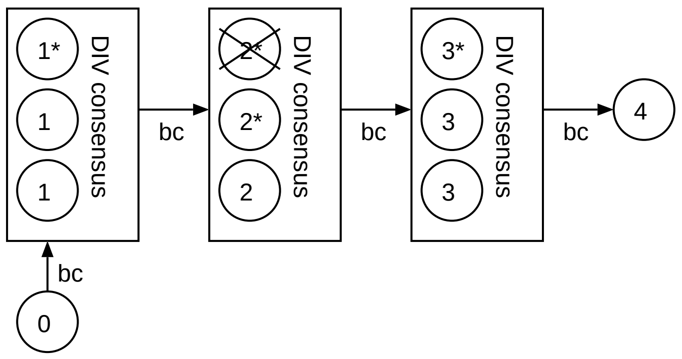


Figure 2.7: Mobile agent fault-tolerance with Fatomas.

algorithm requires another place to propose a value. This is depicted in stage 2 of Figure 2.7, where the place initially executing the agent for stage 2 fails, and another place starts executing. A majority of non-failed places is required for DIV consensus to succeed, in which case all places for the stage agree on which place executed the agent, the result of the execution, and the next places for the stage. When execution of the agent has completed, the result is forwarded to the next stage using reliable broadcast. While reliable broadcast could be externalized, Fatomas has instead extended the DIV consensus algorithm to send the results of its decision to the places in the next stage. This avoids the complexity of having a separate implementation of reliable broadcast in addition to the DIV consensus algorithm.

As mentioned, DIV consensus requires a majority of non-failed places within each stage to agree on the proposed value. If not, the algorithm blocks. This means that for certain communication failures and failures where the majority fails, the agent computation will not make progress until failures recover.

A Transaction Approach

The approach from Rothermel and Strasser [151] uses a fundamentally different approach than Fatomas to provide exactly-once semantics. The approach is based on transactional message queues and the use of atomic commit protocols similar to established standards like X/Open DTP or CORBA OTS.

Each stage in the computation is executed by a *worker*, where the progress of the worker is monitored by a set of *observers*. All places participating within a stage are assumed to fail independently of each other and implement a transactional message queue using stable storage.

The protocol chooses a place to be the worker for the stage. The rest of the places are observers. The worker and the observers retrieve the agent from the message queue. The worker then executes a transaction that executes the agent, puts the resulting agent into the message queues of the next stage, and tries to commit the transaction. If the worker fails to execute the transaction, the observers restart the protocol by choosing a new worker.

Multiple workers are detected by having the observers in stage i sending the identity of the worker that they are monitoring to the places comprising the next stage $i+1$. The observers in stage $i+1$ then use majority voting to decide whether there are multiple workers when the transaction is attempted committed. If there is more than one worker, the observers in stage $i+1$ abort the transaction, otherwise the worker commits the results on the queues.

One of the problems with using an atomic commit protocol such as two-phase commit (2PC) is that it is vulnerable to blocking [21], something which cannot be fully redeemed even by the use of three-phase commit (3PC). The work of Assis Silva and Popescu-Zeletin [164] improves the algorithm by using a different leader election protocol in combination with 3PC. Their approach unfortunately breaks the exactly-once property, although they suggest the use of a *distributed context database* to fix this. The problem with the distributed context database is that it requires replication for fault-tolerance, all adding up to a very complicated approach.

The protocol also only handles network failures when occurring among the members of the places comprising a stage, which may be a problematic assumption if the approach is deployed in Internet environments.

Checkpoint Approaches

The Mobile Agent Framework [41] makes a fundamental distinction between communication within a local area network and between local area networks. The idea is to partition the network on the agent itinerary into domains, where each domain spans a local area network. Each domain hosts a *checkpoint manager* (CM). The tasks of the CM is to keep track of which agents belong to its domain, and to keep checkpointed state for each agent. An agent may at any time during execution checkpoint its state to the CM of the current domain. Agents periodically send ping messages to the CM they belong to, to indicate that they are alive. There are three main interactions between the CMs and the agents, all executed within the context of an atomic commit protocol (e.g., 2PC) to ensure that they are performed in a consistent manner:

- Intra-domain migration. The agent is moved from one place to another place within the domain, and the current location of the agent is updated to the CM.

- Inter-domain migration. The agent is moved from the place in the old domain to the place in the new domain. A checkpoint of the agent is made at the CM of the new domain, and the checkpoint of the agent at the CM in the old domain is removed.

- Recovery of an agent. The last agent checkpoint is retrieved from the latest CM, and the current place of the agent is updated in the CM.

This approach is visualized in Figure 2.8. Circles depict agent stages, and the numbers denote the stage number. Circles with 'CM' are content managers. Solid arrows depict an agent migrating from one stage to another stage, and dashed arrows communication from the agent to the CMs.

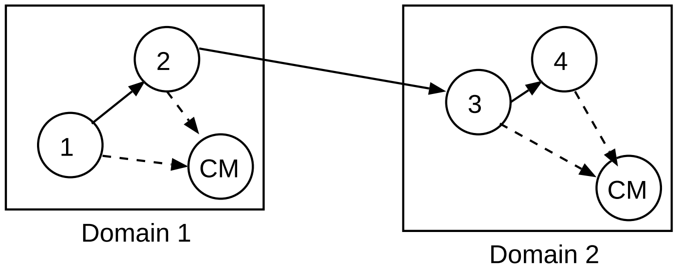


Figure 2.8: Mobile agent fault-tolerance with checkpoint managers.

Handling recovery upon failures is left to the agent program. The CM will not attempt to recover the agent like a primary-backup approach would. The main advantage of this protocol is that by the nature of the atomic commit protocol, exactly-once semantics are preserved upon failures. The protocol assumes that the CM does not crash, since this will cause all agents within the domain to be unable to make progress. In addition, the actual atomic commit protocol is subject to blocking when the coordinator fails during execution of the transaction [21].

A similar approach is the checkpoint based protocol in the Messengers system [64]. Rather than depending on a checkpoint manager, an agent performs local checkpoints to stable storage coordinated by a single *checkpoint coordinator*. The checkpoints are coordinated according to a distributed snapshot protocol that guarantees consistency. Upon failure of an agent, one of the places undertakes the task of initiating the recovery procedure. Recovery discovers which processes are still alive in the system and ask them to roll back to the last consistent checkpoint. The place performing the recovery procedure then retrieves the last checkpoint of the failed agent from stable storage and restarts the computation. The protocol uses the Transmission Control Protocol (TCP) for communication and failure detection, and is vulnerable to network failures with respect to providing exactly-once semantics. The protocol is, however, non-blocking.

NetPebbles

The NetPebbles [128] environment offers a programming model where agents are modeled as scripts that call a series of component interfaces. When calling a component, the script and its state are migrated to the location where that component resides and executes there. After all components have been invoked the script returns to the initial place with the results.

During execution, scripts are subject to a variety of failures, such as a place failing, network failures and component failures. Fault-tolerance is provided by using non-determinism in choosing where to execute the script, and effectively routing the script around failures instead of waiting for the failures to recover. More specifically, the fault-tolerance algorithm works as follows: when an agent moves from one place p to the next p', place p keeps a copy of the agent and starts monitoring the agent at the place p'. If place p detects that p' has failed, the agent is routed to another place p'' and the stage is

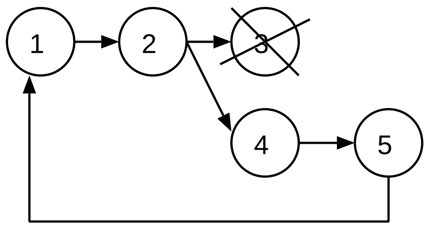


Figure 2.9: Fault-tolerance in NetPebbles.

re-executed. The scheme is depicted in Figure 2.9. Circles depict the script, arrows the monitoring relations, and the number denotes the place identity. Here, a script executes two places, and moves to the third. The second place finds that the third place has crashed, and moves the script to place four instead. After successfully completing place four, the script moves to place five before returning to the initiating place.

A problem here is that the concurrent failure of a place p and the next place p' would block the script, since no place would be able to route the script to another place. NetPebbles solves this by having all the places where the previous stages executed monitor their successor places, where successor places send heartbeat messages back to the preceding places. Heartbeats are sent with a frequency decreasing with the distance between two places, where the distance is measured by the place number. A place p only detects an agent as failed if it fails to receive a heartbeat message from all successor places. Doing so makes NetPebbles tolerate as many concurrent failures as the distance between places the script has visited.

The failure-detection mechanism in NetPebbles may cause a redundant script to be executed upon false detections of failures. However, since a script is assumed to return to the initial place upon completion, duplicate scripts eventually arrive at the initial place where the actions they have performed can, if desired, be aborted. The fundamental problem with this approach is that it requires the actions performed by a component to support rollback, something which is not always possible to ensure if the component makes use of an external tool that has side-effects.

2.5 Summary

We started this chapter by introducing the computational model for mobile agents, and discovered that the performance of migrating the agent is important when competing against the cost of RPCs. We then investigated how various systems implement and optimize the procedure of migrating a process, which revealed two important optimizations called *pre-copy* and *lazy-copy*.

The classic transparent migration approaches often require significant support from the operating system, and depend on a homogeneous software and hardware environment. For this reason, several mobile agent systems implement weak migration, where the state of the computation is managed by the agent itself and does not require support from the operating system.

One of the challenges shared by both process migration and mobile agent systems is providing robustness when there are crash and network failures. In the field of fault-tolerance, such robustness can be implemented by a variation of the replicated state machine or primary-backup approach. We discovered that primary-backup is attractive for mobile agent computations in grid environments, since the approach does not require redundant computations in failure-free runs. We also established that mobile agent computations for the grid do not require the exactly-once property for correctness, but is instead an efficiency concern to avoid wasting resources. The various approaches to mobile agent fault-tolerance differ slightly in how they handle the balance between preserving exactly-once and blocking:

- In Fatomas, exactly-once properties are enforced but for certain failures such as network failures computations may block for an unbound amount of time.
- In the transactional approach from Rothermel et al., the computations also block to enforce exactly-once. This was fixed in subsequent extensions to the protocol, but required an external context database to enforce exactly-once, leading to a very complex system.
- Of the two checkpoint approaches, the first blocks to preserve the exactly-once property, while the second allows duplicate agents to avoid blocking.
- In NetPebbles, exactly-once properties are not enforced during the computation, but since all agents return back to the initial place, duplicate agents can be detected and their effect on the environment rolled back.

The main contributor to the problem of balancing exactly-once and non-blocking properties of a mobile agent computation is the impossibility of perfect failure detection in an asynchronous environment. Thus, before we devise an algorithm for mobile agent fault-tolerance, we need to gain insight in how communication breaks down in the application domain we consider, grid computing.

Chapter 3

Characterizing Wide-Area Communication

Before devising a mobile agent fault-tolerance protocol that is suitable for grid environments, we need experience in the field of grid computations. In particular, we need to study how grid computations behave when deployed in environments where they are subject to crash and network failures. This chapter provides the background for the analysis of the first grid computation algorithm that is presented in Chapter 4.

3.1 Master-Worker

Computational grids provide powerful environments for executing large-scale distributed computations. As an example, consider GriPhyN[1](**Gri**d **Phy**sics **N**etwork) which is a research project to implement a Petabyte-scale computational environment for data intensive research projects. The project, whose requirements are being defined in the context of four current large physics experiments, deploys computational environments called Petascale Virtual Data Grids (PVDGs) to meet the data-intensive computational needs of a diverse community of thousands of scientists spread across the globe.

One of the functions of a PVDG is the reconstruction of "virtual" data, which is data that is derived from the raw or processed output of experiments. Some reconstructions will be large enough to warrant utilizing the resources of several computation farms spread across the Internet. Reconstruction is highly parallelizable, which makes a master-worker computation [162] attractive for reconstruction.

In a master-worker computation, a master process splits the entire computation into a set of N independent tasks. These tasks are then distributed by the master to a set of worker processes that execute the tasks. When a worker has completed a task, the result is sent back to the master process and a new task is distributed to the worker. The computation completes when the master knows the result of all N tasks. The master and worker processes are typically deployed on different computers to increase task parallelism, so the master and worker processes often use network communication to orchestrate the computation.

[1]GriPhyN homepage: http://www.usatlas.bnl.gov/computing/grid/griphyn/

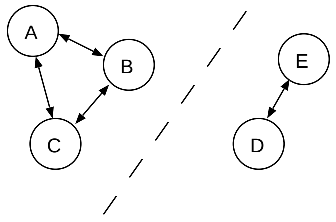


Figure 3.1: Partition with two components.

For network communication, we distinguish between communication across two types of networks in this dissertation. A *local area network* is comprised of multiple computers, or network nodes, typically connected by one or more network switches, where at least one switch is connected to a network router for Internet access. Multiple connected local area networks comprise a *wide area network*. Wide area networks are vulnerable to *network partitions*. A network partition occurs when two sets of non-faulty processes on two different network nodes are unable to communicate with each other due to problems at the network communication layer. Two network nodes, where at least one node can communicate with the other, are said to be in the same *connected component* of the network. Hence, a partitioned network has more than one connected component. Consider the example in Figure 3.1, which depicts a partition with two components. Nodes A, B and C form one component, while nodes D and E form the other component.

There has been a large amount of work on the problem of master-worker computations, for instance [20, 162, 180]. However, few have investigated protocols for master-worker computations deployed in networks that can suffer from network failures and thus, network partitions. The problem was identified as amenable to *partition aware solutions* [12], which are applications that can make progress even if the network is partitioned. It has been argued that group communication services provide a convenient framework for writing such applications, although the exact details of group communication may have a significant impact on the design [11, 173].

There has been recent work on the complexity of solving a variation of master-worker computations. This work has produced an algorithm named *AX* that is based on group communication services [66]. Before we go into detail on how AX works in Chapter 4, we start by investigating how group communication behave when deployed on the Internet. Given the general applicability of group communication systems for implementing fault-tolerant systems, this investigation is also important for determining whether group communication is an attractive abstraction for implementing mobile agent fault-tolerance.

3.2 Wide-Area Group Communication

Prior to investigating the applicability of group communication for master-worker computations, this section gives a background on group communication and approaches to dealing with network failures. We then investigate the behavior of Internet packet

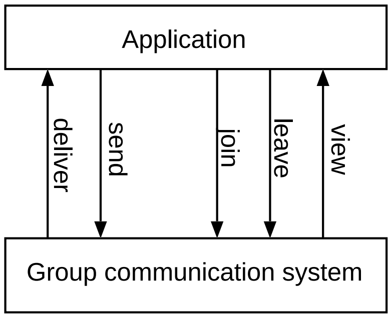


Figure 3.2: The application programming interface of group communication systems.

routing, and how packet routing impacts group communication systems. Finally, we present our results from experiments on the performance of group communication.

3.2.1 Group Communication Systems

Group communication systems [25] typically provide two services to applications: i) reliable multicast communication among processes that are organized into groups, ii) membership information on processes in groups.[2]

The application programming interface that is exposed by most group communication systems is shown in Figure 3.2, where the arrows denote the direction of control flow. A *group* is a set of processes that together comprise the *members* of the group. Group communication systems offer applications two abstractions for membership management: A process becomes a group member by requesting to *join* the group; it can cease being a member by explicitly requesting to *leave* the group or, implicitly, by failing.

Each group is associated with a name. Processes multicast to group members by sending a message to the group name with the *send* operation. The group communication service delivers the message to the group members through the *deliver* operation (some systems allow applications to register callbacks for handling the deliver operation). *When* the deliver operation is issued depends on the ability of the members to communicate with other group members, which we elaborate below.

Group communication systems are *view oriented*, which means that they provide membership information and deliver messages in a well-defined order among all members. Group communication systems differ in the details on how they implement such an ordering.

The task of a *group membership service* is to track the membership of the group as it evolves over time. When the membership changes, the application processes are notified at an appropriate point in the delivery sequence. The output of the membership service is called a *view*, which consists of the list of the current members in the group and a unique identifier that allows the application to distinguish the view from other views

[2]Some group communication systems do not provide i), but they always provide ii).

with the same list of members to order the sequence of views. Views are used in two main ways:

1. Since the members agree on the content of the views, they can deterministically assign roles to each other or do leader election without using further communication. For example, the first process in the membership list can serve as a coordinator for all the processes in the membership list.

2. Consider any two processes that are both in the membership of a view v when they both install the same new view v'. The group communication system ensures that both processes have delivered the same sequence of messages while in view v.

A challenge in group communication systems is how to provide appropriate view semantics upon network partitions, and there are generally two approaches for handling partitioned operations: *primary partition (or component) group membership* and *partitionable group membership.*

Primary Partition Group Membership

Primary partition group membership services require that only one view exists within the group. Thus, either all members of the group install the same view, or they block. Consider the following scenario: a group membership service running at process p signals to the other group members that it is ready to install a new view. However, before installing the view, p detects that the rest of the group members are partitioned away. Thus, p cannot install the new view. The rest of the group can, however, install the view since it was signaled to do so by p (the resulting view in p is called a *hidden view*). When this occurs, p enters a recovery protocol that has two outcomes: i) p is reconnected with the primary partition, or ii) p blocks or terminates.

There are many group communication systems that implement primary partition group membership. In Isis [149], a process that finds itself to be outside the primary partition terminates to ensure that only the processes of the primary partition survive. Other systems, such as Phoenix [120,121], block the process while the network partition takes place. Thus, both Isis and Phoenix allow some progress to be made while the partitions are occurring. Other group communication systems, such as Consul [127] and xAMP [150], only ensure safety properties of views as long as there are no partitions.

Primary partition group membership is commonly used for applications that require globally shared state to be updated in a consistent manner [50, 59] since a single view ensures that data access can be serialized across the view members. However, this comes at a cost of availability (and for some applications, scalability), since groups with members across several local area networks will only harvest the resources of a single connected component when there are network partitions.

Partitionable Group Membership

In contrast to primary partition approaches to group membership, a *partitionable group membership service* [10, 95] is designed to operate in wide area networks. It supports writing *partition-aware* applications, which are applications that can make progress despite network partitions [11].

Partitionable group membership protocols monitor the network connectivity using an unreliable failure detection service [33]. The reported failures are used to instigate view changes. When the failure detection service stabilizes such that communication is possible among all the processes in the connected component, a new view can be delivered to the processes in the connected component. Different partitionable group communication systems have different delivery rules associated with messages that are sent while the failure detection service is stabilizing; The reference [173] contains a summary of the rules of six different protocols. Some systems allow messages to be sent during periods of unstable failure detection, while others do not allow such communication.

3.2.2 Moshe

The algorithm we study in Chapter 4 requires group communication, and master-worker has been advocated as an application that fits well with partitionable group communication [12]. In this context, we study the Moshe [95] partitionable group communication system. The reason for choosing Moshe is that it has been specifically designed for scalability and performance in a wide area network. Moshe has been subject to real-world testing by running it in a wide area network and measuring the performance. Testing has confirmed that Moshe performs well [95].

There are three main design features in Moshe that set it apart from other partitionable group membership systems. First, to ensure scalability, Moshe is structured according to a client-server architecture. Second, to reduce the network load during times when the network is unstable, Moshe avoids sending obsolete views. Finally, Moshe is built to perform well in the common case where the network is stable, and only requires one round of communication for this case.

The interface of the membership service is depicted in Figure 3.3. A *Notification Service* (NS) implements an unreliable failure detector that is used to determine the status of processes and network links. *Clients* use the NS when requesting to `join` and `leave` groups. The NS sends an `event` to the *Membership Servers* (MS) for every group `join` and `leave`, and piggybacks failure detection information onto these event messages. This way the MS maintains the group membership information according to the input from its NS. There is one NS per MS, and there may be several clients connected to each MS, but a client is only connected to a single MS.

The MS sends two types of messages to its clients. The `startchange` message indicates that a membership change is about to happen, and includes the suggested group members for the new view. The `view` message informs the client of what has been decided as the current view for the group, and also includes the group members of the group in the message.

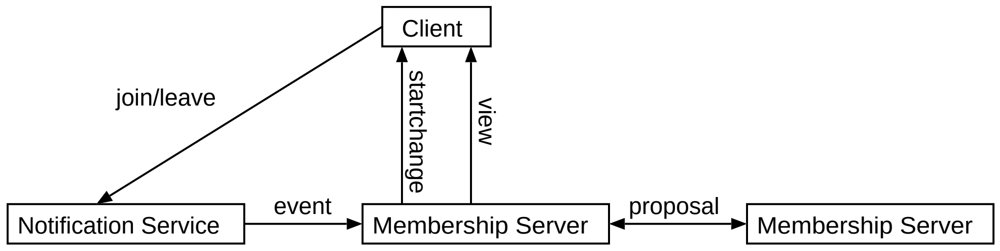


Figure 3.3: Moshe client-server interface for group membership.

The One-round Algorithm

The Moshe algorithm starts when the NS sends an `event` to the MS. When an MS receives an `event` from its NS, it notifies all clients of the MS that a membership change is taking place with a *startchange* message. Subsequently, the MS sends a *proposal* message to all other MSes it knows about. The `proposal` message includes the same suggested view as the clients receive in the `startchange` message. The MS then waits for a `proposal` from all the other MSes, and if they agree on the view, a `view` message is sent back to the clients with the same members of the group as in the previous `startchange` message.

Agreement among the MSes on the state of the group members can thus be reached with one round of communication. There are, however, scenarios that will cause the one-round algorithm to block.

The Slow Algorithm

Since `proposal` messages may be submitted simultaneously by several MSes during times when the network is unstable, there may be disagreement on the state of the network. Moshe detects this condition by keeping track of the last `proposal` message sent by each MS. If, during the one-round algorithm, an MS detects that another MS has a different last proposal than itself, the one-round algorithm is declared as being blocked, and Moshe enters the slow algorithm.

The slow algorithm works similar to the one-round algorithm by MSes exchanging `proposal` messages. The major difference compared to the one-round algorithm is that the submission of `proposal` messages is synchronized. Upon detecting a blocking situation, the MS multicasts a `proposal` message tagged with marker that indicates that the slow algorithm is being run. The message includes a *proposal number* that is used in the following way: If a MS receives a `proposal` with a proposal number larger than its recorded proposal number, the server records the proposal number in the message and sends a new `proposal` with this number to the other MSes. If, however, a NS sends an `event` to a MS, a new `proposal` message will be sent with a proposal number greater than the highest known proposal number. This means that unless an NS sends an `event`, every MS will eventually send a proposal with the same `proposal` numbers and the same view members, in which case the algorithm terminates. The slow

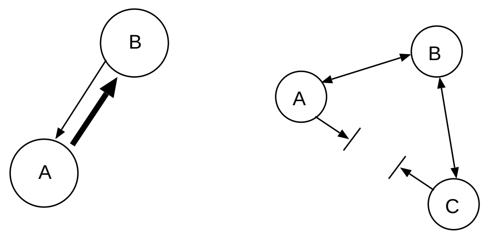


Figure 3.4: Asymmetric and non-transitive communication example.

algorithm terminates similar to the one-round algorithm (i.e., when all MSes agree on the state of the group).

3.2.3 Blocking

Requiring agreement on views among the group members in Moshe implies that all group members agree on which group members are part of the view. Disagreement on the state of the view is primarily due to *non-transitive communication* among three or more group members or *asymmetric communication* among two or more group members. Consider the graphs in Figure 3.4 that depict asymmetric and non-transitive communication. Solid circles denote network nodes and arrows denote the communication links between nodes. Arrow width indicate the level of connectivity.

Asymmetric Communication. In the graph on the left in Figure 3.4, network nodes A and B try to communicate with each other. Node A can communicate with node B with good connectivity and bandwidth. Communication from node B to node A, however, is limited and thus causes packets sent from node B to A to get lost. This failure scenario is an instance of asymmetric communication.

Non-transitive Communication. In the graph on the right in Figure 3.4, network nodes A, B and C try to communicate with each other. Node A is able to communicate with B but not with C, and node C is able to communicate with B but not with A. Thus, node B is the only node that is able to symmetrically communicate with A and C (although A and C can communicate *through* B). This failure scenario is an instance of non-transitive communication.

To summarize, when a connected component's communication ability relation lacks symmetry or transitivity (i.e., it is not a *clique*), some blocking of communications occurs until the component forms a clique again.

3.3 Internet Packet Routing

There are many reasons why asymmetric and non-transitive behavior occur. The main reason is due to the way packets are routed on the Internet. Although a complete background on Internet routing is beyond the scope of this dissertation, the following section gives an overview on the Internet architecture and issues related to packet routing.

3.3.1 Internet Organization

The Internet is organized as a graph of independently operating Autonomous Systems (ASes) that communicate with each other. Typical ASes range from large Internet Service Providers (ISP) to a university campus or corporate networks. An AS defines its own administrative domain with respect to routers and routing policies, and communicates routing information with other ASes.

The ASes forming the Internet are organized in a three-tier system as depicted in Figure 3.5. Arrows denote communication paths between ASes. The ASes of the first tier form the “backbone” of the Internet (also referred to as the Default Free Zone, or DFZ) and consists of large national or international ISPs, for instance AT&T, C&W, Qwest, and Sprint. Tier 1 ASes know about all other ASes within Tier 1, which makes all possible communication paths between ASes within this Tier a fully formed mesh. Unlike Tier 2 providers, Tier 1 providers do not buy connectivity from other providers.

Tier 2 providers are typically ASes covering large geographical regions, for instance the southern part of Norway. Tier 2 providers buy upstream connectivity from one or more Tier 1 providers. In theory, there exists a class of Tier 3 providers that are smaller corporations, local ISPs or user networks which buy upstream connectivity from one or more Tier 2 providers. However, separating Tier 2 and Tier 3 networks is becoming increasingly hard because of the mesh-like organization of the Internet. For instance, since the Internet is not organized in a hierarchy, a Tier 2 AS may connect to several other Tier 2 ASes or several Tier 1 ASes, resulting in multiple communication paths from one AS to another. ASes that are connected through more than one other AS are referred to as a *multi-homed* AS. There are generally two reasons for multi-homing:

1. Backup. If the route to one AS is down, a provider still has a mean to reach the Internet through another AS.

2. Load-balancing. Having route redundancy allows using more than one route for sharing traffic across several backbones.

Due to the lack of structure in the Internet outside Tier 1, ASes in Tier 2 and Tier 3 must provide each other with routing information that states what networks are reachable from a particular AS. The process of exchanging such information is called *peering*, and the protocol used for peering is called the Border Gateway Protocol (BGP-4) [146].

3.3.2 Border Gateway Protocol

Routing information is typically maintained within an AS by a network service provider, for instance owned by a University or an ISP. Routing information is shared between

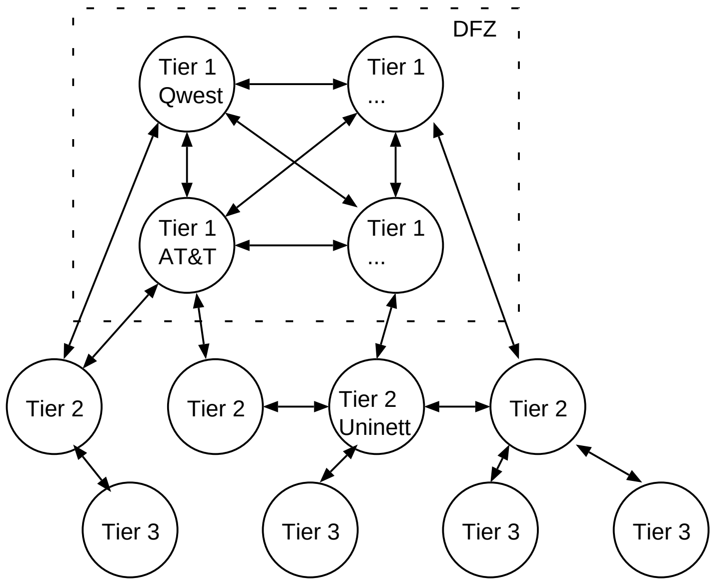


Figure 3.5: The organization of Autonomous Systems (ISPs) forming the Internet.

ASes through BGP-4. BGP-4 uses TCP for communication with BGP-4 routers in other ASes. Internally within an AS, several alternative IP-level protocols are used for communication, such as Open Shortest Path First (OSPF) and Interior Gateway Routing Protocol (IGRP) [81]. Since the internal protocols work across a limited set of participants, complete topology knowledge within the AS is easy to attain. For instance, OSPF periodically floods the network of internal routers with complete information about link connectivity and the configuration of the local interface. This way, routers running OSPF build a shortest path topology view to every other router within the AS. Intradomain OSPF routing has been shown to achieve a stable network view in a few hundred milliseconds [161]. Typically, routers within an AS form a full mesh. However, the number of ASes (reported to 24,000 by the end of 2006 in [83]) prevents the same level of connectivity and information sharing between ASes compared to intradomain routing.

Given the current (and large) number of ASes, the scalability of the Internet depends on how routing information is aggregated. BGP-4 is called a *path-vector* protocol because it aggregates information about the path going from the source to the destination. Routing information consists of Internet Protocol (IP) addresses in contiguous blocks of 32 bits, called prefixes, and a mask length. For instance, the prefix `129.242.16.0/24` owned by the University of Tromsø refers to the 256 IP addresses in the interval from 129.242.16.0 to 129.242.16.255.

A simplified version of the state machine for a BGP-4 session is depicted in Figure 3.6. To explain how the state machine works, consider the example on how a given BGP-4 session works when three ASes use BGP-4 to exchange routing information. Assume AS1 has established a session to AS2, AS2 to AS1 and AS3, and AS3 to AS2. Initially, AS1 starts exchanging its active routes by announcing to its peers, in this case AS2,

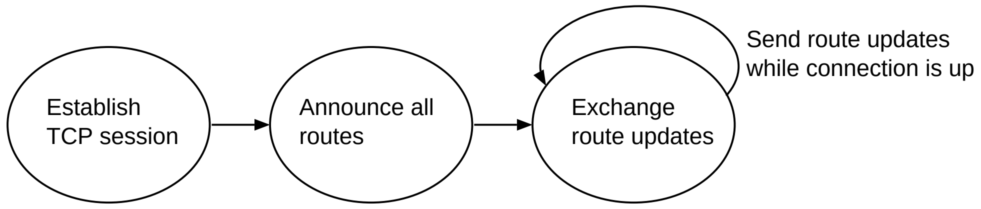


Figure 3.6: Simplified BGP-4 session state machine.

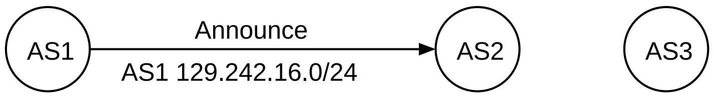


Figure 3.7: AS1 announces its routes to AS2.

that it owns a given network. We use the network `129.242.16.0/24` as an example here in Figure 3.7.

Upon receiving this information, AS2 includes its own routing information in the announcement and relays it to AS3. When A3 receives the announcement, it knows that it can reach the network `129.242.16.0/24` through AS2, as depicted by Figure 3.8. After the protocol has completed, the routing tables at AS3 will include information that the network `129.242.16.0/24` is available from AS1 and AS2, as depicted by Figure 3.9.

Upon receiving a route advertisement, a router must determine whether to use the advertised path or not, and if it is used, whether to propagate information about the path to its peers through a route update.

Handling a route advertisement involves three actions:

- Which routes should be considered used (*import policies*).
- Which path should be used for routing (*path selection*).
- What information should be advertised to other routers (*export policies*).

Unlike the routing within an AS, BGP-4 does not use shortest-path analysis for its routing decisions. Rather, BGP-4 supports a set of policy parameters used to discriminate between multiple paths. A *local preference* can be set on a path to favor it over another path to the same destination prefix. The *multiple exit discriminator* is set on advertisements to neighbor ASes and determines how incoming traffic is preferably

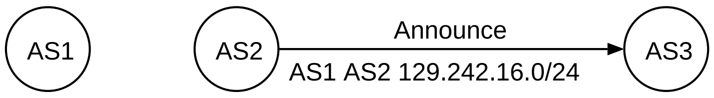


Figure 3.8: AS2 announces its routes to AS3.

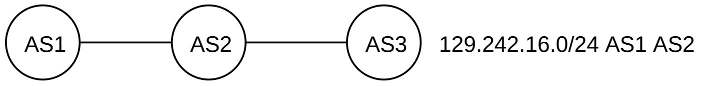


Figure 3.9: AS3 routing tables after announcements from AS1 and AS2.

routed. Many routers are configured to prefer the shortest path when there are multiple paths to the same prefix. This feature is frequently exploited to limit the amount of incoming traffic to a router. The technique is known as *AS prepending*, where an AS prepends itself to a path and sends a route update, with the intent that neighbor ASes disfavor the longer route with the prepended AS.

There are two types of route updates in BGP-4, i) route announcements, and ii) route withdrawals. A route announcement means that a router has discovered or decided that it prefers another path. A route withdrawal means that a router has decided that a network prefix is no longer reachable and withdraws it from further communication. Withdrawal can either be explicit, where the prefix is withdrawn by a withdrawal message sent on the BGP-4 session, or implicit, where the current routing paths are overridden by a new announcement message from a router.

For instance, if we assume that AS1 loses communication with A2, and then resumes communication again, AS2 will view this as an implicit withdrawal since it gets a new announcement from AS1 without a preceding explicit withdrawal message. AS3, however, is notified with an explicit withdrawal of the prefixes owned by AS1 from AS2, and is subsequently sent updates about the prefixes being reachable when AS1 resumes communication.

3.3.3 Routing Instability

Routing instability between ASes happens when the network communication fails between ASes and causes route updates in BGP-4. The cause for network communication failures is typically transient problems with (physical) communication links, router software problems or misconfigured routers. Route updates during instability form three categories, i) forwarding instability, ii) policy changes, and iii) pathological updates.

Based on these three categories, Labovitz et al. have established a well-known taxonomy for analysis of BPG routing instabilities [99, 101]:

- **WADiff**: A route is explicitly withdrawn and no alternative routes to the destination prefix exist. This is classified as a forwarding instability.
- **AADiff**: A route prefix is implicitly withdrawn and updated with routing attributes indicating a different route. This is classified as either a pathological update or a forwarding instability.
- **WADup**: A route prefix that has previously been known as failed is announced as reachable again. This is classified as a forwarding instability.

- **AADup**: A route prefix is implicitly withdrawn and replaced with a duplicate of the original route. This is classified as a policy fluctuation (when other attributes than the route are updated) or a pathological update when only the route path is updated (since BGP-4 should only send updates when there are actual changes in the route).

- **WWDup**: A currently unreachable prefix causes repeated withdrawals or duplicate announcements of routes. This is classified as a pathological update.

During route instability, the state of the routes is known as *converging* while routers determine stable paths. During convergence, several router update messages may be transmitted which in turn generates additional overhead on the Internet infrastructure both in terms of bandwidth and router hardware. Experience with wide area network backbones has shown that a router that gets overloaded during pathological updates frequently causes a phenomenon termed *route flapping* to occur [101]. Route flapping happens when an overloaded router is marked as unavailable by other routers, where the other routers subsequently submit route updates to their neighbors. The unavailable router eventually manages to respond, causing new route updates from the other routers that have previously marked the router as unavailable. The redundant route updates may propagate throughout the Internet and eventually cause significant outages for millions of network nodes [99]. Besides pathological updates, route flapping can also be caused by route policy changes. A study has shown that there exist policy configurations that exhibit persistent route oscillations, where a route never converges even when there are no changes in the actual network topology [177].

Countermeasures

A technique called *route flap damping* is used by most Internet routers today to decrease the effect of route flapping [179]. It works by associating a penalty value with all prefixes announced by a given neighbor. Whenever the state of the route changes, the penalty is incremented by a fixed value. Over time, the penalty value is decayed exponentially, making the penalty value a measure of the instability of a route. Routers are typically configured with two threshold values related to the penalty value, i) the suppress threshold where a route penalty exceeding this threshold is suppressed by the routers, ii) the reuse threshold that determines when a previously suppressed route should be reused again. Although route flap damping does not prevent BGP-4 route oscillations, it makes them run slower, which improves the overall stability of the Internet routing system. However, the work in [123] shows that route flap damping can increase the convergence time to over an hour even on relatively stable routes.

Another solution for addressing route instability is using route aggregation. With route aggregation a set of prefixes are combined and announced in a single route announcement or “supernet” announcement. The idea is to reduce the number of prefixes that are visible to increase the stability in the Internet. An AS will establish a path to an aggregated prefix as long as one of the path components within that prefix is stable. The problem with route aggregation is that orchestration of aggregation in an Internet without hierarchy is hard in practice. Multiple backbones owned by different

ISPs require close cooperation, something which may be hard when routing policies are governed by commercial contracts [82]. In addition, the growing number of multi-homing ASes makes aggregation hard, since routers need to maintain an extra prefix for each multi-homed AS [101].

Lixin Gao et al. have studied the problem of policy fluctuations that cause persistent route oscillations in [63]. In their work, an abstract model of BGP-4 is used to formalize a set of guidelines ASes should apply when configuring the import policies of routers. They found that by confining the set of policies that can be used by routers, route convergence is guaranteed without the need for global coordination.

In another work, the properties of BGP-4 route convergence were analyzed statically in order to establish which routes were safe [71]. Among other questions asked in this study are the questions of *reachability*, *asymmetry*, and *solvability*. The study formalizes a model of BGP, and based on this model show that reachability, asymmetry and solvability are all *NP-complete* or *NP-hard*. This study indicates that even given complete knowledge of import policies in a BGP system, static analysis does not guarantee global convergence, and that adaptive techniques such as route flap damping or other heuristics are important practical solutions to the route convergence problem.

Packet Loss

During periods of network link oscillations caused by forwarding instabilities, studies have shown that BGP-4 uses a significant amount of time (up to several minutes, sometimes as long as 30 minutes [98, 100]) to establish which routes are up and down. During this time of network route convergence, network packets routed on faulty paths will be lost, and result in communication failures between two ASes. For communication based on TCP, such packet loss can cause active connections between two network nodes to time out and disconnect if one node is waiting for data acknowledgments.

It has been shown that the amount of packet loss during instabilities can be high and cause dramatic performance degradations for communication over the Internet. For instance, in a study on TCP bulk transfer throughput [131], the TCP response time is shown to drop significantly once packet loss exceeds approximately 30%. This result indicates that such packet loss is likely to cause similar performance impact on protocols besides TCP that rely on point-to-point communication.

3.3.4 Asymmetric and Non-Transitive Communication

When routing is stable, filtering the amount of routing information provided by each AS allows BGP-4 (and thus the Internet) to scale effectively. However, as the previous section has shown, this filtering for scalability comes at a cost, since BGP-4 does not maintain information that is important for routing failure resilience, and thus, for preventing packet loss and communication failures.

What actually happens in terms of ASes may be as depicted in Figure 3.10. Solid circles denote network nodes and dashed circles denote Autonomous Systems. Arrows denote the communication path between nodes and ASes. In the right part of the figure we have a non-transitive communication failure. Traffic between A and B is routed

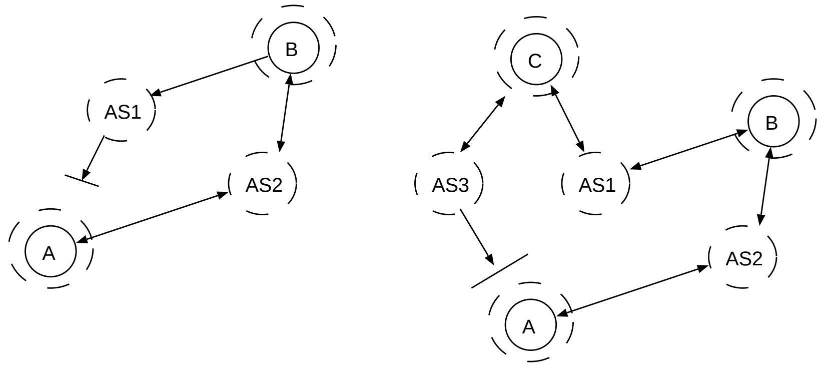


Figure 3.10: Asymmetric and non-transitive communication example.

through AS2, traffic between B and C is routed through AS1, whereas traffic from C to A is routed through AS3. Assume there is a route instability that causes the path from AS3 to A to fail. During route convergence there is no is advertised route between AS3 and A so traffic does not get routed from C to A.

In the left part of the figure we have an asymmetric communication failure where traffic from A to B is routed through AS2 and traffic from B to A is routed through AS1, for instance due to policy configuration governed by a commercial contract stating which routes B should use to A. The route from AS1 to A then fails and during route convergence no packets go from B to A.

Experience from studies of large ISPs show that asymmetric routing is very common on the Internet [133]. Sometimes as much as 50% of the routes are asymmetric [134]. Although unrelated to BGP-4 routing failures, asymmetric network connectivity is frequently employed as a feature when connecting leaf nodes to ISPs with an Asymmetric Digital Subscriber Line (ADSL). ADSL is based on the assumption that leaf nodes run applications requiring very little upstream traffic compared to downstream traffic, and is specifically tailored to web-browsing or other "download" applications. However, if application protocols run on leaf nodes and assume that communication is symmetric, problems may arise when the actual upstream bandwidth is orders of magnitude slower than downstream.

3.4 Wide-Area Group Communication in Practice

We have so far established that the actual frequency and duration of route divergence in BGP may be a major contributor to the amount of blocking because of the resulting non-transitive or asymmetric communication. A study on Internet path faults has reported that 90% faults last less than 15 minutes, and that 70% of faults last less than 5 minutes [49]. Other studies show that BGP-4 routing convergence can take as long as 30 minutes (e.g., [100] and [98]). However, work in analyzing BGP routing has revealed that most routing instability stems from a small number of unpopular destinations and that the majority of BGP routes among popular sites are reasonably stable [147]. Hence, it would seem that the actual blocking due to BGP routing instability should be small enough to be neglectable for wide-area group communication. In addition, previous research in wide-area group communication showed that the blocking due to non-transitive and asymmetric communication was over 20 seconds only in 1.5% of the measured cases in one configuration and in 0.35% of the cases in another configuration [95].

Summarizing:

- The Moshe wide-area group communication system blocks upon non-transitive and asymmetric communication.
- Non-transitive and asymmetric communication is caused by Internet routing failures.
- Previous work on wide-area group communication indicate that the amount of asymmetric and non-transitive communication is relatively low.

The master-worker algorithm we have chosen to study, AX, requires group communication. While AX is optimal in terms of the number of redundant task executions, it is likely that the performance of this algorithm to some degree depends on the performance of the underlying group communication system.

A problem with the results from wide-area group communication presented in [95] is that they include only five participating networks. Restricting large scale master-worker computations to less than six networks is undesirable, so we need to investigate the impact of increasing the number of networks has on the amount of blocking for group communication, as well on the number of partitions we encounter. Hence, in Chapter 4, before we investigate the actual performance of algorithm AX, we need to investigate the following two properties of wide-area group communication given a larger number of participating networks than in [95]:

1. The number of network partitions.
2. The amount of blocking due to asymmetric and non-transitive communication.

Given the lack of accurate existing models of network connectivity which would allow us to analyze wide-area group communication behavior, our investigation and performance measurements are based on analysis of network traces. Using trace data means that it is difficult to generalize from our findings to other network configurations. Hence, we also give a simple model for the expected behavior of the algorithm that uses the group communication system given metrics that can be measured. We show via simulation that the model has predictive value.

3.4.1 Trace Data

We used two different sources for trace data sets. The first data set comes from the Resilient Overlay Networks (RON) project [4]. RON is an application-layer overlay on top of the existing Internet routing substrate. The nodes that comprise RON monitor the function and quality of the Internet paths among themselves, and use this information to decide whether to route packets directly over the Internet or by way of other RON nodes. The RON project uses a testbed deployed at sites scattered across the Internet to demonstrate the benefits of the architecture. Since the probing reported in the trace data referred to in [4] is not frequent enough for our analysis, we used another trace that was kindly collected and supplied to us by the RON group. In this trace, there are sixteen nodes that are spread across the United States and Europe. Each pair of nodes probe each other via User Datagram Protocol (UDP) packets once every 22.5 seconds on average. To probe, each RON node independently picks a random node j, sends a packet to j, records this fact in a log, records if there was a response, and then waits for a random time interval between one and two seconds. The logs from each machine are then collected and merged into a single trace.

All traces from the RON project are structured as multiple lines where each line has the following format:

```
source dest ron send1_time rec1_time send2_time rec2_time
```

Here, `source` and `dest` denote the source and destination host, `send1_time` the time right before data is sent from the source, `rec1_time` the time when the packet is received at the destination. `send2_time` and `rec2_time` indicate the time when the packet was sent from the destination and received at the source, respectively. The `ron` parameter is used to specify whether the packet was routed by RON or sent directly on the Internet, and is thus not used by our simulator.

From the RON trace, we generated a directed *communication graph* with the RON nodes as the vertices in the graph. An edge is drawn from node i to node j if a process at j can successfully receive messages from a process in i. A node i may be marked as *crashed*, which indicates that the process at node i is crashed.

The graph is initially connected and all nodes are marked as not crashed. A node i is marked as crashed if the log indicates that i did not send a probe for 5 minutes. We chose the value of 5 minutes based on the result that 70% of routing faults last less than 5 minutes [49]. Once i subsequently sends a probe, the node i is marked as not crashed. If the trace shows that the last three messages (either probes or responses) a node i sent to j were not received by j, then the directed edge from i to j is erased. The edge from i to j is added again when the trace records j having received a packet from i (either a probe or a response). The trace contains continuous probing for the two week period from August 2 through August 16, 2002.

Group communication services for wide area networks can be factored to run on top of specialized failure detector services [95], and so it can be hard to come up with a general connectivity model that would predict how group communication services would behave in any given situation. One obvious model, however, is based on TCP connectivity. UDP connectivity is clearly worse than TCP connectivity since packets can be lost due to congestion and there is no retry (unlike TCP). We chose the method of declaring a link down only when three packets are lost to make the connectivity graph a more conservative estimator of how group communication services would perform.

The second set of Internet communication traces was collected by Omar Bakr and Idit Keidar. They studied the running time of TCP-based distributed algorithms deployed in a widely distributed setting over the Internet. They experimented with four algorithms that implement a simple and typical pattern of message exchange that corresponds to a communication round in which every host sends information to every other host. The traces we analyzed are described in [13].

The information was collected from universities and commercial ISP hosts. The machines were spread across the United States, Europe, and Asia. Each host sent a `ping` packet (using the Internet Control Message Protocol, or ICMP) to each other host once a minute. Each process records to a local log the ping packets it receives from other processes.

Our simulator used the RON file format as the canonical network trace format since this format has the least redundancy of information compared to what we needed in our simulations. Since the `ping` traces from Keidar and Bakr were structured completely differently, we wrote a simple converter from ping traces to the RON file format.

The communication graph is constructed in a similar way as was done for the RON traces. The graph is initially connected. The edge from a process i to another process j is removed when 3 minutes elapses without j having received a ping from i. The edge is put back when j finally receives a ping from i. There is not enough information in these traces for us to be able to mark a node as crashed, and no node is ever marked as having crashed.

There were three traces generated in total: one with nine nodes, one with eight nodes, and one with ten nodes. We denote these three traces as *exp1*, *exp2* and *exp3*. Each trace recorded more than three days' worth of probing. In exp1, two links had high loss rates: one from National Taiwan University to a commercial site in Utah sustained a 37% loss rate, and another from National Taiwan University to a commercial site in California sustained a 42% loss rate. In exp3, links from National Taiwan University and from Cornell University had high loss rates. Because of these high loss rates, the study by Bakr and Keidar [13] also considered the subset of exp1 with the process at National Taiwan University removed (they did not analyze exp3 in their paper). We did not remove the data concerning National Taiwan University and Cornell from our traces because leaving it in represented a real-world situation for grid computations.

3.4.2 Trace Analysis

First, we would like to know how often partitions happen and what duration they have. A partition occurs when the communication graph contains more than one component, and all components contain non-crashed nodes (recall Figure 3.1).

The partition data from the traces is shown in Tables 3.1. Each partition is identified by when it started in the trace and how long it endured. The percentage of time during which there was a partition in each trace is reported in Table 3.1. Summarizing,

1. There are eleven partitions in the RON trace. Nine of the eleven endured for less than 5 minutes, and the two others endured for over 5 hours. For over 96% of the trace there was no partition.

2. There are two partitions in the exp1 trace. One lasted for 5 minutes and the other lasted for over an hour. For over 99% of the trace there was no partition.

RON (16 nodes): F=18%, Pct Partition time = 3.9%				
	Non-partition	Partition period		
	period	Start time	Duration	Two-clique time
1st	60,843.5	60,843.5	2.4	100%
2nd	259,966.6	320,812.5	1.7	100%
3rd	290,407.9	611,222.1	56.4	100%
4th	41,421.3	652,699.8	27,639.6	98%
5th	224,965.6	905,365	70.6	0%
6th	16,591.4	922,027	27.9	0%
7th	149,180.1	1,071,235	98.2	100%
8th	7,995.6	1,079,328.8	22.4	100%
9th	912	10,80263.2	95.9	100%
10th	60,003.8	114,0362.9	92	100%
11th	49,995.9	1,190,450.8	19149	94%

exp1 (9 nodes): F=32%, Pct Partition time = 0.89%				
	Non-partition	Partition period		
	period	Start time	Duration	Two-clique time
1st	74,520	74,520	3,720	82%
2nd	21,360	99,600	300	100%
3rd	360,900	-	-	-

exp2 (8 nodes): F=5%, Pct Partition time = 0%

exp3 (10 nodes): F=46%, Pct Partition time = 0%

Table 3.1: Partitions in the four traces. Time values are given in seconds.

3. There were no partitions in the exp2 or exp3 traces.

Since there is no information in the three traces about crashes, the partitions time reported for exp1 is only an upper bound.

All partitions that we found in the traces resulted in exactly two connected components. In all but one of the partitions, one of the components contained exactly one node; in the case, which is in the RON trace, the smaller component contained two nodes (one of these two nodes subsequently crashed during the partition). This two-node component was symmetrically and (by definition) transitively connected.

Second, we would like to know how often communication is not symmetric and transitive. At any point in time, each process i is in a connected component of the communication graph. For a process i, let $c(i)$ be the fraction of time during a trace in which its connected component is a clique, and P the number of nodes in the graph. We then compute:

$$F = 1 - \sum_{i \in P} c(i)/P$$

F is the percentage of the time where a view is not transitive or symmetric. Thus, larger values of F indicate that an algorithm using group communication will be less likely to make progress (i.e., communicate) with the rest of the group, because of the lack of a fully formed view. Table 3.1 reports the values of F for the four traces we considered. The values in the duration and the period columns are given in seconds. All traces record a significant value for F, and trace exp2 has the lowest value.

Recall that in all partitions, the smaller of the connected components contained either one or two nodes and was both symmetrically and transitively connected. The larger connected component was not always symmetrically and transitively connected. So, we also list the fraction of time of each partition during which the larger component was symmetrically and transitively connected. For the most part, the larger component's communication graph is a clique.

Looking at these values, we observe that:

- The periods of time during which there is no partitioning are usually quite long. In the RON traces, the shortest such period was about 15 minutes and the longest period was over three days, and in the exp2 and exp3 traces, both of which record three days' worth of probing, there were no partitions.
- When they happen, partitions can endure for a long time. In the RON trace, the longest partition lasted for over 7 hours, and in the exp1 trace the longest partition lasted for over an hour.
- The periods of time during which communications are not symmetric and transitive is significant. It appears that such periods are often due to one troublesome node or link. For example, from the RON traces during partitions, the larger connected component is usually fully connected. Being able to predict which nodes will be the troublemaker, though, may be hard. For example, Cornell participated in all the traces but was troublesome only in trace exp3.

3.5 Summary

We started this chapter by choosing master-worker computations as a way of investigating issues related to grid computations. This led us to an algorithm called AX, which requires group communication. In the following study of wide-area group communication, we found that the performance of such systems typically depends on the amount of asymmetric and non-transitive communication.

The reason for asymmetric and non-transitive communication is due to how packets are routed on the Internet. Internet routing is designed for scalability and simplicity at the cost of effectively handling link failures. Previous research on wide-area group membership indicates that the amount of blocking is small. However, since these results are based on a small number of participating networks, we investigate, through simulation based on network traces, the impact that increasing the amount of participants has on blocking, as well as on the number of partitions we encounter.

The results from these experiments show that, although partitions do not occur frequently, they last long when they occur. Also, the amount of non-transitive and asymmetric communication is quite significant compared to the results presented in earlier research [95].

Based on our findings, the next chapter investigates how algorithm AX performs when run through the same network traces in order to establish how it performs with respect to blocking and whether it performs redundant tasks.

Chapter 4

Master-Worker Computations using Group Communication

The previous chapter analyzed the behavior of wide-area group communication, which is the fundamental building block of algorithm AX. This chapter first describes how AX works. We then run a simulation of AX on the same traces we used for analyzing the performance of wide-area group communication.

4.1 The Algorithm

Algorithm AX is a variation of the master-worker computation we specified in Section 3.1, and considers the case where the amount of redundant task executions is to be kept as small as possible [65, 118, 119]. Algorithm AX solves a problem called *OMNI-DO*:

OMNI-DO Problem The problem of performing a set of N independent tasks on a set of P message-passing processors, where each processor must learn the results of all N tasks.

The difference between OMNI-DO and our previous specification of master-worker in Section 3.1 is that *all* processors must know the results of *all* tasks, not just the master. From a theoretical point of view, this variation is important. Consider a centralized master-worker computation in which only the master needs to know the results of all tasks:

- If the master fails by crashing, the execution of redundant tasks can be minimized by using stable storage and a simple failover mechanism for recovering the failed master.
- If a worker partitions away from the master, the result of the task that the worker is currently computing may be unavailable from the master for an unbounded time. In this case, termination can require redundant task executions.

From a practical point of view, it is also important for a master-worker computation to complete in a timely manner. This motivates the use of a partition-aware approach. Furthermore, a practical wide-area master-worker computation, similar to *OMNI-DO*, would distribute the results among processors in several local area networks to increase the availability of the results even in the face of a long-lived network partition.

AX employs a coordinator-based approach and uses a partitionable group membership service to track changes in the group's composition. There is a set of processes, one for each processor in P, that are cooperating to solve the *OMNI-DO* problem, and there is a set of tasks T known by all of the processes that are to be computed. All tasks have the same duration.

During execution, each process i maintains the local sets D_i, R_i, U_i and G_i:

D_i – The set of tasks whose results process i knows.

R_i – The results of the tasks in D_i.

U_i – The set of tasks whose results process i does not know: $U_i = T - D_i$.

G_i – The set of processes in i's view.

For each process i, $\mathit{rank}(i, G_i)$ is the rank of i in G_i, when the process identifiers are in some well-known order, such as the order they appear in the view membership list. For a task u in U_i, $\mathit{rank}(u, U_i)$ is the rank of u in U_i, when the task identifiers are sorted in ascending order.

The task allocation rule for each processor i is:

- if $\mathit{rank}(i, G_i) \leq |U_i|$, then processor i performs task u such that $\mathit{rank}(u, U_i) = \mathit{rank}(i, G_i)$.
- if $\mathit{rank}(i, G_i) > |U_i|$, then processor i does nothing.

AX structures its computation in terms of rounds. Each process executes at most one task in each round. At the beginning of each round, each processor i knows G_i, D_i, U_i and R_i. Since all processors know G_i, each processor deterministically chooses a group coordinator, which is the process with the highest ID in G_i. In each round, each processor i reports D_i and R_i to its coordinator. The coordinator receives and collates these reports and sends the results to all the members of the group. After receiving this message from the coordinator, each process i updates D_i, R_i, and U_i, and then chooses a new task to compute using the task allocation rule.

Initially, all processes are members of a single initial view that contains all the processes. If a regrouping occurs, then the affected processes receive the new views from the group membership service, complete any tasks that they are currently computing, and report the results to the new coordinators. Each new coordinator will then start the first round in the new view.

We can classify all the tasks into three types as follows. Given a view G:

Fully done tasks $FD(G)$: the tasks $\{t \in T | \forall i \in G : t \in D_i\}$.

Partially done task $PD(G)$: the tasks $\{t \in T | \exists i, j \in G : t \in D_i \wedge t \in U_j\}$.

Undone tasks $UD(G)$: the tasks $\{t \in T | \forall i \in G : t \in U_i\}$.

Consider the situation in which the view G partitions into two or more views. Clearly, all the tasks in $FD(G)$ will not be re-executed by any process in G. In any algorithm, all of the tasks in $UD(G)$ are at risk of being executed redundantly because of the need for liveness: the network partition may last for an unbounded time, and so the tasks in $UD(G)$ may need to be computed by at least one process in each new view. Any task u in $PD(G)$ is also at risk of being executed redundantly in any new view that does not know u's result. By using a round structure, AX ensures that the size of $PD(G)$ is never larger than $|G| - 1$. No algorithm can ensure $PD(G)$ is smaller unless it does not allow all processes in G to be computing at the same time. Hence, using a round structure is one way to reduce the number of redundant tasks executed.

There are other ways to ensure $|PD(G)| < |G|$. For example, each process i could send its result of executing task u to all members of G_i rather than just sending it to the coordinator. Process i would not allocate another task until it knew that all other processes j in G had updated D_j, R_j and U_j. AX was not designed this way because it would introduce many more messages [66], and given that the tasks in T all have the same execution time, there would not be much to be gained if this additional communication were used.[1]

Even though it is optimal (in terms of the worst-case number of tasks executed, as a function of the number of views installed) and has a low message overhead, AX was not meant to be a practical algorithm. It has some obvious problems that can be easily addressed. For example, each processor sends D_i and R_i to its coordinator at the end of each round. Doing so makes the propagation of results when views merge trivial to implement, but it could result in a huge message overhead. It would be simple for i to limit the size of D_i and R_i by having each process maintain a vector denoting the results it knows. Such a vector could probably be kept small with a suitable encoding technique. These vectors could be managed in a manner similar to logical clocks [102] so that when i sends a result to its coordinator, it only sends the part of D_i and R_i that i does not know that the coordinator knows.

Another problem that could be easily addressed would reduce the number of tasks executed redundantly in some runs (recall that AX is optimal only in terms of the worst-case behavior). Consider what AX does if the processes partition into two or more views in the initial state. The processes in each view will deterministically start executing the same set of tasks. One could instead have a coordinator choose tasks randomly without replacement for the processes in its view to compute. If T is large and the partition does not last a long time (as measured in task computation time), then this would result in a smaller expected number of redundant task executions.

A third problem arises from its use of coordinators. In a real system the tasks would not have exactly the same execution time. Each round runs until the longest task in that round has completed, and with a large number of workers, the longest task in each

[1]Note that the problem can be solved with *no* message communication, so trying to include optimality in terms of message complexity is not an interesting exercise [119].

round could take significant time to complete compared to the shortest task. Despite the larger number of messages, the variation of AX we gave above that does not use a coordinator would probably be a better choice.

In the next section, we discuss a more important practical problem with AX which arises from its use of group communication. This problem occurs even if all the changes above are made to AX.

4.2 Analysis of AX

AX was designed with the idea of reducing the number of redundantly computed tasks. Redundant task execution can occur when the network partitions. The exact number of redundant tasks executed depends on many factors, including the length of time a task executes, the number of processes, how the processes partition from each other, at what point in the algorithm the partitions occur, and for how long each endures.

AX was not explicitly designed to compute the results of *OMNI-DO* quickly, but in fact it would appear to do so. It is work conserving: if there is a task to be executed and a process available to execute a task, then the task is executed. Since tasks have the same computation time, blocking due to the round structure should be small and the order that the tasks are executed is immaterial. Hence, since AX reduces the number of redundant tasks, it would appear to be a fast algorithm as well. However, as was discussed in Section 3.2.1, group communication services can perform poorly when communication is not symmetric and transitive. The actual performance of AX depends strongly on the frequency and duration of non-symmetric or non-transitive communication. Furthermore, worst-case behavior is not necessarily the only important metric to consider. In practice, the average execution time is probably of more interest to grid computation consumers.

We can estimate the slowdown factor of AX with a simple model. Assume that the network does not partition but suffers from periodic times of unstable connectivity: for α seconds it is not a clique, for β seconds it is a clique, for α seconds it is not a clique, and so on. Let each task compute for T seconds. Since the tasks have the same length, they will complete at close to the same time. The master then broadcasts the new assignment to start the next round. With probability $\alpha/(\alpha+\beta)$, the broadcast will occur during an α period, and will block on average $\alpha/2$ seconds; otherwise, the broadcast occurs during a β period and does not block. Hence, it blocks on average $\alpha^2/2(\alpha+\beta)$ seconds. The slowdown factor should then be $(T+\alpha^2/2(\alpha+\beta))/T$. This simplifies to:

$$\text{Slowdown factor} = 1 + \frac{\alpha F}{2T}$$

where F is the formula derived in Section 3.4.2 that calculates how large is the portion of the trace where communication is non-transitive or asymmetric.

4.2.1 Simulation Results

In order to establish if our observations on the impact of non-transitive and asymmetric communication are correct, we simulated AX running over the RON trace. We used the communication graph to generate a set of view changes, and we used the group membership semantics in which a process's multicast is blocked while its connected component is not a clique. Given the round nature of AX, our results would not differ if only the delivery of multicast messages were blocked during these times.

We chose each of the sixteen nodes monitored by RON to have ten processes. Since processes in one node are in the same local area network, we decided that the communication among them is always symmetric and transitive. We chose there to be 1,000 tasks that run for a time much longer than it takes for messages to be transmitted over the wide area network. We decided that communication are effectively instantaneous among a component when its communication graph is a clique (otherwise, it is blocked as described above).

We first ran the simulations over a period of time in the RON traces during which there were no node crashes and there were no partitions. A run $Sim(i)$ indicates a simulation in which each task runs for $100i$ seconds. The results of our simulation are shown in Table 4.1. The column "Start time" indicates where in the trace the simulation was started. The row "Slowdown" gives the slowdown factor for different simulations and the row "F" gives the value of F for the segment of the run during which the simulation ran.

F was either very small or very large. When F is small, there is no effect on the running time, and large values of F can make AX run between about 50% to over 11 times slower. As predicted, the slowdown factor decreases with increasing T. The slowdown factor equation requires a value for α, and so we computed a histogram of values of α and β over the entire run (when the network was partitioned we considered only the larger connected component). Table 4.2 gives the histogram. The average and median values for α are 143 and 22 seconds, and for β are 665 seconds and 162 seconds. The ranges of values for both are very large. If we use the average value of α in the slowdown formula, then the slowdown factors in Table 4.1 are usually much higher than the slowdown formula predicts. This deviation is caused by some larger periods of non-transitive communication during the simulation.

We then ran $Sim(1)$ over five different time intervals in the RON trace during which F was close to 50%. The results are shown in Table 4.3. With $\alpha = 143$ seconds, the slowdown formula predicts a slowdown factor of 1.36, which is close to the values computed by simulation.

Since no partitions occurred during any of these runs, the number of redundantly executed tasks should be zero, which is what the simulation reported. We then ran $Sim(1)$ starting at five different times, where at each starting time there was a partition but no crashed nodes. The results are shown in Table 4.4. In the first simulation, the network was partitioned throughout the whole execution, and so all tasks were executed redundantly. In the other cases, the partition lasted for only a short time. Here, since the partition result in one side having a single node with ten workers, and redundant executions starts when there is a partition, there will be only ten redundantly executed

Start time	Value	$Sim(1)$	$Sim(2)$	$Sim(3)$	$Sim(4)$	$Sim(5)$
500,000	Execution time	700	1,400	2,100	2,800	3,500
	Slowdown	1	1	1	1	1
	F	0%	0%	2.7%	2.1%	1.6%
510,000	Execution time	700	1,400	2,100	2,800	3,500
	Slowdown	1	1	1	1	1
	F	0%	0%	0%	0%	0%
520,000	Execution time	700	1,400	2,100	2,800	3,500
	Slowdown	1	1	1	1	1
	F	1%	0.5%	0.9%	0.6%	0.5%
530,000	Execution time	3,266	6,320	6,320	6,858	8,156
	Slowdown	4.67	4.51	3.01	2.45	2.33
	F	99.2%	99.2%	99.2%	99.3%	99.4%
540,000	Execution time	2,696	3,780	4,081	5,211	5,211
	Slowdown	3.85	2.70	1.94	1.86	1.49
	F	98.6%	98.9%	98.7%	98.6%	98.6%
550,000	Execution time	7,819	9,573	10,188	10,188	10,825
	Slowdown	11.17	6.84	4.85	3.64	3.09
	F	99.4%	99.2%	99.1%	99.1%	99.2%
Average	Execution time	2,647	3,979	4,482	5,110	5,782
	Slowdown	3.78	2.84	2.13	1.82	1.65

Table 4.1: AX simulation results for traces *with no* partitions. Time values are given in seconds.

Time range	Non-transitive views		Transitive views	
	Number	Percentage	Number	Percentage
10	425	28.41%	240	16.05%
100	1,271	84.96%	637	42.61%
1,000	1,460	97.59%	1,227	82.07%
10,000	1,494	99.87%	1,489	99.60%
100,000	1,496	100%	1,495	100%

Table 4.2: The length of views. Time values are given in seconds.

Start time	526,900	688,000	856,500	1,070,000	1,189,300	Average
Execution time	1,118	813	1,498	1,012	976	1,083
Slowdown	1.6	1.16	2.14	1.45	1.39	1.55
F	52%	41%	55%	44%	50%	48.4%

Table 4.3: Simulation with $\alpha = 0.5$. Time values are given in seconds.

	Partn 1	Partn 2	Partn 3	Partn 4	Partn 5
Start time	652,700	905,365	922,027	1,071,235	1,140,363
Duration	27,700	71	28	98	92
Execution time	10,000	19,104	2,642	3,852	761
Slowdown	14.3	27.3	3.8	5.5	1.1
Redundant tasks	1,000	10	0	10	10

Table 4.4: AX simulation results for traces *with* partitions. Time values are given in seconds.

tasks. In the third simulation, the number of redundant tasks is zero because the larger component is not a clique during the partition, and so only the smaller component makes progress.

4.3 Summary

Algorithm AX depends on group communication to maintain which processes are available during the master-worker computation. The previous chapter showed that when network communication is not symmetric or transitive, group communication blocks. This chapter investigated the impact this blocking has on the performance of algorithm AX. We found that the impact was significant. When running AX on top of the traces, blocking caused the computation to slow down from 50% up to 11 times. The results give a strong indication that there are better approaches to solving the *OMNI-DO* problem in wide area networks than algorithm AX.

Chapter 5

Master-Worker Computations without Group Communication

The previous chapter showed that algorithm AX, which is based on wide-area group membership, blocks during times of unstable network connectivity. In this chapter, we devise a simple protocol called **W**ide-**A**rea **M**aster-**W**orker (WAMW), that we conjecture is more appropriate for the network connectivity we observed in the previous chapter. While WAMW is a partition-aware solution, it does not require group communication. We evaluate the properties of WAMW using the same traces as we did for algorithm AX.

5.1 Protocol Requirements

The previous chapter showed that although algorithm AX is optimal in terms of the worst-case number of redundant tasks, maintaining this property causes blocking when communication fails. Thus, the measure of success of an alternative algorithm is both the number of redundant task executions and the amount of blocking. We also require that an alternative algorithm maintains the safety and liveness properties of *OMNI-DO*. The following sections elaborate on the key protocol requirements, *avoiding blocking*, *avoiding redundant tasks*, and *satisfying OMNI-DO*.

5.1.1 Avoiding Blocking

In Section 4, our experiments showed that algorithm AX blocks during times when the network is not transitive and symmetric. Ideally, a partition-aware master-worker protocol should make progress even when the network is unstable. Flooding protocols and gossip protocols [26, 112] are often used to meet that requirement.

The performance of flooding and gossip protocols depends on the connectivity between the nodes in the communication graph. Let $d(a, b)$ be the shortest distance between nodes a and b in a graph G. The time it takes to complete a multicast in the communication graph using flooding depends on $D(G) = \max_{a,b \in G} d(a, b)$. We call $D(G)$ the *maximum distance* of G.

Trace	Time $D(G) > 1$	Time $D(G) = 2$	Percentage
RON	215,043.4	214,376.8	99.7%
exp1	145,560	138,360	95%
exp2	235,20	22,440	95%
exp3	213,180	208,260	98%

Table 5.1: Maximum distance, where the time values in the first and second column are given in seconds.

Table 5.1 gives information on $D(G)$ where G is the communication graph when not partitioned and the larger connected component when partitioned. The first column gives the amount of time that G is not a clique, and the second column gives the amount of time that $D(G) = 2$. The third column is the percentage of time that the graph is not a clique and $D(G) = 2$. These values are all close to 1, and so a flooding protocol should be fast most of the time. The observation that $D(G)$ is rarely more than two has been noted by many others, for instance in [4].

5.1.2 Avoiding Redundant Tasks

Partitions cause redundant tasks to be executed in AX, something that cannot be fundamentally prevented unless we allow blocking for an unbounded amount of time. Algorithm AX does not execute redundant tasks when communication is asymmetric and non-transitive because the group communication system blocks. However, the disadvantage is that the probability of experiencing a network partition increases with the time the computation takes to complete. Thus, in practice, the approach AX takes to minimize redundant tasks may actually increase the number of redundant tasks instead of decreasing them.

Avoiding redundant tasks requires global knowledge of which tasks have been completed. An approach to reduce the effects communication problems have on synchronizing global state, is to statically set up groups of processors within each local area network, or subnet, and to have these groups work in isolation as long as possible. Each group periodically flood its view of the global state to the rest of the groups. Since the maximum distance is seldom more than 2, global state updates should be disseminated between groups fast.

Given small amounts of contention on the global state of the computation, an attractive abstraction for synchronizing access and detecting failures is leases [68].

5.1.3 Satisfying *OMNI-DO*

When there is a partition that splits the computation in two cliques, AX computes the same set of remaining tasks in both partitions in order to satisfy the *OMNI-DO* safety property which states that the computation terminates when all masters know the results of all tasks.

Given the approach stated in the previous section, where groups are statically set up within each subnet and the work split among the groups, we need to ensure that if one of the groups partitions away, the *OMNI-DO* safety property is still enforced. An approach to ensure this is to associate a lease with each group, and if the lease expires start redoing the work allocated to the expired group in another group.

5.1.4 Summary

To summarize, our alternative algorithm uses the following strategies to ensure that requirements are met:

- Flooding instead of group communication.
- Split work in disjoint sets to reduce synchronization in global state.
- Leases to synchronize access to global state and detect failures.
- Recompute the sets of other groups when leases expire.

5.2 WAMW

This section presents WAMW, our alternative to algorithm AX.

5.2.1 System Model

We assume a distributed system comprised of computers, or processors, connected to local area networks, where the local area networks are connected into a wide area network. Processes on different processors can only communicate by sending messages over the network.

We assume that processors can fail by crashing and that the network communication can fail by dropping messages. The use of leases requires the *timed asynchronous* system model [40], since leases assume that all participating processors have the same clock rate. Thus, we assume that processors have clocks that progress at a rate close to real time, but the time between some process p sending a message and the intended destination p' delivering the message can be arbitrarily long.

We assume that network communication provides FIFO delivery order: if process p sends message m to process p' and then sends message m' to process p', then p' may receive just m, or just m', or m before m', but never m' before m.

5.2.2 Algorithm Structure

WAMW uses masters and workers to perform a computation. Masters schedule tasks, and workers request and execute tasks given by the masters. Initially, each subnet has at least one master and its set of workers as depicted in Figure 5.1, where there are three master processors (M) and their associated worker processors (W). Arrows denote

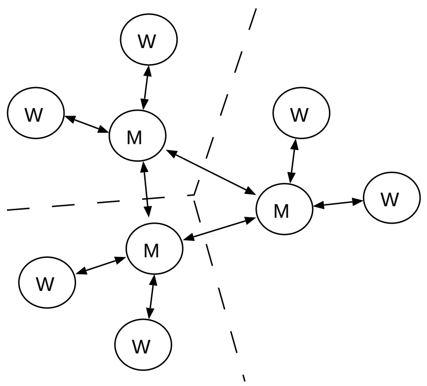


Figure 5.1: WAMW spanning three subnets.

possible directed communication paths among master and worker processes. The general approach of WAMW is to split the computation into disjoint sets of *tasks*. A set of tasks is performed by a deterministically chosen master process and its workers. When all sets of tasks have been performed, the computation terminates. The next subsections will elaborate on the various parts of the algorithm.

Pseudocode 5.2.1 Worker main loop.

```
def worker:
  result = NULL
  while true:
    task = request_work_from_master(result)
    if task == NO_MASTER:
      terminate()
    lease_timeout = extract_lease_timeout(task)
    fork(task)
    task_completed = false
    while not task_completed:
      result = block_until(expired(lease_timeout) or task_completed)
      if result == LEASE_EXPIRED:
        lease_timeout = renew_lease(task)
      else:
        task_completed = true
```

5.2.3 Workers

The main control flow of the worker is given in Pseudocode 5.2.1. A worker processor executes a single task at a time. When a worker starts up, it requests a task from its designated master by sending a TASK_REQUEST message. Upon receiving a task, the

Pseudocode 5.2.2 Worker to master request handler.

```
def request_work_from_master($result$):
  $masterid$ = get_master()
  while true:
    $task$ = task_request($masterid$, $result$)
    if $task$ == REQUEST_FAILED:
      mark_as_unavailable($master$)
      $masterid$ = get_master()
      if not $masterid$:
        return NO_MASTER
    else:
      return $task$
```

worker starts executing it and sends the result of the task to the master upon completion. When requesting a task from a master, the request can also fail (see Pseudocode 5.2.2) if the master has crashed or, for masters in other subnets, has partitioned away. When a request fails, the worker marks the current master as unavailable, chooses a new master, and re-issues the request to this master. If this master fails, a new is chosen, and so on until there are no masters that the worker can choose. A worker terminates when all masters have terminated, meaning the computation has terminated (see handling of REQUEST_FAILED in Pseudocode 5.2.2).

5.2.4 Masters

The main control flow of a master is given in Pseudocode 5.2.3. A master first checks if there is a worker request pending. If so, the master extracts the result from the previous task done by the worker from the request, and marks the task as done. The worker is subsequently added to a list of idle workers. The master then checks if there are any unallocated tasks left to be performed. If there are tasks to be performed, the master allocates a task, picks an idle worker, and sends the task to the worker. The masters are terminated after every master knows the results of all tasks.

5.2.5 Leases

To detect when a master or worker has crashed, WAMW uses two types of leases. The first lease is the *worker lease*. When a worker is given a task by a master, the master registers a lease on the task. The value of the worker lease is the estimated completion time of the task. From a practical point a view it is hard for a master process to predict the time it takes to execute a given task, since worker processors may have varying CPU resources. However, if a worker finds that it cannot complete its task within the estimated completion time, it calculates a new lease and sends a RENEW_LEASE message to the master (see Pseudocode 5.2.1). If, however, a master finds that a worker lease has expired, the worker is assumed to have crashed.

Pseudocode 5.2.3 Master main loop.

```
def master:
    pmax_interval = interval between each master state update
    pmax = current_time() + pmax_interval
    idle_workers = []
    while true:
        state_update = false
        type, workerid, result = get_worker_request()
        if type == RENEW_LEASE:
          set_worker_lease(workerid)
        if type == TASK_REQUEST:
          idle_workers.append(workerid)
          if result != NULL:
             state_update = true
             mark_task_as_done(result)
        while len(idle_workers) > 0 and num_unallocated_tasks() > 0:
          task = allocate_task()
          workerid = idle_workers.pop()
          set_worker_lease(workerid)
          send_task_to_worker(task, workerid)
          state_update =true
        update_master_state(state_update)
        check_leases()
        if all_tasks_completed():
          terminate()
```

Pseudocode 5.2.4 State updates from master to the other masters.

```
def update_master_state(state_update):
    if current_time() > pmax or state_update:
       pmax = current_time() + pmax_interval
       send_local_state_to_masters()
    state = receive_state_from_masters()
    if state:
       update_local_state(state)
```

Pseudocode 5.2.5 Checking for expired leases in the master.

```
def check_leases:
  for lease in master_lease:
    if expired(lease):
      unallocate_all_tasks_allocated_by_master()
  for lease in worker_lease:
    if expired(lease):
      unallocate_task_allocated_by_worker()
```

The second type of lease is the *master lease.* Every master registers a master lease with every other master. If a master lease expires, then the master for which the lease expired is assumed to have crashed or partitioned away.

Leases require clocks to progress at a rate close to real time. If not, a master with a faster clock rate than another master may erroneously expire leases and start executing redundant tasks without any communication failures having occurred. Similar, a worker with a slower clock rate than its master may fail to renew its worker lease and cause the task to be redundantly executed by another worker.

5.2.6 Task Allocation

Let T be a vector of all tasks in the computation. The index operation for T gives the task associated with the index (e.g., $T[i]$ returns task i). As in AX, task allocation in WAMW assumes that all masters know all tasks T to be executed in advance. WAMW also assumes that initially all masters know the identity of all other masters and that masters are totally ordered by $0, 1, \ldots, (n-1)$ where n is the number of masters. Before the computation starts, the complete task list for the computation is divided evenly in a deterministic way among all masters. The size of the slice for each master is $\frac{|T|}{n}$. The slice of tasks to perform for master i, S_i, is then given by:

$$S_i = T[i \times size, (i+1) \times size]$$

and the *remote set* R_i for master i is defined by:

$$R_i = \{S_0, S_1, \ldots, S_{i-1}, S_{i+1}, \ldots, S_{n-1}\}$$

Tasks are allocated by master i to idle workers as follows: if there is an unallocated task in S_i, then mark the first such task as allocated; else mark the first unallocated task in R_i as allocated. When a worker lease held by master i expires, all tasks marked allocated with the expired lease are marked unallocated in S_i or R_i. When a master lease for master j held by master i expires, master i marks all tasks in S_j as unallocated (see Pseudocode 5.2.5 for lease expire logic). When a task completes, it is removed from S_i or R_i.

A task allocated from S_i by master i only causes redundant task executions when a worker lease expires. A task allocated from R_i by master i always causes a redundant task execution, unless a remote master j has crashed before allocating this task in S_j. A task $u \in S_j$ in R_i will only be allocated after the master lease of master j has expired.

5.2.7 State Dissemination

At each task completion (and the corresponding task allocation) and at the frequency *pmax*, a master broadcasts its state. This state includes the tasks that have been completed, the tasks that have been allocated, the task results, and master lease information to the other masters (see Pseudocode 5.2.4). Broadcasting at task completion reduces the chance of two masters both allocating the same tasks in their remote sets. Broadcasting at *pmax* frequency increases the chance of overcoming temporary communication failures. The value of *pmax* is a function of the average task execution time and the value for the master lease. For instance, if a task takes c seconds to complete on average and given that $D(G)$ is almost always 2 or less, then *pmax* should be set to $\frac{c}{2}$ to ensure that state about individual task completions is broadcast at least twice. However, for a master lease m, if $\frac{c}{2} > m$, *pmax* should be set to a smaller value than $\frac{c}{2}$ to avoid a master leases expiring first.

5.3 Worst-case Analysis of WAMW

Redundant task execution occurs in WAMW when leases expire. Compared to AX, the number of redundant task execution can be large. Let:

M be the number of masters.

K_i be the number of workers associated with master i; $K_i > 0$.

K be the number of workers: $K = \sum_{i=1}^{M} K_i$.

N be the number of tasks.

W be the total number of tasks executed.

Since communication is asynchronous, we can delay messages for an arbitrary amount of time without having a partition (assuming that we define a "partition" to mean communications are broken for an even longer time, perhaps forever). In this model, W can be as large as NK: Delay all messages between masters so that the master leases expire. Thus, each master will compute all N tasks. Consider master i. Since it is work-conserving, it will assign K_i tasks in each round. Number the workers as w_1 through w_x where $x = K_i$. We can delay the replies long enough so that the worker leases expire. The master will not assign a task until a worker responds with results of a task; let this worker be w_1. Thus, the task that w_1 computed will not be redone. We can reassign all the other previously assigned tasks, for example, by assigning w_2's task to w_1, w_3's tasks to w_2, and so on.

The total number of tasks that master i computes is the number of tasks w_1 computes plus the number of tasks w_2 computes ... plus the number of tasks w_x computes, which gives a worst case of:

$$N + (N-1) + (N-2) + \ldots + (N - K_i + 1) = K_i \frac{(2N - K_i + 1)}{2}$$

The total number of tasks W_{wc} is thus:

$$W_{wc} = \sum_{i=1} K_i \frac{(2N-K_i+1)}{2} = NK + \frac{K}{2} - \sum_{i=1}^{M} \frac{K_i^2}{2}$$

From this it follows that W_{wc} is maximized when $\forall i : K_i = 1$ in which case $W_{wc} = NK$.

Suppose we adopt a stronger model: Communications can be arbitrarily delayed only if there is a partition, and partitions are detected by timeouts of master leases. For example, if a set of masters c_1, c_2, c_3, c_4 send messages to each other and then partition into two components c_1, c_2 and c_3, c_4, then by the time the partition is detected c_1 and c_2 have received each other's messages (but not necessarily c_3 and c_4's messages). We also assume that workers communicate with their masters in a timely manner and so no worker leases expire.

Under this model, W still depends on more than the number of installed views. Consider the simple case of the M masters partitioning into two: One side, A, consists of M_1 masters and the other side, B, consists of M_2 masters. Assume that each master has the same number of workers. Have the masters each complete their local sets, but delay all messages they send among themselves. After the partition, each master in A will redo the tasks that were computed in B and vice versa. So, before the partition the A side computed NM_1/M tasks and the B side computed NM_2/M tasks. Afterwards, the A side computes M_1NM_2/M tasks and the B side computes M_2NM_1/M tasks. Summing these, we get the total number of task computations, and so the work is:

$$W_{sc} = N + (2NM_1M_2)/M$$

W_{sc} is maximized when $M_1 = M_2 = M/2$, giving in the worst case $W_{sc} = (1 + M/2)N$. Even with only two views installed, the worst-case of W_{sc} scales with the number of masters and is thus not optimal.

We expect that communication will not be as poorly behaved as it is needed to attain these bounds. Assuming that we can assign lease values in an appropriate manner, we expect that the number of redundantly executed tasks can be kept small in practice.

5.4 Assigning Leases

The length of master leases influences both the number of redundant tasks executed and the total execution time of all tasks. Assume that all masters work in isolation during the whole computation, causing W_{wc} tasks to be executed. For the total execution time of the computation to be optimal, the master lease should not be larger than the time t it takes to execute the local set S_i for a master i, since the master immediately starts executing tasks in R_i once S_i is completed. Hence, t seems like a reasonable upper bound for the master lease. For long-lasting partitions, though, t is unrealistically large to use for master leases, since doing so will cause the total execution time to become very large. The question is then, how small can the master lease be to make the number of redundant tasks small, while at the same time avoiding blocking the computation unnecessarily.

One obvious value for a master lease is the expected duration of a partition. In general, such a value is impossible to establish, but we can use a value based on our traces. Partition durations in the RON trace were less than 2 minutes for 9 out of 11 partitions. For the exp1 trace, one partition lasted 5 minutes and one for over an hour. Hence, if we disregard the very long partitions, partition in the traces typically lasted 5 minutes or less. We thus set the master lease to 10 minutes for our experiments to mask the majority of partitions while ensuring that the computation does not block for long periods during long-lasting partitions.

The value for the worker lease is based on the expected execution time of a given task. We assume that for most cases when a worker sends a RENEW_LEASE message to the master, the message gets to the master successfully and in a timely manner, since all communication between workers and masters is within subnets with small latency and low packet loss. Thus the accuracy of the worker lease is not important, as workers can periodically renew the lease if it is an underestimate. In our simulations we assume that workers do not crash, and hence the worker lease has no significance for our results.

5.5 Simulation

We used the RON traces to determine whether a message could be successfully sent from one computer (i.e., RON node) to another during an unreliable broadcast. If a message sent from node i to node j in the RON trace fails, then we assume that all subsequent messages sent in WAMW from node i to node j fail until a message is successfully sent from node i to node j in the trace.

Since the task execution times are identical, we added a small random jitter in the interval $[0, 2]$ to the execution time of the tasks to avoid unrealistic synchronous behavior in master to master communication. Adding jitter causes a master state broadcast for all task completions (and corresponding allocations). Since network jitter was not part of the simulation for AX, we normalized the execution time for WAMW to exclude the additional time caused by the jitter.

To increase tolerance to communication failures when doing the state broadcasts in the master, additional broadcasts are sent every $pmax$ seconds. In our experiments, tasks have lengths $100i$ seconds where $i = 1, \ldots, 5$ and to ensure we have at least one additional broadcast between every task completion broadcast, we set $pmax$ to $\frac{100}{2} = 50$. Note that we do not increase $pmax$ for $i > 1$ to avoid master leases from expiring due to temporal communication failures. As with the AX simulation we assigned one master and ten workers on each of the 16 nodes monitored by RON, and specified 1,000 tasks to be executed totally. The computation terminates when all masters know the results of all tasks. As with AX (see Table 4.1) we first ran the simulations over the same periods of time in the RON traces that we used for AX and reported in Table 4.1. The results are shown in Table 5.2.

We also report the number of messages that WAMW uses. The values are significantly larger than the number used by AX. The average transmission rate is larger for shorter tasks. With $Sim(1)$ the rate is about 26 messages/second and with $Sim(5)$ it is about 9 messages/second. If the message size is large, then the overhead caused by the messages may pose a scalability issue when task duration is short.

The number of redundantly executed tasks is always 0, as expected given that there

		$Sim(1)$	$Sim(2)$	$Sim(3)$	$Sim(4)$	$Sim(5)$
Fastest running time		700	1,400	2,100	2,800	3,500
Start	Slowdown	1	1	1	1	1
at	Total msgs	18,360	21,720	25,080	28,440	31,800
500,000	Pct msgs failed	0.02%	0.02%	0.06%	0.08%	0.11%
Start	Slowdown	1	1	1	1	1
at	Total msgs	18,360	21,720	25,080	28,440	31,800
510,000	Pct msgs failed	0.08%	0.10%	0.19%	0.23%	0.28%
Start	Slowdown	1	1	1	1	1
at	Total msgs	18,360	21,720	25,080	28,440	31,800
520,000	Pct msgs failed	0.20%	0.16%	0.19%	0.21%	0.24%
Start	Slowdown	1	1	1	1	1.01
at	Total msgs	18,360	21,720	25,080	28,440	31,830
530,000	Pct msgs failed	7.40%	6.07%	6.38%	6.15%	6.02%
Start	Slowdown	1	1	1	1.02	1
at	Total msgs	18,360	21,720	25,080	28,485	31,800
540,000	Pct msgs failed	6.14%	6.05%	6.36%	6.03%	5.69%
Start	Slowdown	1.07	1.04	1.02	1.02	1
at	Total msgs	18,405	21,765	25,125	28,485	31,800
550,000	Pct msgs failed	11.82%	11.72%	11.79%	11.95%	11.82%

Table 5.2: WAMW simulation results from traces *with no* partitions. Time values are given in seconds.

	Partn 1	Partn 2	Partn 3	Partn 4	Partn 5
Start time	652,700	905,365	922,027	1,071,235	1,140,363
Duration	27,700	71	28	98	92
Execution time	10,000	700	700	700	700
Slowdown	14.3	1	1	1	1
Redundant tasks	1,000	0	0	0	0
Total msgs	78,480	18,360	18,360	18,360	18,360
Pct msgs failed	28.19%	3.56%	2.64%	4.85%	1.38%

Table 5.3: WAMW simulation results from traces *with* partitions. Time values are given in seconds.

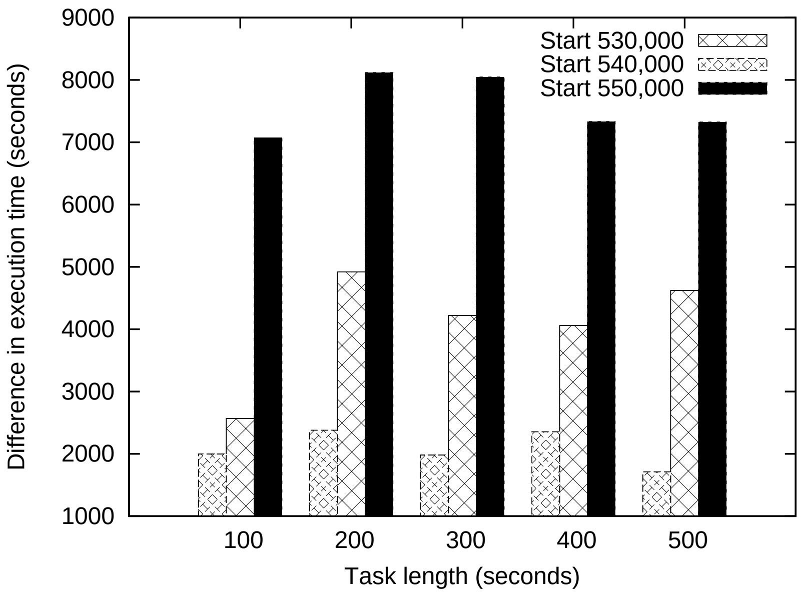


Figure 5.2: Difference between AX and WAMW execution times when there are no partitions in the trace. The height of the bars depict the amount of time AX uses in excess of WAMW for the same computation.

were no partitions, and hence the probability of a lease expiring is low. 80% of the slowdown factors are 1, and the maximum value is 1.07 which is much better than the performance of AX, where the *largest* slowdown was 11.17 and the *smallest* slowdown larger than 1 was 1.49. For the results where the slowdown is larger than 1, the execution time is always optimal plus *pmax*, where *pmax* is 50 seconds. In these cases, the last task completion broadcast from a controller failed to reach all other controllers, but the successive broadcast which happens *pmax* seconds later completed the broadcast. Figure 5.2 shows the difference in task execution time between AX and WAMW for this simulation. Since the difference is 0 for start times 500000, 510000 and 520000, these were omitted from the figure. The figure clearly indicates that the computation completes much faster with WAMW than with AX.

We then ran $Sim(1)$ over the five different time intervals in Table 4.4. The results are shown in Table 5.3. Since the network is partitioned during the whole execution in the first simulation, all the tasks are executed redundantly, as they are with AX. For the other simulations, no tasks are executed redundantly, since the duration of the partitions are less than the master lease. The slowdown factor of the first simulation is 14.3, and that of all the other simulations are 1, all of them optimal. Figure 5.3 shows the execution time of WAMW compared to AX in this case.

To summarize, in most cases WAMW runs faster than AX, since it does not block during non-transitive communication. And in practice, the number of redundant tasks executed by WAMW is not larger than AX. Attaining this performance requires using good values for the lease times. In practice, a user might set the master lease times to

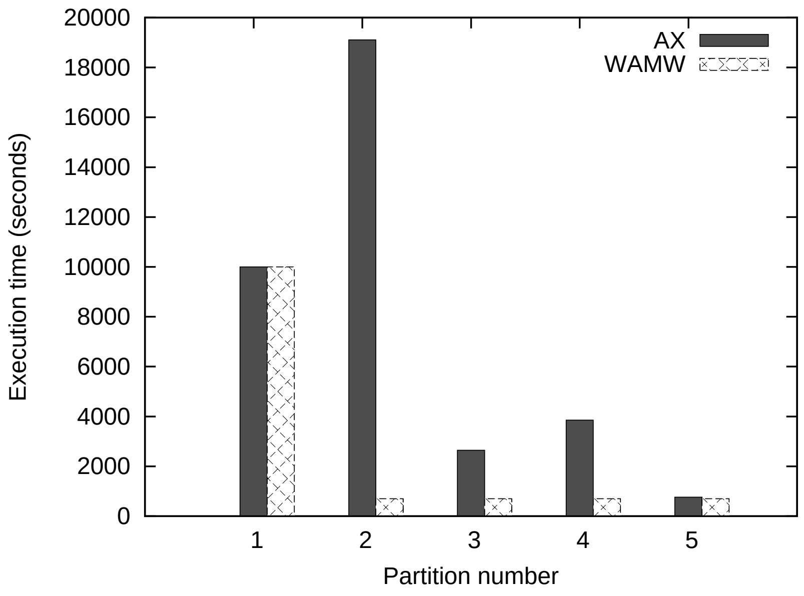


Figure 5.3: Comparison of WAMW and AX execution times when there are partitions in the trace.

balance off his or her own particular trade off between running time and redundant task execution.

5.6 Observations

The motivation for this study was to gain insight in the issues of constructing a wide-area master-worker framework for computational grids. We chose to investigate a protocol that had been developed to minimize the amount of redundant tasks executed and evaluated it both with an analytical model and via simulation. The protocol was not meant to be a practical one, but on the surface it does not appear hard to make it more practical. We also compared the performance under simulation against a simple protocol that does not use group communication and that has a significantly higher upper bound on the amount of redundant work.

Our investigations reveal that the frequency of non-transitive or asymmetric communication can be significant. In an earlier study [95], results indicated significantly fewer periods of poor communication connectivity. However, the network considered in this study contained only five nodes. From the traces we used to evaluate AX, it appears that as the number of nodes grow, the more likely it will be that one will encounter periods of non-transitive or asymmetric communication. Hence, using wide-area group communication systems upon which to build wide-area master-worker does not seem to be a good idea.

There are two ways to reduce the impact of poor communication connectivity and reduce the amount of blocking due to non-transitive and asymmetric communication, i) static link configurations, and ii) overlay networks.

Static Link Configuration. The approach here is to identify off-line which communication links will prove to be unreliable and then configure the topology of the group communication system to avoid these links. An example of doing this is shown in [95]. The drawbacks of this approach are i) it isn't clear how accurately and completely unreliable links can be identified in advance, although research has shown that routing among popular links is reasonably stable [101], and ii) the graph may have small node and link cut-sets, thereby increasing the chances of a partition. For example, in [95] to avoid a relatively unreliable link, the topology is set up as a tree. The failure of one node subsequently caused a long-lasting partition.

In [184], a tool is presented that filters BGP-4 messages into a smaller number of alerts describing where major routing disruptions occur. Such alerts are then investigated by operators in root-cause analysis of routing changes, and could also be used when setting up the topology of the group communication system to avoid communication across links which are known to be problematic. Furthermore, if parts of the network are known to be problematic in advance, AS-level path inference can be used to identify the paths which are likely to be used when two nodes are communicating. A study [122] reports that the accuracy of AS-level path inference can be as good as 70%-88%.

Overlay Networks. Blocking due to lack of transitivity or asymmetry of communication can be avoided by relaying information across stable links. This is what overlay networks like RON do [4] and the Phoenix group communication protocol do [120]. However, scalability of such overlay networks is not clear and is an ongoing research problem. If scalability can be addressed, this would be a possible approach. However, our initial results with the simple WAMW algorithm indicate that the kind of poor connectivity that exists is masked quite well by limited flooding and presumably also by gossip protocols. Hence, it isn't clear that the generality of an overlay network is required. If one had other reasons for wanting partitionable group communication, then it would be worth considering using very limited gossip or flooding underneath.

What group communication offers for wide-area master-worker computation is a mechanism to ensure that in the worst case, the number of redundantly executed tasks is small. Our initial experience with the simple WAMW algorithm, though, indicates that the expected number of redundantly executed tasks can be kept quite small without using group communication. If the worst-case scenario were more dire—for example, as it is in the Bancomat problem [173], then it would be worthwhile running a group communication system over a communication layer that relayed information to avoid poor links. For wide-area master-worker, it is hard to justify such an expenditure of effort to avoid a highly unlikely worst-case behavior.

There are, however, problem domains in wide-area master-worker where group communication provides an attractive solution, for instance, replicating masters to a small degree within each local area network. This would be done both to balance load and to mask crash failures of masters. Since communication is more reliable in a local area network, using a primary-backup approach within a local area network should work well.

5.7 Mobile Agents and the Grid

We have investigated two different solutions to master-worker computations on the grid. However, with either solution, a key problem with grid computations is how to *manage* the grid resources [96, 97]: grid computers may come and go, parts of the grid network topology may become unavailable for extended periods, and the available resources on a grid computer may vary over time. With this in mind, there are several properties of mobile agents that can be leveraged, for instance:

- Searching the grid for a particular type of resource during the computation or before the computation starts [97, 178].
- Monitoring the progress of grid applications [78].
- Moving the grid computation executing at one computer to another computer when there are lack of local resources [61].
- Upgrading and installing software required by the grid infrastructure or the grid computation itself [178].

A major challenge that faces the adoption of mobile agents in grid environments, is how to make mobile agent computations robust enough to sustain the life-cycle of potentially long-running grid applications [57]. Thus, in the next chapter, we devise a mobile agent fault-tolerance protocol that is appropriate for grid environments.

5.8 Summary

We started this chapter by establishing that a protocol based on flooding or limited gossip is likely to be suitable for the kind of network behavior we have experienced when running the experiments on AX in the previous chapter. A protocol, called WAMW, is specified and is shown to i) complete faster than AX in most cases, and ii) does not compute more redundant tasks than AX. WAMW uses more messages than AX even for failure free runs, and approaches to lower the number of messages are presented.

The main results of this study can thus be summarized:

- The amount non-transitive and asymmetric communication can be significant and cause wide-area group communication to block. Blocking may impact the application which is using group communication and impair performance.
- Simple, topology-aware solutions to master-worker computations can perform well in practice despite not being optimal in terms of the number of redundant task executions.

Chapter 6

NAP - Norwegian Army Protocol

In this chapter we devise a protocol for mobile agent fault-tolerance called the **N**orwegian **A**rmy **P**rotocol (NAP). We start by specifying the operations and system model for mobile agent computations in Section 6.1. We then specify the derived requirements for NAP in Section 6.2. Following this, we specify the basic operations of a NAP computation in Section 6.3. Based on these requirements and operations, we infer the formal properties that the NAP protocol must satisfy, and then derive a specification of the NAP protocol in Section 6.4. We then describe how to implement the derived protocol in Section 6.5, and present an implementation of NAP using the TOS platform in Section 6.6. We end this chapter by discussing NAP and network failures in Section 6.7.

6.1 Mobile Agent Computations

This section specifies the environment and the operations that our mobile agent computations must support. Following the specification in Section 2.1, a computer, or processor, hosting a mobile agent system constitutes a *place*. In practice, a processor can host more than one place, but without loss of generality we assume that each processor hosts a single place. All places execute identical deployments of the mobile agent system. To start executing a mobile agent *stage* at some processor H, the place at H is provided with the code and state necessary to execute the stage. This implies that executing a stage leaves no residual dependencies, as we discussed in Section 2.1.1.

We specify three operations for mobile agent computations: **init**, **move**, and **terminate**. The **init** operation is used to create a new agent and start its first stage. That stage may be executed at a different place than the place invoking **init**. Hence, if a mobile agent computation is started within an application, the **init** method is normally used to start the first stage of the computation. The **move** operation is used to migrate the agent from one place to another, and takes the stage code to execute and the place to execute the stage on as parameters. A mobile agent terminates its execution using a **terminate** operation.

More formally, the execution of a mobile agent is a sequence of stages, where each stage is terminated by a **move**, or a **terminate** operation. The **init** operation does not

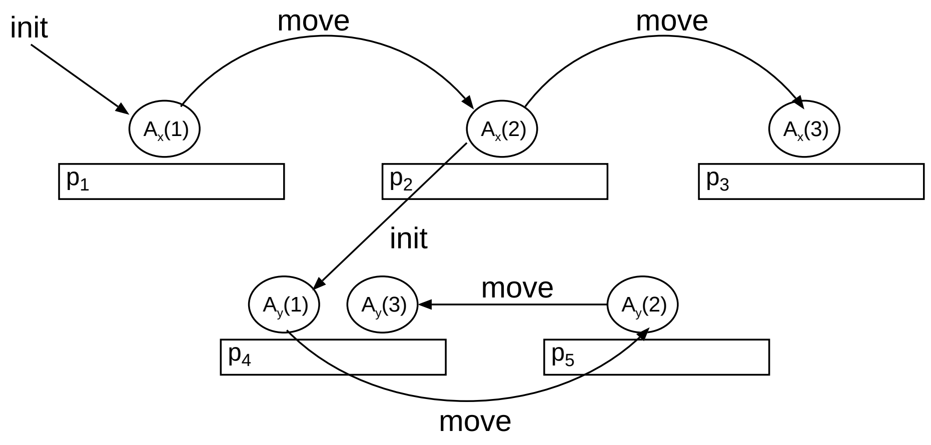


Figure 6.1: Stages of mobile agents x and y.

terminate a stage since the mobile agent is not necessarily executing a stage when **init** is invoked. For a mobile agent named m, we denote the ith stage of this agent as $A_m(i)$.

Figure 6.1 shows a mobile agent computation with an agent called x, originating at p_1 with $A_x(1)$ after **init** has been invoked. The second stage of the agent, $A_x(2)$, starts when $A_x(1)$ executes **move** to place p_2 and terminates when $A_x(2)$ executes **move** to place p_3. During execution of $A_x(2)$, the agent invokes the **init** operation that creates a new agent y on place p_4. The new agent y executes a **move** to place p_5 in its second stage and then a **move** to place p_4 again in its third stage. When only one agent is used, the subscript indicating the agent name is omitted from the presentation.

6.1.1 System Model

A mobile agent computation is susceptible to *crash* and to *network* failures. Consequently, we make the following assumptions:

Crash Failures. A place can crash, and this causes all mobile agents executing on that place to crash. A processor itself can crash, resulting in the place and all other programs running on that processor crashing. Finally, the mobile agent itself can crash without the processor or place crashing. A crash causes the mobile agent to halt execution.

To tolerate crashes, we assume that associated with each processor p is some small, well-defined set of places that can detect the crash of p, the place at p, and any mobile agents being executed at p. Postulating this set of places is a practical way of stating the *fail-stop* failure model [157].

Reliable Communication. We assume that each pair of processors is connected by a FIFO communication link: if process p sends message m to process p' and then sends message m' to process p', then p' receives m before m'. Further, we assume that transient communication failures between non-crashed processors are masked by a lower level transport protocol, similar to the semantics offered by TCP communication.

6.2 Derived Requirements

This section specifies the derived requirements for our mobile agent fault-tolerance protocol, and is based on the investigation of related work in Chapter 2. The two requirements are that the protocol must be *non-blocking* and have *efficient resource usage.*

Non-blocking. In Section 2.4.1 we observed that enforcing the exactly-once property can cause a mobile agent computation to block for an unbounded amount of time given certain network communication failures. From this, we determined that the enforcement of the exactly-once property of a mobile agent stage is an efficiency concern and not a safety concern. Hence, for efficiency, we require that all stages must be executed by NAP exactly once, as long as the protocol does not block for an unbounded amount of time.

Efficient resource usage. In Section 2.4 we observed that some of the very early work on fault-tolerance for itinerant computations required that every place an itinerant computation visits be replicated. In this work, failures were masked by having multiple instances of each stage executing concurrently [126]. However, mobile agent computations are typically deployed because different places provide different resources. Hence, NAP must not require that stages or the stage execution is replicated for fault-tolerance when there are no failures. This can be satisfied by adhering to the primary-backup approach that we discussed in Section 2.3.2.

6.3 Fault-Tolerant Mobile Agent Computations

Based on the derived requirements specified in the previous section, we now specify the operations we require for fault-tolerant mobile agent computations.

The TACOMA mobile agent system originally supported recovery from crashes by use of *cabinets* [91]. Cabinets provide persistent storage of the agent state at the place the agent is currently executing. The cabinet approach works by writing the state of the agent to disk before an agent starts execution at a place. With an option to the `meet` operation, as described in Section 2.1.5, the computation can specify that a mobile agent should be restarted from the state given in the cabinet when a processor and its place recovers from a crash. The main problem with cabinets is that the mobile agent computation blocks until the crashed processor and place hosting the cabinet have recovered, thus violating our non-blocking requirement. A common approach to avoid such blocking is to have sufficient backup places available to detect the crash and recover the mobile agent, similar to the primary-backup approach that we discussed in Section 2.3.2.

To better accommodate failure detection and recovery, our approach is to extend the definition of a mobile agent stage in TACOMA by using *fault-tolerant actions* [156]. A fault-tolerant action FTA can be written as

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

where A is called a *regular action* and R is called the *recovery action* associated with A. A computation can consist of a sequence of fault-tolerant actions:

$$\begin{array}{rl} FTA_1\text{:} & \textbf{action} \;\; A(1) \;\; \textbf{recovery} \;\; R(1) \\ FTA_2\text{:} & \textbf{action} \;\; A(2) \;\; \textbf{recovery} \;\; R(2) \\ & \qquad\quad \ldots \\ FTA_N\text{:} & \textbf{action} \;\; A(N) \;\; \textbf{recovery} \;\; R(N) \end{array}$$

When a sequence of fault-tolerant actions is used, $A(i)$ specifies the ith regular action. $R(i)$ is the recovery action associated with $A(i)$. Similar to a mobile agent stage, each action $A(i)$ executes at a place. If an action $A(i)$ is detected as failed, the corresponding recovery action $R(i)$ is executed. However, we assume that $R(i)$ does not execute at the same place as $A(i)$, since crash failures can only be tolerated when there is sufficient capacity to continue execution elsewhere. Hence, $R(i)$ for $A(i)$ is executed by one or more backup places.

Fault-tolerant actions are general enough to program any kind of fault-tolerance scheme based on detection and recovery. For example, given an operation undo/redo mechanism [21], fault-tolerant actions can be used to implement atomic transactions. Also, masking intermittent crash failures can be implemented by having, for each regular action A, the recovery action of R be identical to A.

We extend the definition of an action further in Section 6.4.1.

6.3.1 Output Commit

A mobile agent computation often needs to communicate with the environment to perform an *external action*. An external action can be to print a document on a printer or format a disk drive. Such external actions cannot always be rolled back as part of the recovery procedure when there is a failure. Hence, before starting an external action, the state leading up to that external action must be recoverable if a failure occurs. This is commonly known as the *output commit* problem [48].

A similar problem surfaces as a consequence of using the primary-backup approach: the environment of the backup places executing the recovery code is different from the environment executing the mobile agent stage. This limits what the recovery code can do. For example, consider a mobile agent stage s at place p executing an output commit action that queries the user for a Uniform Resource Locator (URL) u. The document that u refers to is subsequently retrieved in s. The task of the corresponding recovery code r is to log to a file that retrieving u failed. Assume that s fails, and that place q detects this failure. However, since the recovery code r executing on q does not know the value of u, it cannot log that retrieving u failed.

In the work of Alvisi et al. [47], the output commit problem is specified in the context of rollback-recovery protocols. An approach to solve the output commit problem in rollback-recovery protocols, is to save the state required to roll back an action on stable storage before the application processes the state further. Doing so allows the saved state to be read by the recovery code after a failure occurs. As an analogy to our example, the mobile agent stage at p would need to make the URL u from the user available to all its backups before retrieving u. By following this approach, output commit problems are handled by using the backups as stable storage and then distributing the state to the backups before the output commit action is executed. To facilitate this, NAP provides an operation called **checkpoint** that enables a mobile agent stage to distribute the mobile agent state to its set of backups. Similar to the **move** and **terminate** operations, **checkpoint** terminates a mobile agent stage, and the next stage starts executing on the same place when the state of the agent has been distributed to its backups.

6.3.2 Example: License Checker

We now illustrate the use of fault-tolerant actions with a simple mobile agent computation that uses TACOMA folder operations to manage the mobile agent state. The mobile agent starts executing at an originating place o, and visits a set of places, specified as a parameter that is stored in the `place` folder. For each place p visited, the mobile agent creates a folder `h_p` that contains the action the agent executed at place p or whether it detected place p as failed. These folders are returned to the originating place o when all places have been visited.

The agent is required to update the places with some care. In the event of a place crashing while the changes are taking place, the computation that started the agent must be notified. The crash may be due to a bug in the mobile agent, so such information is useful for debugging. It may also be that the place was crashed in an effort to thwart the mobile agent.

The mobile agent computation consists of the four fault-tolerant actions **visit**, **update**, **alert**, and **report**. We assume that the originating place does not crash, but it is straightforward to rewrite this program to use a set of backup places should one wish to tolerate failures of the originating place. The agent is started with the operation **init**, which causes the first action **visit** to be started at the first place specified in the `place` folder.

Because we assume that the originating place o does not crash, there is no handling of **init** failing. Even if we did allow the originating place to crash, there are no backups when **init** is executed. Specifying a recovery action would be futile.

The four actions for our license checker agent are:

visit This action determines how to proceed based on the license file and the following rules A1-A3:

A1 If the file `license` exists and contains a valid key, then the mobile agent renames the file `program` to `old_program` and writes a new file `program`.[1]

[1] A real system would execute action A1 as a transaction, but we ignore this detail for simplicity of

A2 If the file `license` exists and contains the word "demo", then the mobile agent takes no action.

A3 Otherwise, the mobile agent deletes the file `program`.

The action creates a folder `h_p` with the name of the place and records in it which of the rules A1-A3 to follow. The action terminates with a **checkpoint** leading to the action **update**.

The recovery action creates a folder h_p using the name of the place p and records in it the fact that this place was not available. The recovery action terminates with a **move** of the action **visit** to the next place if there is another place to visit. Otherwise, it terminates with a **move** of the action **report** to the originating place.

update This action updates the files as instructed by the contents of the `h_p` folder, and records the fact that the files were updated in the `h_p` folder. The action terminates with a **move** of the action **visit** to the next place if there is another place to visit. Otherwise, it terminates with a **move** of the action **report** to the originating place.

The recovery action records in `h_p` that the place failed before the action could take place. If there is a next place to visit, the recovery action executes an **init** of a new agent to the originating place where the action **alert** is executed. The recovery action then terminates with a **move** of the action **visit** to the next place. If there is no next place to visit, the recovery action terminates with a **move** of the action **report** to the originating place.

alert This action writes a message to the user who initiated the agent indicating that a place crashed before the action that updates the local file system completed. The local files may have been correctly updated, or the crash may have left the file system in an inconsistent state. The action terminates with a **terminate**. The recovery action is **terminate**.

report This action writes the current contents of the briefcase into disk file in a predefined file location. The recovery action does the same thing. Both actions terminate with **terminate**.

6.3.3 Backup Management

In Section 2.3, we observed that an f fault-tolerant computation withstands f failures within a bounded time interval. A way to characterize an f fault-tolerant *itinerant* computation is the **Bounded Crash Rate**: For any integer $0 \leq i \leq f$, there can be no more than i failures of places during the maximum period of time a mobile agent uses to traverse i distinct places. A property of the Bounded Crash Rate characterization is that f is fixed during the entire itinerant computation, which allows an implementation

exposition.

to make backup management (i.e., adding and removing backups to maintain f backups during the computation) *transparent* to the mobile agent.

However, transparent backup management has disadvantages. First, choosing a value for f is difficult for mobile agent computations. Traditionally, backup places are assumed to have similar reliability. This assumption is questionable in grid environments. Here, f would instead vary as a function of i) the place being visited by the agent, ii) how long the visit lasts (since the probability of multiple concurrent failures increases with the length of the visit), and iii) the places acting as backups.

An alternative to transparent backup management is to have the set of backups chosen by the mobile agent computation itself, similar to how mobile agents **move** to satisfy resource requirements:

- Choose backups based on the CPU load of the places that the agent has currently visited. A lightly loaded place may be preferable since it may execute recovery code faster and may be less likely to fail due to the load on the place.

- Choose backups according to software requirements of the recovery code.

- Choose backups based on the network topology. Our experience with WAMW in Chapter 5 has shown that network communication across local area network boundaries should be minimized to avoid problems caused by non-transitive and asymmetric communications.

- Choose backups based on the suspicion of malicious software. If the computation moves to a network where the places are likely to host software to foist the computation into a bad state, backups should be chosen on the likelihood of not being compromised.

An approach to accommodate a more flexible backup management scheme is to encode the place that executes the regular action and the associated set of backups as part of the mobile agent state. More specifically, we encode the set of places P in a totally ordered *failover list* $P = [p_1, p_2, ..., p_n]$ for n places, where the first place in the failover list is the place that executes the regular action, and the remaining $n-1$ places are backups that execute the recovery action. An action $A(i)$ or $R(i)$ can modify this failover list during the execution of the stage to specify the place that will execute the next regular action $A(i+1)$ as well as the place or places that will execute the next recovery action $R(i+1)$.

The position in the failover list defines the *failover order* for the places. For instance, the failover list $[p_1, p_2, p_3]$ for regular action $A(1)$ specifies that $A(1)$ should execute at place p_1, and if $A(1)$ fails at p_1, place p_2 should execute the recovery action $R(1)$, followed by place p_3 executing $R(1)$ if p_2 fails.

Enforcing the failover order defined by the failover list requires that NAP implements a protocol where the places in the failover list agree on the contents of the failover list. In the following section, we derive a protocol specification for NAP that specifies the semantics of this protocol and the sequencing of regular action and recovery action execution based on the failover list and its failover order.

6.4 Deriving a Specification of NAP

Our strategy for deriving a specification for NAP starts by defining the exact properties we want NAP to satisfy in Section 6.4.1. We then develop a simple protocol based on election in Section 6.4.2; we call this protocol SNAP. We show that SNAP satisfies the properties of NAP, but has too coarse granularity to be practical. In Section 6.4.3, we refine SNAP, and use the resulting protocol to explore some tradeoffs. In Section 6.4.4 we look more carefully at the meaning of an action completing without failing. In Section 6.4.5 we change the derivation into a set of rules executed by participants of the protocol. We call this protocol NAP2 and show that it satisfies all of the NAP properties.

The derivation is performed without using the mobile agent operation terminology specified in Section 6.3, but we explain how to implement the **move**, **checkpoint**, **terminate**, and **init** mobile agent operations based on the resulting protocol in Section 6.6.

6.4.1 Definitions

Let p be a place, P be the set of places involved in the computation, F be the set of failed places where $F \subset P$, and F_p be the set of places that a place p has detected as failed. $A(i)$ is the ith regular action, and $R(i)$ the ith recovery action. We use the term *terminate* to specify that the execution of an action has completed. If the execution terminates by failing, then this is explicitly stated. We further refine the meaning of an action terminating by failing in Section 6.4.4.

NAP satisfies:

P1 For all $i > 0$, $A(i)$ eventually starts executing.

P2 $A(i)$ terminates exactly once.

P3 $A(i)$ will not start executing before all $A(i' : i' < i)$ have terminated and all $R(i' : i' < i)$ that have started executing have terminated.

P4 Once $A(i)$ starts executing, neither $A(i')$ nor $R(i')$ for $i' < i$ will start executing.

P5 $R(i)$ starts to execute iff $A(i)$ terminated by failing.

P6 If $R(i)$ starts to execute, then all $R(i)$ that have started executing earlier have terminated by failing.

P7 If $R(i)$ starts to execute, then some execution of $R(i)$ will terminate without failing.

Properties P1 through P4 specify how regular actions execute: each regular action starts executing in order and without overlapping previous actions. Properties P5 through P7 specify how recovery actions execute: only after the failure of a regular action, and only one action terminates without failing.

The state of a place, $s(p)$, is specified by a pair $< r, i >$:

- $s(p).r$ specifies a totally ordered sequence of unique places $[p_1, p_2, ..., p_n]$ for a finite value of n. We refer to this sequence as the failover list.
- $s(p).i$ is a positive integer that specifies the version of the failover list r. There can only be one version of the failover list for each stage, so i also specifies the stage of the mobile agent computation at place p.

Thus, for a place p, $s(p).r$ is the r component of the state of p, and $s(p).i$ is the i component of the state of p. If $p \notin s(p).r$ then we denote the state of p as $<>$. We can set the state of a place p to $<>$ by setting $s(p).r$ to the empty sequence $[]$.

A failover list r can be indexed by its places, where $r[1]$ is the first place in the list, $r[2]$ is the second place, and so on. When required, we refer to a specific version i of the failover list r as $r(i)$. Failover lists, F, and P are operated upon with set operations. For failover lists, set operations apply to the sets whose elements are places in the list.

Each place p has its own local view F_p of F. We assume no false failure detections: $F_p \subseteq F$. We also assume that F monotonically increases (i.e., places do not recover from failures), so F and F_p for all p do not become smaller. We also assume failures are eventually detected:

$\forall p, q : \Box(q{\in}F \Rightarrow \Diamond(q{\in}F_p \vee p{\in}F))$

A place p is electable if it has not failed, and it has itself in its failover list. A place p is i-electable if it is electable and has version i of its failover list:

p electable : $\wedge p \in P \backslash F$
$\wedge p \in s(p).r$

p i-electable : $\wedge p$ electable
$\wedge s(p).i = i$

Given a sequence of places $r : p \in r$, let $B(r, p)$ be the maximal prefix of r that does not contain p. For example, if $r = [a, b, c, d, e]$, then $B(r, c) = [a, b]$. We will only use $B(r, p)$ when r contains p.

A place p is elected if it has detected that all places before it in the failover list have failed, and it has the smallest failover list version of all the electable places. A place is i-elected if it is both elected and is i-electable:

p elected : $\wedge p$ electable
$\wedge B(s(p).r, p) \subseteq F_p$
$\wedge \forall q \in P : (q \text{ electable}) \Rightarrow s(q).i \geq s(p).i$

p i-elected : $\wedge p$ i-electable
$\wedge p$ elected

For now, assume the following property:

OnlyOneVersion $\Box(\exists r(i) : \forall p \in P : (p \text{ } i\text{-electable}) \Rightarrow (s(p).r = r(i)))$

That is, it is always the case that all places that have the same version have the same failover list. If OnlyOneVersion holds, then only one place can be elected at any time. We show this next by proving that property Mutex holds.

Mutex No more than one place can be elected at any time.

Proof. We show that Mutex holds by contradiction. By the definition of p elected above, only an electable place p with the smallest failover list version can be elected. Assume that two places p and q are elected. By definition, both are electable and have the smallest version i of all versions of all electable places, so both places are i-electable. By OnlyOneVersion, both places have the same failover list $r(i)$. However, one of the places must occur first in the failover list. Assume without loss of generality that this place is p. Since q is elected, $p \in F_q$, and since we assume a perfect failure detector for F, $p \in F$. Thus, p is not electable, which is our contradiction. $\square$

By this, OnlyOneVersion is fundamental to understanding how to implement NAP. All that OnlyOneVersion requires is that the places with the same version have the same failover list. Hence, as long as the state of a place is updated atomically (i.e., indivisibly with respect to failures), and the version number does not decrease, we have much of a protocol that implements NAP.

Before specifying a protocol that implements NAP, we make the following assumptions about the environment where the protocol executes:

F1 For all values of $i > 0$: it is always true that some place in $r(i)$ does not fail: $\forall i : i > 0 : \Box((r(i) \backslash F) \neq \emptyset)$. This implies that each $r(i)$ contains at least one non-failed place.

F2 For each $i > 0$, the execution of $A(i)$ and of $R(i)$ terminates in finite time.

6.4.2 SNAP: A Simple NAP

Pseudocode 6.4.1 specifies a simple algorithm for NAP, which we call SNAP. We specify SNAP as a sequence of actions that are executed like a traditional non-distributed computer program. However, in our case, the actions might executed at different places. We use this syntax to allow us to focus on a specific problem; in Section 6.5 we use a more conventional syntax that makes the message flow of the algorithm explicit. Actions between $\ll$ and $\gg$ brackets are assumed to be executed atomically across all $p \in P$. The syntax of the `when` statement that is used in the algorithm is as follows:

`when` $S1$ Q `until` $S2$

Assume that $S1$ and $S2$ are predicates that are initially false and that Q is a sequence of actions. The `when` statement evaluates $S1$ and $S2$ until either predicate evaluates to true. If $S1$ evaluates to true and $S2$ to false, then Q is executed, and the `when` statement completes. If instead $S2$ evaluates to true, then the `when` statement completes without Q being executed.

We also use statements of the following form:

$\forall p \in P$: Q

The semantics for this statement is that Q is a sequence of actions that is executed on all places p in parallel.

Pseudocode 6.4.1 SNAP - A simple algorithm that satisfies NAP.

```
01: [Initial state: $\forall p \in P \backslash F$ : if $p \in r(1)$: $s(p) =< r(1), 1 >$ else: $s(p) =<>$]
02: $\ll i = 1 \gg$
03: while (true) {
04:     [Loop invariant: $\forall p \in P \backslash F$ : if $p \in r(i)$: $s(p) =< r(i), i >$ else: $s(p) =<>$]
05:     $\forall p \in P \backslash F$:
06:       $\ll$
07:         when ($p$ elected){
08:           if ($p == p.r[1]$): $A(i)$
09:           else: $R(i)$
10:         } until ($A(i)$ or $R(i)$ terminates without failing)
11:       $\gg$
12:     $\ll$
13:       $\forall p \in r(i+1) \backslash F : s(p) =< r(i+1), i+1 >$
14:       $\forall p \in (r(i) \backslash r(i+1)) \backslash F : s(p) =<>$
15:     $\gg$
16:     $\ll i = i + 1 \gg$
17: }
```

Informally, SNAP works as follows. For each value of i, a place p is either elected to execute the regular action $A(i)$ (if p is first in the failover list), or the recovery action $R(i)$. More specifically, the algorithm starts with an initial failover list with version 1 in line 1. At some point during execution, a place is elected in line 7. If this is the first place in the failover list, the regular action is executed in line 8. If the elected place is not the first place in the failover list, the recovery action for the same version is executed in line 9. If the action terminates, all places in the next failover list have their state set to the next failover list and the next version in line 13. All places that are not part of the next failover list, set their state to empty in line 14. The version counter is then incremented in line 16, and the election of a place for executing an action starts again in line 7.

We now show that with SNAP, all places with the same version have the same failover list, thus satisfying property OnlyOneVersion:

Proof. OnlyOneVersion initially holds since all non-failed places $p \in r(1)$ where $s(p).i$ is 1 (i.e., 1-electable places) have $s(p).r$ set to the failover list $r(1)$.

We show that SNAP satisfies OnlyOneVersion by contradiction. Assume that there is a place q that is i-electable yet $s(q).r \neq r(i)$. For all $p \in P$, the only statement where $s(p)$ is changed is in lines 13 and 14, where the r and i component of the state $s(p)$ are atomically updated. By the completion of line 15, all non-failed $p \in r(i+1)$ will set their state $s(p)$ to $< r(i+1), i+1 >$. Assume that place q is one of those places. By definition, $s(q).i$ must be $i+1$ for q to be $i+1$-electable. However, by line 13 in the algorithm, if $s(q).i$ is $i+1$ then $s(q).r$ must be $r(i+1)$, which is our contradiction. $\square$

The SNAP algorithm also implements properties SyncChange and Move that we will later use to prove that SNAP satisfies properties P1-P7:

SyncChange If $s(p)$ is $< r(i), i >$, then no place q will have $s(q) =< r(i'), i' >$ for $i' \neq i$.

Proof. The initial state $s(p)$ for all p is $< r(1), 1 >$ if $p \in r(1)$. From this, all places in $r(1)$ have version 1 of the failover list, so initially no place q can have a failover list with $i' \neq 1$. SNAP executes lines 13 and 14 of the algorithm atomically for each increasing value of i. After line 15 completes, the state a place p for all p is either $< r(i+1), i+1 >$ if $p \in r(i+1)$ or $<>$ if $p \notin r(i+1)$. Hence, after line 15 has completed, no place q will have $s(q) =< r(i'), i' >$ for $i' \neq i$. $\square$

Move No place p sets $s(p)$ to $< r(i+1), i+1 >$ before an execution of $A(i)$ or $R(i)$ terminates without failing.

Proof. For increasing values of i, either $A(i)$ or $R(i)$ is executed in lines 8 and 9 of the algorithm, respectfully. Lines 13 and 14 of the algorithm execute iff lines 7 through 10 of the algorithm terminate without failing. The state $s(p)$ is atomically set to $< r(i+1), i+1 >$ for all $p \in r(i+1)$ in line 13. Since line 13 executes iff lines 7 through 10 terminated without failing, no place p sets $s(p)$ to $< r(i+1), i+1 >$ unless lines 7 through 10 terminated without failing, implying that $A(i)$ or $R(i)$ executed without failing. $\square$

Given SyncChange, it is straightforward for a place to determine that it is elected. Recall:

$$\begin{aligned} p \text{ elected} : &\wedge p \text{ electable} \\ &\wedge B(s(p).r, p) \subseteq F_p \\ &\wedge \forall q \in P : (q \text{ electable}) \Rightarrow s(q).i \geq s(p).i \end{aligned}$$

SyncChange implies that if p and q are electable, they have the same version. Thus, the third conjunct of p elected is trivially true, and a place p is elected when p is electable and p knows that all places before it in its failover list have failed.

We now show that SNAP satisfies the properties P1-P7 that specify NAP. The first property, P1, poses a metaphysical problem: is there a way to tell whether an action $A(i)$ starts executing and then fails? Consider a place $p.r[1]$ that executes in line 8, and starts executing $A(i)$ but crashes while executing the first CPU instruction of the action. From this we can say that $A(i)$ started executing. If the place p crashed before executing line 8, then we can say that $A(i)$ did not start executing. Externally, however, we can not distinguish between these two cases. Thus, we adopt the following pragmatic approach: if the local state of $p.r[1]$ before crashing would have led to $A(i)$ executing, we say that p started executing $A(i)$. We will expand on this issue in Section 6.4.4.

P1 For all $i > 0$, $A(i)$ eventually starts executing.

Proof. Since an action $A(i)$ can fail before executing the first CPU instruction of the action, we can say that $A(i)$ starts executing if line 8 starts to execute. For any value of i, lines 8 or 9 of the algorithm terminate because $r(i)$ is finite, and by F2, the execution of $A(i)$ and $R(i)$ must take finite time. Lines 13 and 14 of the algorithm terminate because the sequence length of $r(i+1)$ and $r(i)$ is finite. Thus, for each value of i, $A(i)$ starts to execute. □

P2 $A(i)$ terminates exactly once.

Proof. From OnlyOneVersion, since all places with the same version have the same failover list, only one place can be first in this list, and thus execute $A(i)$. □

P3 $A(i)$ will not start executing before all $A(i' : i' < i)$ have terminated and all $R(i' : i' < i)$ that have started executing have terminated.

Proof. From Move, since no place p sets $s(p)$ to $< r(i+1), i+1 >$ before an execution of $A(i)$ or $R(i)$ terminates without failing, $A(i)$ will not start executing before $A(i')$ or $R(i')$ for $i' < i$ have terminated. □

P4 Once $A(i)$ starts executing, neither $A(i')$ nor $R(i')$ for $i' < i$ will start executing.

Proof. From SyncChange, when $A(i)$ starts executing $s(p)$ is $< r(i), i >$. SyncChange states that if $s(p)$ is $< r(i), i >$, then no place q will have $s(q) =< r(i'), i' >$ for $i' \neq i$. Hence, by lines 8 and 9 in the algorithm, neither $A(i')$ nor $R(i')$ for $i' < i$ can start executing. □

P5 $R(i)$ starts to execute iff $A(i)$ terminates by failing.

Proof. From line 8 of the algorithm, $A(i)$ will only be executed if a place p is elected and p is the first place in $r(i)$. If a place q in $r(i)$ is elected, and q is not the first place in $r(i)$, then by the definition of i-elected, $A(i)$ must have terminated by failing at p, and by line 9 in the algorithm, $R(i)$ is executed at q. □

P6 If $R(i)$ starts to execute, then all $R(i)$ that have started executing earlier have terminated by failing.

Proof. If executing $A(i)$ at p terminates by failing, then by P5, $R(i)$ will start executing at a place q. By line 7 in the algorithm, $R(i)$ will only start executing again if a place other than q is elected. However, by the definition of elected, this can only happen if q has failed, which means that $R(i)$ terminated by failing while executing at q. Hence, $R(i)$ will not start executing again unless the previous execution terminated by failing. □

P7 If $R(i)$ starts to execute, then some execution of $R(i)$ will terminate without failing.

Proof. From our assumption F1 it is true that some place in $r(i)$ does not fail, so each $r(i)$ contains at least one non-failed place q. If $A(i)$ terminated by failing, then by P5, $R(i)$ will start executing. If all places in $r(i)$ except q fail, then by line 7 in the algorithm, q will be elected and $R(i)$ will terminate without failing at q. Hence, some execution of $R(i)$ will terminate without failing. □

6.4.3 Improving SNAP

SNAP is impractical since it has several coarse actions that have to execute atomically within the `while` loop. Breaking up the coarse atomic actions into a more fine-grained protocol allows the parts of the algorithm that update the failover list to be implemented in a way that makes the algorithm perform faster. The algorithm in Pseudocode 6.4.2, makes SNAP more fine-grained by breaking up the atomic actions of lines 12 through 14.

Pseudocode 6.4.2 A more fine-grained version of SNAP.

```
01: [Initial state: $\forall p \in P\backslash F$ : if $p \in r(1)$: $s(p) =< r(1), 1 >$ else: $s(p) =<>$]
02: $\ll i = 1 \gg$
03: while (true) {
04:       [Loop invariant: $\forall p \in P\backslash F$ : if $p \in r(i)$: $s(p) =< r(i), i >$ else: $s(p) =<>$]
05:       $\forall p \in P\backslash F$ :
06:         $\ll$
07:           when ($p$ elected) {
08:             if ($p == p.r[1]$): $A(i)$
09:             else: $R(i)$
10:           } until ($A(i)$ or $R(i)$ terminates without failing)
11:         $\gg$
12:       $\forall p \in r(i+1)\backslash F$ : $\ll s(p) =< r(i+1), i+1 > \gg$
13:       $\forall p \in (r(i)\backslash r(i+1))\backslash F$ : $\ll s(p) =<> \gg$
14:       $\ll i = i + 1 \gg$
15: }
```

With the changes in lines 12 and 13, Move still holds, but SyncChange does not since all the places no longer change their state atomically. However, SyncChange can be weakened. If two places have different non-empty states, then their version must only differ by one:

OnlyTwoVersions $\Box(\exists i : \forall p \in P\backslash F)$:
$\vee s(p) =< r(i), i >$
$\vee s(p) =< r(i+1), i+1 >$
$\vee s(p) =<>$

This weakening of SyncChange means that it is more difficult for a place p to determine whether it is elected. Not only does all places preceding p in $s(p).r$ need to be in F_p, but p needs to know that there are no electable places with a version less than $s(p).i$.

One approach to simplify the process of determining whether a place is elected, is to impose an order on how the places change their states. Consider the following property, OrderedChange, where $P(i)$ is the set of places in the state $< r(i), i >$:

OrderedChange $\forall p \in P(i), q \in P(i+1)$:

$\vee\; q \in B(r(i), p)$

$\vee\; p \in B(r(i+1), q)$

OrderedChange and OnlyTwoVersions together simplifies for a place p to determine if it is elected. Let a place p be in $< r(i), i >$. If p knows all places p' not in $<>$ are in $< r(i), i >$ or in $< r(i-1), i-1 >$, then p knows it is elected when $B(s(p).r, p) \subseteq F_p$ and all places in $r(i-1)$ are in $P(i)$, in $<>$, or in F_p. Thus, when p enters $P(i)$, it waits for all places in $r(i-1)$ to no longer be in $P(i-1)$. Once this happens, for as long as it remains in $P(i)$, p can elect itself when $B(s(p).r, p) \subseteq F_p$.

We will further refine property OrderedChange in Section 6.4.4. Before doing so, we examine whether there is an order that can be imposed on how places assign failover lists to their state so that OrderedChange is satisfied. Initially, OrderedChange holds because all non-failed places in $r(i)$ are in $P(i)$ and the rest are in $<>$. Since there are no places $q \in P(i+1)$, OrderedChange trivially holds. There are three sets of places whose state is changed:

Leaves The set of places in $r(i)\backslash r(i+1)$. Places in this set change from $P(i)$ to $<>$.

Moves The set of places in $r(i) \cap r(i+1)$. Places in this set change from $P(i)$ to $P(i+1)$.

Joins The set of places in $r(i+1)\backslash r(i)$. Places in this set change from $<>$ to $P(i+1)$.

A place in set Leaves can be changed immediately to $<>$ and OrderedChange will continue to trivially hold. The places in set Leaves can be changed to $<>$ in any order. OrderedChange can be kept true in two ways when changing the places in set Moves:

C1 Change the places in increasing $r(i)$ order. Doing so ensures that $q \in B(r(i), p)$ holds.

C2 Change the places in decreasing $r(i+1)$ order. Doing so ensures that $p \in B(r(i+1), q)$ holds.

Once all the places in set Leaves have changed to $<>$ and set Moves have been changed to $P(i+1)$, there are no places remaining in $P(i)$. Thus, OrderedChange trivially holds again. The places in set Joins can be changed to $P(i+1)$ as soon as all the places in set Leaves and set Moves have changed. They can be changed in any order.

The second choice, C2, for updating the places in set Moves has the property that if all the places in $B(r(i+1), q)$ are in F_q, then any place that has version i of the state has failed. Thus, a place can use the simplified rule to determine that it is elected:

p i-elected: $\wedge p$ i-electable

$\wedge \forall q \in B(s(p).r, p) : q \in F_p$

Hence, using C2 and the simplified rule is attractive since a process p knows that it is elected when it is electable and all processes before it in the failover list has failed.

Using C2 for updating the places in set Moves combined with the other update rules for set Leaves and set Joins gives the following protocol:

1. Change all places in $r(i) \backslash r(i+1)$ from $P(i)$ to $<>$ in any order.
2. Change all places in $r(i) \cap r(i+1)$ from $P(i)$ to $P(i+1)$ in decreasing $r(i+1)$ order.
3. Change all places in $r(i+1) \backslash r(i)$ from $<>$ to $P(i+1)$ in any order.

Despite this convenience of being able to use the simplified rule for i-elected, this protocol has a drawback that will lead us to use C1 rather than C2. To show why this is so, we expand on the indistinguishability that was brought up when arguing that SNAP implemented P1 in Section 6.4.2.

6.4.4 Termination of an Action

Consider the following execution based on the protocol with C2 given in the previous section. Version 1 of the state has failover list $[a, b, c, d]$ and version 2 of the state has failover list $[c, d, b]$, where a, b, c, d are places.

1. $\{a, b, c, d \text{ in } < [a, b, c, d], 1 >\}$
2. a is 1-elected
3. a fails
4. b is 1-elected
5. $R(1)$ at b starts to execute
6. $R(1)$ at b terminates without failing
7. $\{$ start change to $< [c, d, b], 2 >\}$
8. b changes to $< [c, d, b], 2 >$
9. d changes to $< [c, d, b], 2 >$
10. b fails
11. c is 1-elected
12. $R(1)$ at c starts to execute

This execution violates P6 because $R(1)$ terminated at b without failing in step 6, and $R(1)$ started to execute at c in step 12. The following execution shows how P5 can be violated.

1. $\{a, b, c, d \text{ in } < [a, b, c, d], 1 >\}$
2. a is 1-elected

3. $A(1)$ at a terminates without failing
4. { start change to $< [c, d, b], 2 >$}
5. a fails
6. b is 1-elected
7. $R(1)$ at b starts to execute

In step 7, $R(1)$ starts to execute at b although $A(1)$ terminated without failing in step 3, thus violating P5.

The previous executions highlight a fundamental problem: What is the meaning of an action terminating without failing? For example, a place p could execute all of the code of an action and then fail before starting the protocol that starts the next action. Or, a place p could send a message to a place q to start the protocol for starting the next action and then both q and p could fail. In both cases, there is no external evidence of the action completing even though all of the code of the action has been correctly executed. An approach to avoid this problem is to assume the following environmental property:

F3 If the state of a non-failed place is consistent with an execution where a place p executed an action $A(i)$ or $R(i)$ that terminated by failing, we assume that p executed $A(i)$ or $R(i)$, and that the action terminated by failing.

This definition is internally consistent, but an external observer might be able to tell the difference. This creates an output commit issue [47]: external actions must be idempotent. F3 implies, as we have already assumed, that a place may never start executing an action yet we will assume that it started executing the action, which then terminated by failing.

Consider the first execution. After step 10, there is evidence that $R(1)$ at b terminated without failing: d is propagating the change from failover list $r(1)$ to $r(2)$. However, if d were to fail at the same time b does, then the evidence could be lost, and so, in terms of F3, P6 is not violated. In the second execution, in terms of F3, P5 is not violated.

F3 means that a place should not execute $R(i)$ if there is evidence that $A(i)$ or $R(i)$ has executed somewhere else without failing, where "evidence" means that some nonfailed place has changed its state to $< r(i + 1), i + 1 >$. This can be ensured with the following strengthening of OrderedChange:

$$\forall q \in P(i) : P(i + 1) \subseteq B(r(i), q)$$

For q to be i-elected, it is necessary that all places in $B(r(i), q)$ have failed. Since the only places that could be in version $i + 1$ are contained in $B(r(i), q)$, there is no evidence that any place has set its state to $< r(i + 1), i + 1 >$. This strengthening of OrderedChange can be implemented by using C1 to change the places in set Moves. We call this protocol NAP2.

6.4.5 NAP2 Specification

Instead of continuing to specify NAP2 as the execution of a program as with SNAP, we specify NAP2 as a set of rules that each place participating in the protocol implements. We have two definitions NAP2Elected and NAP2Change:

NAP2Elected A place p is i-elected when:

$\wedge s(p).i = i$
$\wedge \forall q \in B(s(p).r, p) : q \in F_p$
$\wedge \forall q \in r(i-1) \backslash F_p : s(q).i \neq i-1$

NAP2Elected states that for a place q to be i-elected, the version of the state is i, all places in $B(r(i), q)$ have failed, and no places are in $< r(i-1), i-1 >$.

NAP2Change Define an operation `change`:
`change` places in P `to` state v `in order` E

In NAP2Change, E defines an irreflexive partial order of the places in P. The states of places in P are changed to v such that:

- For two places $p, q \in P$: if p is before q in E, then it is invalid to set the state of p to v after the state of q has been set to v.
- Let C be the subset of places in P that have changed their state to v. If $C \backslash F$ is always nonempty, then the `change` operation eventually terminates with all places in $P \backslash F$ having set the state to v.

Thus, there are two possible ways for the `change` operation to terminate:

1. All places in $P \backslash F$ have changed their state to v. This *must* be the outcome if the place that initiated the `change` operation does not fail. If the place that initiated the `change` operation fails, then this *may* be the outcome.
2. All places in $P \backslash F$ have left their state unchanged.

The `change` operation can thus be considered a reliable broadcast protocol [76] with the additional constraint that places deliver v consistent with some order.

NAP2 Rules

In NAP2, a place p in P uses the following three rules, R1-R3:

R1 Initially, the state of p is $< r(1), 1 >$ if p is in $r(1)$, and is $<>$ otherwise.

R2 When p becomes i-elected: If p is the first place in $r(i)$ then start executing $A(i)$ else start executing $R(i)$.

R3 When a place p terminates an action, let Q be all the places in $r(i) \cup\ r(i+1)$:

- For $p, q \in Q$, let p be before q in E iff:
  - p and q are both in $r(i)$, and p is before q in $r(i)$.
  - p is in $r(i)$ and q is not in $r(i)$.
- `change` all non-failed places q in $r(i) \cup\ r(i+1)$ `to`:
  - $< r(i+1), i+1 >$ if q in $r(i+1)$.
  - $<>$ if q not in $r(i+1)$.

  `in order` E.

Proving that NAP2 Satisfies NAP

We now prove that NAP2 satisfies the properties P1-P7. Recall the environmental assumptions we made earlier:

F1 For all values of $i > 0$: it is always true that some place in $r(i)$ does not fail: $\forall i : i > 0 : (r(i) \backslash F) \neq \emptyset$. This implies that each $r(i)$ contains at least one non-failed place.

F2 For each $i > 0$, the execution of $A(i)$ and of $R(i)$ terminates in finite time.

F3 If the state of a non-failed place is consistent with an execution where a place p executed an action $A(i)$ or $R(i)$ that terminated by failing, we assume that p executed $A(i)$ or $R(i)$, and that the action terminated by failing.

In addition to F1-F3, we show that the OnlyTwoVersions property we specified in Section 6.4.3 holds. We also specify a new lemma NAP2Progress that is used to show that P1 and P3 hold. We now prove that OnlyTwoVersions and NAP2Progress hold.

OnlyTwoVersions $\Box(\exists i : \forall p \in P \backslash F)$:
$\vee s(p) =< r(i), i >$
$\vee s(p) =< r(i+1), i+1 >$
$\vee s(p) =<>$

Proof. We prove this by induction on i.

Base case $i = 1$: From R1, OnlyTwoVersions initially holds, since all places in $r(1)$ are in state $< r(1), 1 >$, and all places not in $r(1)$ are in $<>$.

Induction step $i > 1$: Assume that there is some version i: all noncrashed places are in $<>$ or $< r(i), i >$. From F1 and F2, we know that there is some place p in $< r(i), i >$ that terminates its action and by R3 executes NAP2Change.

From the definition of NAP2Change, places set their state to $<>$ or $< r(i+1), i+1 >$ in order E. Since p completes the action, p changes its state $< r(i), i >$ to either $<>$ or $< r(i+1), i+1 >$. Thus, all places in $r(i) \cup r(i+1)$ that do not fail change their state from $<>$ or $< r(i), i >$ to either $<>$ or $< r(i+1), i+1 >$. Hence, during the execution of NAP2Change, all nonfailed places are in $<>$, $< r(i), i >$, or $< r(i+1), i+1 >$, and by the end of R3, all noncrashed places are in $<>$ or $< r(i+1), i+1 >$ and OnlyTwoVersions holds. □

NAP2Progress If there exists a place p in $< r(i), i >$ that is i-elected, then for all q in $r(i)$, q eventually fails or q is in $< r(i), i >$.

Proof. We prove this by induction on i.

Base case $i = 1$: In the initial state, all nonfailed q in $r(1)$ are in $< r(1), 1 >$, so NAP2Progress holds.

Induction step $i > 1$: From F1, there is a place x in $r(i-1)$ that does not fail. If p is in $r(i-1)$, then by NAP2Elected, by the time p is elected, x is in $< r(i), i >$ and so by NAP2Change and R2 eventually all q in $r(i)$ that do not fail are eventually in $< r(i), i >$. If p is not in $r(i-1)$, then by R3, x is in $< r(i), i >$ before p is in $< r(i), i >$. So, again by NAP2Change and R3, eventually all q in $r(i)$ that do not fail are eventually in $< r(i), i >$. □

Now that OnlyTwoVersions and NAP2Progress hold, we prove that P1-P7 are satisfied by NAP2.

P1 For all $i > 0$, $A(i)$ eventually starts executing.

Proof. We prove this by induction on i.

Base case $i = 1$: by R1, all places in $r(1)$ have state $< r(1), 1 >$. Since $r(1)$ is a totally ordered sequence, by F1 and NAP2Elected some place will become elected, and by F1 there is only one place that eventually executes rule R2. By F3, this means that $A(1)$ started executing.

Induction step $i > 1$: From F2, some place p in $r(i-1)$ starts executing an action. From NAP2Progress, all places in $r(i-1)$ that do not fail eventually enter $< r(i-1), i-1 >$. From F1, there is some place p in $r(i-1)$ that does not fail, which means that some place that is $i-1$-elected terminates an action, and successfully completes the `change` operation in R3. From F1, there is a place q in $r(i)$ that does not fail, and from R3 this place is eventually in $< r(i), i >$. From R3, each place in $r(i)$ either fails or is eventually in $< r(i), i >$. Hence, there is a place, not necessarily q, that successfully executes an action of version i. From F3, this means that $A(i)$ started executing. □

P2 $A(i)$ terminates exactly once.

Proof. From P1, $A(i)$ eventually starts executing. From R2, only one place in $r(i)$ can execute $A(i)$. Thus, $A(i)$ terminates exactly once. □

P3 $A(i)$ will not start executing before all $A(i' : i' < i)$ have terminated and all $R(i' : i' < i)$ that have started executing have terminated.

Proof. We prove P3 by contradiction. Assume that there is an action $A(i)$ that has started executing before all $A(i' : i' < i)$ and $R(i' : i' < i)$ have terminated. From R2, $A(i)$ will only be executed if a place p is i-elected and p is the first place in $r(i)$. By NAP2Elected, i-elected implies that $s(q).i \neq i'$ for all q in $r(i')$. Hence, by NAP2Progress, a place q in $r(i')$ must have completed the `change` operation without failing by the time p is i-elected, and R3 thus completed at q. This implies that R2 terminated without failing at q, and that $R(i')$ or $A(i')$ executing at q terminated without failing. Hence, when $A(i)$ started executing, all $A(i' : i' < i)$ and $R(i' : i' < i)$ must have terminated, which is our contradiction. □

P4 Once $A(i)$ starts executing, neither $A(i')$ nor $R(i')$ for $i' < i'$ will start executing.

Proof. From NAP2Elected, no place is i-elected until all places in $< r(i-1), i-1 >$ are failed, in $<>$, or in $< r(i), i >$. Since a place executes an action of $i-1$ only when it is $(i-1)$-elected, P4 holds. □

P5 $R(i)$ starts to execute iff $A(i)$ terminated by failing.

Proof. From R2, $A(i)$ will only be executed if a place p is elected and p is the first place in $r(i)$. If a place q is elected, and q is not the first place in $r(i)$, then by NAP2Elected, $A(i)$ must have been detected as failed at p. By F3, detected as failed means that p may have failed before $A(i)$ started executing, during the execution of $A(i)$, or during the execution of the `change` operation in R3. By NAP2Elected and R2, when $A(i)$ is detected as failed, $R(i)$ is executed at q. □

P6 If $R(i)$ starts to execute, then all $R(i)$ that have started executing earlier have terminated by failing.

Proof. If executing $A(i)$ at p terminates by failing, then by P5, $R(i)$ will start executing at a place q. By NAP2Elected and R2, $R(i)$ will only start executing again if a place other than q is elected. However, again by NAP2Elected and R2, this can only happen if the execution of $R(i)$ at q has failed, and the `change` operation in R3 did not complete. Hence, $R(i)$ will not start executing again unless the previous execution terminated by failing. □

P7 If $R(i)$ starts to execute, then some execution of $R(i)$ will terminate without failing.

Proof. From F1, it is true that some place in $r(i)$ does not fail, so each $r(i)$ contains at least one non-failed place q. If $A(i)$ terminated by failing, then by P5, $R(i)$ will start executing. If all places in $r(i)$ except q fail, then by NAP2Elected and R2, q will be elected, and $R(i)$ will terminate without failing at q. Hence, some execution of $R(i)$ will terminate without failing. □

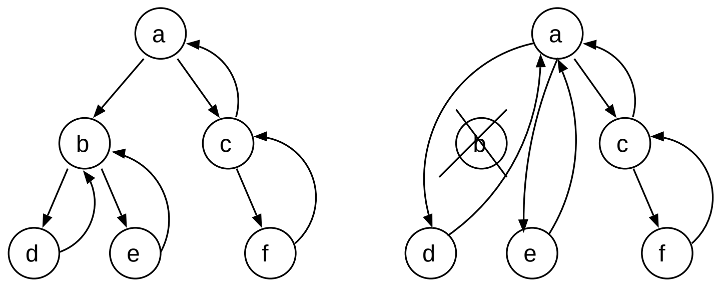


Figure 6.2: Fault-tolerant broadcast with a tree strategy.

6.5 From Specification to Implementation

This section specifies how to map the derived NAP2 specification to an implementation. We start by describing one of the building blocks for doing so, called Fault-Tolerant Broadcast (FTB) [160], in Section 6.5.1. We then specify a variation of FTB in Section 6.5.2 that we use for implementing NAP2 in Section 6.5.3.

6.5.1 Fault-Tolerant Broadcast

The `change` operation specified in Section 6.4.5 that is executed by rule R3 can be considered a reliable broadcast protocol with delivery order constraints, so we use a variation of reliable broadcast to implement `change`. More specifically, we use a refinement of the FTB protocol [160]. The FTB protocol is presented using a tree-based broadcast strategy. The broadcast strategy specifies how a broadcast proceeds, how participants in the protocol monitor each other for failures, and how failures are handled. We start by describing how FTB works with a tree-based broadcast strategy. In the FTB paper [160], the term *processor* is used to identify the participants of the protocol, but without loss of generality we use the term *place* in the subsequent sections.

In FTB, a broadcast strategy is represented by a tree where the root is the place that initiates the broadcast. Assume such a strategy is defined by a graph (V, E), and that there are two places p and q where the relation p *succ* q denotes that q is a successor node to p in the graph, and $succ(p)$ denotes the set of successor nodes for p. Consider the strategy given by the tree to the left in Figure 6.2. The place a that is the root, broadcasts a value m to a group $G = \{a, b, c, d, e, f\}$, where $succ(a) = \{b, c\}$, $succ(b) = \{d, e\}$ and $succ(c) = \{f\}$. The root a ensures that all non-faulty places in G either deliver m or do not deliver m. It does so by sending m to $succ(a)$ and waiting for an acknowledgment from the members of $succ(a)$. The members of $succ(a)$, which are $\{b, c\}$, ensure that unless they fail, all non-failed places in $succ(b)$ and $succ(c)$ deliver m. In general, when a place q in G receives m from p, it is responsible for ensuring that m is delivered by all non-failed places in $succ(q)$. When this obligation is discharged, q sends an acknowledgment to p. Thus, the left tree in Figure 6.2 where there are no failures, a message m will travel from a to b and c, from b to d and e, from c to f, and then the acknowledgment will travel back from f to c, from d and e to b, from c to a, from b to a, and the broadcasts completes with all members of G having delivered m.

Once a place p sends m to $succ(p)$, p monitors for the crash of the members of $succ(p)$. If p detects that a member q of $succ(p)$ has crashed before receiving the acknowledgment from q, then p "adopts" q - in essence, q is removed from $succ(p)$ and the members of $succ(q)$ are added to $succ(p)$. By doing so, p takes over establishing that all of the non-failed places of $succ(q)$ deliver m. Place p sends m to the members of $succ(q)$ and waits for acknowledgment from all places in the changed $succ(p)$. The members of $succ(q)$ acknowledge to p when it is legal to do so. For example, the members of $succ(q)$ can immediately send the acknowledgment to p if they have already sent it to q before q crashed. As an example, consider the right tree in Figure 6.2 where b fails before sending an acknowledgment to a. Since b is in $succ(a)$, a monitors b and upon detecting that b fails, sends m to d and e since they are the members of $succ(b)$. Since d and e have no successors, they immediately send an acknowledgment back to a and the broadcast completes.

FTB must also handle failures of places that are root nodes in the broadcast tree. When a place q receives a value m from its parent place p, q monitors p for failures. It continues to monitor p for failures until it knows that p has received acknowledgment messages from all places in $succ(p)$ and that p has acknowledged its parent (unless p is the root of the strategy). If q detects p's failure, then q sends the value m to the places in $succ(p)$ and adds the places in $succ(p)$ to the set of places q is monitoring.

FTB can be instantiated with a *linear* broadcast strategy. With a linear strategy, each place in the strategy only has a single successor. Thus, a place p broadcasts a value m by sending it to its successor q and waits for an acknowledgment from q. Once p sends m to q, it monitors q for failures. If p detects q's failure before getting an acknowledgment, then p sends m to the successor of q, and waits for an acknowledgment and monitors that place. When a place q receives a value m from a place p, q monitors p for failures, and sends the acknowledgment message to the parent place for p if p fails. Thus, if the broadcast strategy is linear, then sending the acknowledgment for message m is done in the same manner as sending m, except that the strategy is reversed.

FTB assumes that all places initially know the broadcast strategy. This is not required since a place can learn the broadcast strategy with the first message that is broadcast. Before receiving the first broadcast message, a place needs not monitor any other place and so knowing the broadcast strategy is unnecessary. Hence, FTB can be used to initialize itself.

6.5.2 Fault-Tolerant Forwarding

We now specify a protocol that we call Fault-Tolerant Forwarding (FTF), which is based on FTB with linear broadcast strategy. Informally, FTF works as follows. The first place in the strategy starts the FTF by sending a message m to its successor. When a place receives a message m, it sends it to its successor if it has one. A place p monitors the first non-failed successor if there is one, and if the successor fails, then p sends m to the next non-failed successor. p continues to do this until monitoring is explicitly disabled or p fails. If a place finds itself with no non-failed successor, then it executes a specified action. Note that as long as not all of the places in the strategy fail and one place starts executing the FTF, then eventually a non-failed place will execute the specified action.

Pseudocode 6.5.1 gives the algorithm for FTF. **last_up**(*strategy*, *place*) is a predicate that is true when the *place* in the *strategy* has no successor: either the *place* is the last in the *strategy*, or all places after the *place* in the *strategy* have failed. **first_up_after**(*strategy*, *place*) is a function that returns the first non-failed successor of *place* according to the *strategy*. If there are no non-failed successors of *place* in the *strategy*, the function returns NULL. The semantics of the `when` statement is identical to the definition given in Section 6.4.2. The `send` statement sends a message to a place and the `execute` statement executes an action *action*. **receive** is a predicate that is true when a message has arrived from a place and is ready to be received by the computation executing the FTF. The values of the received message are represented by the variables specified in the signature of **receive**. Note that the values in the message are only assigned to the local values if the value of v in the message is larger than the local value of *version*. Thus, *version* is used to distinguish between superseded and newer versions of the message being sent, and must be increased for each consecutive FTF. For each place participating in the FTF, *local_place* is initialized with the name of the place.

The `when` statements in lines 17 and 22 are assumed to start evaluating when *monitor* is set to true. The `when` statement in line 32 is assumed to start evaluating when the first FTF is invoked. While not explicitly shown in the algorithm, all the `when` statements are assumed to be re-invoked when $S1$ is true and Q has executed, until the $S2$ predicate is true. As an example, this means that the `when` statement in line 32 is evaluating as long as the program invoking the FTF is executing. Doing so ensures that messages sent by a place in line 30 of the algorithm are always received and handled by the receiving place.

An FTF is started by invoking the **ftf** function at line 8. If the version v of the message is greater than the locally stored *version*, *monitor* is set to true, causing lines 18-20 or 23-30 to execute depending on the outcome of evaluating **last_up**. Initially, *last_sent_to* is NULL, so if a place p has a successor q in the strategy, !**last_up**(*strategy*, p) evaluates to true, *last_sent_to* is set to q, and the message is sent from p to q in line 30. If a place q in the strategy is detected as failed by a place p, and q is not the last place in the strategy of the FTF, !**last_up**(*strategy*, p) evaluates to true since *last_sent_to* is no longer equal to the first non-failed place q. Hence, the message is sent to the successor of q, *last_sent_to* is set to the successor of q, and the failed place q is removed from *strategy* in line 27. If the place q is the last place in the strategy of the FTF, **last_up**(*strategy*, *local_place*) evaluates to true and the *action* is executed at p in line 20.

We can use two FTFs to express a protocol that is similar to linear FTB, which we call FTF-FTB. Sending a message m with FTF-FTB using a linear strategy r is an FTF of m with r followed by an FTF of the acknowledgment of m with the reverse of r. The first FTF is disabled by the acknowledgment of m, and the second FTF is disabled by the broadcast of the next message. The action *action* of the first FTF starts the second FTF, and the action *action* of the second FTF enables the broadcast of the next message.

As an example execution of FTF-FTB, consider the following sequence of events

Pseudocode 6.5.1 Fault-Tolerant Forwarding.

```
01: Place $local\_place$ /* The name of the place */
02: Place $last\_sent\_to$ /* The current successor place */
03: Message $msg$ /* The current message being sent */
04: Strategy $strategy = []$ /* The current strategy */
05: Integer $version = 0$ /* The version of the current message being sent */
06: Action $action$ /* The current action to execute at the last non-failed place */
07: Boolean $monitor$ = false /* Whether this place is monitoring a successor or not */

08: def **ftf**(Strategy $s$, Message $m$, Action $a$, Integer $v$):
09:   /* Start FTF if the message version is greater than local version */
10:   if $version < v$:
11:     $msg = m$
12:     $strategy = s$
13:     $version = v$
14:     $action = a$
15:     $last\_sent\_to$ = NULL
16:     $monitor$ = true

17: when (**last_up**($strategy$, $local\_place$)) {
18:   /* No non-failed successor place, execute action */
19:   $monitor = false$
20:   execute $action$
21: } until ($monitor$ == false)

22: when (!**last_up**($strategy$, $local\_place$)) {
23:   /* Check if a message must be sent to the current successor place */
24:   if $last\_sent\_to$ ! = **first_up_after**($strategy$, $local\_place$) && $monitor$:
25:     /* We have a new successor place, send message */
26:     if $last\_sent\_to$ != NULL:
27:       remove $last\_sent\_to$ from $strategy$
28:     $last\_sent\_to$ = **first_up_after**($strategy$, $local\_place$)
29:     if $last\_sent\_to$ != NULL:
30:       send ($message$, $action$, $strategy$, $version$) to $last\_sent\_to$
31: } until ($monitor$ == false)

32: when (**receive**($m$, $a$, $s$, $v$)) {
33:   /* Store message if message version is greater than local version */
34:   if $version < v$:
35:     $strategy = s$
36:     $action = a$
37:     $message = m$
38:     $version = v$
39:     $last\_sent\_to$ = NULL
40:     $monitor$ = true
41: } until (false)
```

when sending a message m where *version* is set to 1, to a strategy consisting of the three places a, b, and c, encoded as $[a, b, c]$. The action x associated with FTF1 starts a new FTF2 with the strategy $[c, b, a]$, the message m' and *version* set to 2. The action y of FTF2 enables a subsequent FTF-FTB to take place.

1. FTF1 is started at a by invoking **ftf**($[a, b, c]$, m, a, 1).
2. Place a has a successor, !**last_up**($[a, b, c]$, a) evaluates to `true`, and `send` (m, x, $[a, b, c]$, 1) to b is invoked.
3. **receive**(m, x, $[a, b, c]$, 1) evaluates to true at b.
4. Place b has a successor, !**last_up**($[a, b, c]$, b) evaluates to `true`, and `send` (m, x, $[a, b, c]$, 1) to c is invoked.
5. **receive**(m, x, $[a, b, c]$, 1) evaluates to true at c.
6. Place c has no successor, **last_up**($[a, b, c]$, c) evaluates to `true`, and `execute` x is invoked.
7. FTF2 is started at c by invoking **ftf**($[c, b, a]$, m', y, 2).
8. Place c has a successor, !**last_up**($[c, b, a]$, c) evaluates to `true`, and `send` (m', y, $[c, b, a]$, 2) to b is invoked.
9. Place b fails.
10. Place c detects that b has failed.
11. Place c has a successor, !**last_up**($[c, b, a]$, c) evaluates to `true` since *last_sent_to* no longer equals place b, and `send` (m', y, $[c, b, a]$, 2) to a is invoked
12. **receive**(m', y, $[c, b, a]$, 2) evaluates to true at a.
13. Place a has no successor, **last_up**($[a, b, c]$, c) evaluates to `true`, and `execute` y is invoked.

An equivalent execution with the original FTB protocol would be like the following:

1. a sends m to b.
2. b sends m to c.
3. c sends the acknowledgment message to b.
4. b fails.
5. a detects the failure of b and resends m to c.
6. c receives m and sends the acknowledgment to a.
7. a receives the acknowledgment for m and the FTB completes.

Thus, instead of c detecting b's failure as with FTF-FTB, a detects b's failure and resends the message to c.

6.5.3 NAP2 Protocol

Assume that action $A(i)$ or $R(i)$ terminates at p according to rule R2. When R2 has completed, R3 is executed with the `change` operation we specified in Section 6.4.5:

R3 When a place p terminates an action, let Q be all the places in $r(i) \cup\ r(i+1)$:

- For $p, q \in Q$, let p be before q in E iff:
  - p and q are both in $r(i)$, and p is before q in $r(i)$.
  - p is in $r(i)$ and q is not in $r(i)$.
- `change` all non-failed places q in $r(i) \cup\ r(i+1)$ `to`:
  - $< r(i+1), i+1 >$ if q in $r(i+1)$.
  - $<>$ if q not in $r(i+1)$.

  `in order` E.

We implement R3 as two subsequent FTFs, called FTF1 and FTF2. In the following, the function **reverse**(r) returns the strategy r in reverse, and the operation $r1 \circ r2$ returns strategy $r1$ with strategy $r2$ appended to it. At the place resulting from executing **first_up_after**($r(i)$, NULL), the following is executed:

$$\text{FTF1: } \mathbf{ftf}(r(i),\ \mathit{state},\ A1,\ i)$$

The action $A1$ is specified as:

$$\text{FTF2: } \mathbf{ftf}(\mathbf{reverse}(r(i+1)) \circ \mathbf{reverse}(r(i) \backslash r(i+1)),\ \mathit{state},\ A2,\ i+1)$$

Action $A2$ is a no-op, and the *state* is specified by a tuple consisting of $< i, r(i), r(i{+}1) >$. We say that places in $r(i)$ enter a *transitional state* during the execution of R3. We encode the transitional state as part of the mobile agent state at a place p: $s(p) =<< r, i >, T >$, where T is true iff the place p is in a transitional state, and false otherwise. We use $s(p).T$ to denote the T component of the mobile agent state at place p. The following steps are performed at a place p when it receives the *state*, following the execution of lines 34-40 of the FTF algorithm:

In1 When p in $r(i+1)$ receives *state* via FTF1, it sets its state to $<< r(i+1), i+1 >, \mathsf{true} >$.

Out1 When p in $r(i{+}1)$ receives *state* via FTF2, it sets its state to $<< r(i{+}1), i{+}1 >, \mathsf{false} >$.

In2 When p in $r(i) \backslash r(i+1)$ receives *state* via FTF1, it sets its state to $<<>, \mathsf{true} >$.

Out2 When p in $r(i) \backslash r(i{+}1)$ receives *state* via FTF2, it sets its state to $<<>, \mathsf{false} >$.

A place is in a transitional state if it does not know that all non-failed places in $r(i)$ are in $<< r(i+1), i+1 >, \mathsf{false} >$. A place in $<< r(i+1), i+1 >, \mathsf{true} >$ is not $i+1$-electable by rule NAP2Elected in Section 6.4.5. We refine property NAP2Elected to NAP2TransElected to reference the transitional state:

NAP2TransElected A place p is i-elected when:

$\wedge s(p).i = i$
$\wedge \forall q \in B(s(p).r, p) : q \in F_p$
$\wedge \neg s(p).T$

A place in $r(i) \backslash r(i+1)$ in $<<>, \mathsf{true} >$ will never become $i+1$-electable because it will never be in $r(i+1)$. Note, though, that any place in $r(i)$ in the transitional state has a role of monitoring for failures in FTF1.

We now prove that FTF1 and FTF2 implements the `change` operation of R3. We do this by showing that the four following properties hold:

A If a place p enters $<< r(i+1), i+1 >, \mathsf{true} >$, then eventually p fails or enters $< r(i+1), i+1 >, \mathsf{false} >$.

B If a place p enters $<<>, \mathsf{true} >$, then eventually p fails or enters $<<>, \mathsf{false} >$.

C If a place p enters $<< r(i+1), i+1 >, \mathsf{true} >$ and does not fail, then eventually:

- All places q in $r(i+1)$: $s(q) =<< r(i+1), i+1) >, \mathsf{false} >$.
- All places q in $r(i) \backslash r(i+1)$: $s(q) =<<>, \mathsf{false} >$.

D Places in $r(i+1) \cup\ r(i)$ change states according to the order specified by E.

To prove that A,B,C and D hold, we need a lemma FTFProgress:

FTFProgress Consider an invocation of FTF with a strategy s and version i, and let p be a place in s. If p receives a message m via this FTF and does not fail, then all places q after p in s either fail or deliver m. Furthermore, the action a associated with the FTF is executed at least once.

Proof. If p is the first place in s, p executes lines 8-16, sets *last_sent_to* to NULL, and sets *monitor* to true. If p is not the first place in s, some other place q sent m to p, so p executes lines 32-40, sets *last_sent_to* to NULL, and sets *monitor* to true. Let x be the number of places after p in the strategy s that do not fail before the end of the execution of the FTF. Either **last_up**(s, p) is true or false. If it ever becomes true, the all places in s after p have failed, and because p does not fail, p executes action a. This is the case for $x = 0$.

Assume that FTFProgress holds for $x = x'$. Consider an execution in which $x = x'+1$ where $x' + 1 > 0$. Thus, **last_up**(s, p) is never true. Let q be the first place after p in s that does not fail before the end of the execution of FTF. We know that some place will send m to q because, even if all places between q and p fail, p will send m to q since p does not fail. And, from the inductive hypothesis, FTFProgress holds for q. Thus, FTFProgress holds for p as well. □

Proof of A. Assume that a place p enters $<< r(i+1), i+1 >, \mathsf{true} >$ and does not fail. By In1, this means that p has received the state $< i, r(i), r(i+1) >$ from its parent q via FTF1. By FTFProgress, all non-failed successors of p will then deliver the state $< i, r(i), r(i+1) >$. Again by FTFProgress, a non-failed successor of p or p itself will execute the action A1 that starts FTF2.

Assume that FTF2 is started by place q in $r(i)$. If q does not fail, then by FTFProgress, all non-failed successors of q in **reverse**$(r(i+1))$ will deliver the state $< i, r(i), r(i+1) >$, and by Out1 enter $<< r(i+1), i+1 >, \mathsf{false} >$. If q fails, q must be a successor of p in $r(i)$, and by FTFProgress, the action that starts FTF2 will be executed again by the first non-failed parent of q. Since p cannot fail, FTF2 is always started, possibly by p, and since p must be contained in **reverse**$(r(i+1))$, p by Out1 enters $<< r(i+1), i+1 >, \mathsf{false} >$. $\square$

Proof of B. The proof is similar to that of A. Assume that a place p enters $<<>, \mathsf{true} >$ and does not fail. By In2, this means that p has received the state $< i, r(i), r(i+1) >$ from its parent q via FTF1. By FTFProgress, all non-failed successors of p will then deliver the state $< i, r(i), r(i+1) >$. Again by FTFProgress, a non-failed successor of p or p itself will execute the action A1 that starts FTF2.

Assume that FTF2 is started by place q in $r(i)$. If q does not fail, then by FTFProgress, all non-failed successors of q in **reverse**$(r(i))$ will deliver the state $< i, r(i), r(i+1) >$, and by Out2 enter $<<>, \mathsf{false} >$. If q fails, q must be a successor of p in $r(i)$, and by FTFProgress, the action that starts FTF2 will be executed again by the first non-failed parent of q. Since p cannot fail, FTF2 is always started, possibly by p, and since p must be contained in **reverse**$(r(i))$, p by Out2 enters $<<>, \mathsf{false} >$. $\square$

Proof of C. Assume that p is the first non-failed place in $r(i)$ that enters $<< r(i+1), i+1 >, \mathsf{true} >$. From FTFProgress, In1, and In2, this means that all non-failed successors of p in $r(i)$ will enter $<< r(i+1), i+1 >, \mathsf{true} >$ or $<<>, \mathsf{true} >$. By A, all non-failed places in $<< r(i+1), i+1 >, \mathsf{true} >$ enter $<< r(i+1), i+1 >, \mathsf{false} >$. Since p cannot fail, FTF2 is always started. Since all non-failed places in $<<>, \mathsf{true} >$ are contained in the strategy for FTF2, by B, all non-failed places in $<<>, \mathsf{true} >$ eventually enter $<<>, \mathsf{false} >$. $\square$

Proof of D. From R3, p is before q in E iff:

1. p and q are both in $r(i)$, and p is before q in $r(i)$.

2. p is in $r(i)$ and q is not in $r(i)$.

All non-failed places in $r(i)$ participate in both FTF1 and FTF2, while the non-failed places in $r(i+1) \backslash r(i)$ only participate in FTF2. We know from C that all these places update their states. The non-failed places in $r(i)$ update their state in the order of $r(i)$, satisfying the first ordering constraint of R3. The non-failed places not in $r(i)$ update their states in FTF2, which is after all the places in $r(i)$ update their states with FTF1. Thus, the places update their states in the order constrained by E. $\square$

The strategy of FTF1 uses the increasing order of $r(i)$ to ensure that property M2, as specified in Section 6.4.4, holds. The strategy of FTF2 uses the reverse order of $r(i+1)$ to ensure that the next stage $i+2$ is not enabled at place p until all non-failed places in $r(i+1) \backslash B(r(i+1), p)$ are in $<< r(i+1), i+1 >, \mathsf{false} >$, as required by the third conjunct of definition NAP2TransElected.

Places in $r(i+1)$ start executing rule R2 when they receive the message in step Out1 in FTF2. When a place p finds itself to be $i+1$-elected, as specified by NAP2TransElected, p starts executing the next action $A(i+1)$.

Terminating Monitoring

The places in $r(i) \backslash r(i+1)$ may receive the messages in step Out2 after the non-failed place p is $i+1$-elected in step Out1, so there may be a situation where $A(i+1)$ terminates faster than the messages have propagated to all places in $r(i) \backslash r(i+1)$. This can lead to FTF1 for the next version $i+2$ to start before FTF2 for version $i+1$ has completed. Since places in $r(i) \backslash r(i+1)$ are in the transitional state $<<>, \mathsf{true} >$, they will not execute a recovery action for i, so NAP2TransElected still holds. However, there is no way for the places in $r(i+1)$ to determine when the places in $r(i) \backslash r(i+1)$ have delivered the message in step Out2 and terminated the monitoring for FTF1. Hence, the places in $r(i+1)$ must monitor the places in $r(i) \backslash r(i+1)$ even after the `change` for $r(i+2)$ has started to ensure that places in $r(i) \backslash r(i+1)$ execute step Out2.

Optimizations

We would like to avoid having places in $r(i+1)$ monitor the places in $r(i) \backslash r(i+1)$ after $A(i+1)$ or $R(i+1)$ has terminated. In addition, the latency of executing the both FTF1 and FTF2 can be significant, so we want to reduce this latency.

For simplicity of exposition, we call the messages sent to the places in $r(i) \backslash r(i+1)$ during FTF2 `stop` messages. From R3 and the ordering required by E, the order which the places in $r(i) \backslash r(i+1)$ receive the `stop` messages is irrelevant. Hence, instead of sending `stop` messages after all the places in $r(i+1)$ have received the message in step Out1, the `stop` messages can be sent in parallel with the execution of FTF2. More specifically, a non-failed place in $r(i+1)$ sends a `stop` message to each place in $r(i) \backslash r(i+1)$ when it receives the message in step Out1 in FTF2. Doing so, however, requires that we change our environmental assumption F1: if a place p in $r(i+1)$ sends a `stop` message that is delivered to all places in $r(i)$ in step Out2, and p subsequently fails before the FTF2 message is sent from p to its successor q in $r(i+1)$, there are no places in $r(i)$ that will ensure that FTF2 makes progress. Thus, F1 must be strengthened to require that at least one place in $r(i) \cap \ r(i+1)$ does not fail:

F1S For all values of $i > 0$: it is always true that some place in $r(i) \cap r(i+1)$ does not fail: $\forall i : i > 0 : (r(i) \cap r(i+1) \backslash F) \neq \emptyset$. This implies that each $r(i) \cap r(i+1)$ contains at least one non-failed place.

Another optimization is to execute rule R2 optimistically. Recall that R2 is executed when all non-failed places in $r(i+1)$ have received the messages in step Out1 and a place is $i+1$-elected. However, we can start R2 optimistically once FTF1 has completed. To implement this approach, the action $A1$ associated with the completion of FTF1 can be extended to send a special `stable` message to the first place p in $r(i+1)$. When, and if, p receives a `stable` message, it executes rule R2. When rule R2 terminates, however, R3 must not be executed before FTF2 has completed sending the messages up to place p. Doing so ensures that the places in $r(i+1)$ are in $< r(i+1), \mathsf{false} >$ when the protocol for starting the next stage begins, as required by NAP2TransElected.

6.6 Implementation

We now describe our implementation of NAP2 using the optimizations specified in the previous section. Over the course of the TACOMA project, focus shifted from the idea of using mobile agents primarily for traditional itinerant style computations to using mobile agents for Distributed Software Management (DiSM) [109, 111]. In DiSM, the key problem is the distribution, installation and upgrading of software in a heterogeneous distributed environment. For instance, rather than itinerant style computations, DiSM mobility is frequently single-hop. We also observed that DiSM applications rarely need the functionality of a complete mobile agent environment, but rather a minimal set of mechanisms to implement various mobility functions [111]. This observation was strengthened by our experience from using code mobility in several DiSM applications, for instance:

- Installing and maintaining intrusion-detection software [111].
- Installing application-specific master and worker code in grid computations [109].

The experience we gained from using mobility as part of DiSM applications motivated the construction of new abstractions specifically tailored for DiSM, where the resulting system is called TOS. We implemented a new version of TOS as part of this dissertation. This version of TOS is similar to the Java version described in [111], but written in the Python programming language.

Besides our experience with DiSM, the focus towards providing building blocks for code mobility rather than complete mobile agent environments has also been employed by other projects investigating distributed system and software management [22, 138].

We also considered implementing NAP2 using the wrapper concept introduced in TACOMA version 2.0 [171]. The main benefit of wrappers is that they are transparent to the wrapped agent. This way, a regular TACOMA agent can be provided with fault-tolerance without having to be written specifically with fault-tolerance in mind. However, transparently wrapped agents would be unable to specify recovery actions and manage failover lists, which are important features in NAP2.

6.6.1 TOS

TOS is structured as a library that supports asynchronous message passing among threads of execution, inspired by the *actor* model [3]. TOS provides a service through an *extension*, which is a thread of control and a communication channel. The communication channel can be used to exchange messages with other extensions.

The set of TOS extensions within a process comprises a TOS server, comparable to a place in mobile agent terminology. Extensions running within the same TOS server are referred to as local extensions. In terms of control flow, TOS extensions can communicate with each other in a hierarchical or pipelined fashion with message passing, or by using a shared-state Python namespace.

Communication Channels

The TOS communication channels use in-memory queues for communication between local extensions and TCP sockets for communication between extensions on different TOS servers. Local and remote communication between two extensions are FIFO ordered. Each pair of extensions communicating remotely use a dedicated TCP connection.

A TOS communication channel between two places s and r is established when the first message is sent from s to r. Subsequent messages exchanged by s and r reuse the established channel. Establishing a channel in TOS involves setting up a TCP connection with `connect` from s and `accept` from r, followed by the following symmetric authentication protocol between s and r: s sends a byte sequence consisting of 256 random bytes followed by the string `gobbledegook` to r, where the byte sequence is encrypted with a private key known to both s and r. r reads the 256 random bytes and checks that the string after the random sequence matches `gobbledegook`. s then sends the encrypted message using the established channel to r. This simple authentication protocol helps keeping malicious users from exploiting TOS to gain access to resources that they are not privileged to access, and gives us a performance baseline for the cost of providing authentication. A more realistic implementation for authenticating clients and delegating access to resources would probably be based on an established standard such as OAuth [77].

6.6.2 Agent Control Flow

An action in NAP2 terminates with the operation **move**:

$$\textbf{move}(\textit{action}@\textit{place})$$

Which regular *action* to run next as well as the *place* to execute the action on is specified as arguments to the **move** operation.

The **move** operation is also used to implement the operations **checkpoint** and **terminate**, as specified in Section 6.1. The **terminate** operation is implemented by a **move** that is given `NULL` as parameter. In this case, the termination protocol specified in Section 6.6.8 is executed.

If the *place* specified by the **move** operation is the place where the agent is currently executing, **move** behaves the same way as the **checkpoint** operation. More specifically, **move** will cause the current action to terminate, the protocol for updating the places in the failover list to be executed, and the next action started at the same place.

Operations	Operations with **move**
move$(a_1@p)$	**move**$(a_1@p)$
checkpoint(a_1)	**move**$(a_1@p_{local})$
terminate()	**move**(NULL)

Table 6.1: Mobile agent operations modeled with the **move** operation.

When *place* is specified by the **move** operation, *place* is explicitly inserted as the first member of the failover list. If *place* is not specified by **move**, the *place* is the first member specified in the failover list. Table 6.1 summarizes how **move** specifies traditional **move**, **checkpoint** and **terminate**.

As an example of a NAP2 computation, Figure 6.3 depicts a mobile agent computation using the variations of the **move** operation, beginning with action a_1 at place p_1. The second action a_2 starts on place p_2 after the first action terminates. When the second action terminates, action a_3 executes at place p_3, followed by action a_4 on place p_3. The computation ends when a_4 terminates on p_4.

6.6.3 Specifying Actions

Pseudocode 6.6.1 Example action and recovery action.

```
def action1(failed=false):
    if not failed:
        print "Hello, world!"
    else:
        print "Failed to say hello!"
```

The **move** operation terminates an action and specifies which regular action to run next, but does not explicitly specify which recovery action correspond to which regular action. Rather than managing actions and recovery actions in two separate TACOMA folders, as we specified for an earlier version of NAP [89], NAP2 actions are written as a function that accepts a boolean parameter `failed`. When invoked, `failed` is set to `false` for regular actions and `true` for recovery actions. Hence, when an agent specifies which regular action to run next, the next recovery action is implicitly specified. Pseudocode 6.6.1 gives an example action and recovery action structured this way. If the regular action and recovery action are identical, then the `if...else` statements can be removed.

6.6.4 State Management

The TOS version used to implement NAP2 is written in the Python programming language, and we decided that NAP2 agents must also be written in Python. For DiSM

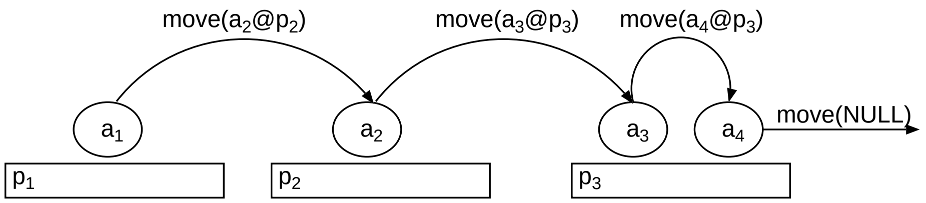


Figure 6.3: Example mobile agent control flow with **move** operation.

Attribute name	Description
id	A globally unique number identifying the agent
rg	The failover list for the next action
rg0	The failover list for the current action
version	The version of the current failover list
code	The source code (Python class) for the agent
next	The name of the next action to be executed

Table 6.2: Message attributes used by NAP2.

tasks, we found that the use of a single high-level language is sufficient for most tasks. Other required functionalities are instead shipped with the agent and executed using mechanisms such as the `system()` system-call in Unix [111].

NAP2 agents are Python class instances and the actions must be methods implemented in the agent class. Python class instances have attributes associated with them, and these attributes determine the state that is shipped along with the agent upon migration. This is similar to how *folders* persist through the `meet` operation in the original TACOMA system. Instance attributes are restricted to those types supported by the Python `marshal` module, which we use for serialization. This include the most common types such as integers, floats, strings, lists, tuples, and dictionaries.

Some state management is also performed outside the agent code body, for instance serializing the agent attributes, transferring the serialized agent to the right place and instantiation of the agent class. This part of the state management is implemented by the `nap` extension, as described in Section 6.6.5.

NAP Specific State Management

NAP2 and TOS require the agent to host a number of NAP2-specific attributes in its state. The particular attributes are listed in Table 6.2, and are used by NAP2 when executing the protocol. All attributes can be accessed and changed by an agent during execution, but mobile agent applications normally only update the *rg* attribute.

The attribute *id* is a globally unique agent identifier used to separate instances of agents from each other. The *id* is generated when the agent is created and must not be changed during the computation. The *rg* attribute is the failover list for the next action,

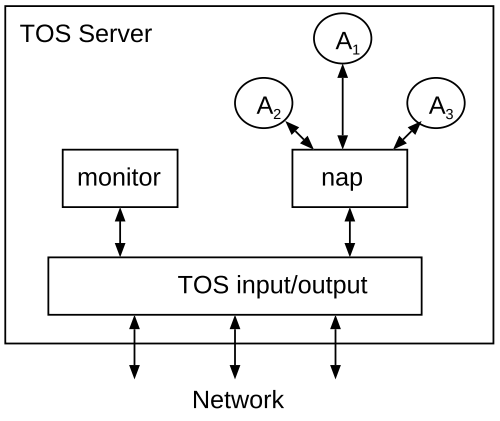


Figure 6.4: NAP on TOS.

and the *rg0* attribute the failover list for the currently executing action. The *version* attribute is an integer that represents the version of the failover list in *rg0*. *code* attribute holds the source code for the agent, and the *next* attribute holds the method name of the action to be executed when the current action terminates. The *next* attribute is not set explicitly by an action, this is done implicitly when an action is terminated with the **move** operation.

6.6.5 NAP2 Extensions

The NAP2 runtime support is implemented by two co-operating TOS extensions, the `monitor` extension for failure detection and the `nap` extension that controls the execution of agents as well as the orchestration of the NAP2 protocol. Separating failure detection from the rest of the protocol makes it simpler to adapt to alternative failure detectors. Figure 6.4 shows a TOS server running agents A_1, A_2 and A_3 within `nap`, and uses TOS messages to communicate with the other TOS servers and the `monitor` extension.

`monitor` Extension

The `monitor` extension provides a fail-stop failure detection mechanism. Local extensions can subscribe to be notified by the `monitor` extension when a specified place fails. Failure detection is performed by sending `ping` messages from the `monitor` extension to a set of destination places on remote TOS servers. Destination places reply with `pong` messages. If a target place fails to respond with a `pong` message within a configured amount of time, a notification message is sent to the local extensions that have subscribed to failure notifications. NAP2 uses this extension to detect failures during execution of the FTFs and detecting whether the place executing an agent has failed.

The `monitor` extension also allows monitoring so-called *activities* at destination places. Activities enable detecting that an individual agent fails without the place or the processor that executes on the place having failed. An activity represents a fine-grained resource within a place, such as a thread or method call. For NAP2, an agent denotes an activity represented by the agent identifier in the *id* attribute. When an activity is flagged as failed within a place, the subsequent `pong` message from the place includes the activity (for `nap`, the agent *id* attribute). When receiving the `pong` message, a place extracts the failed activities (for `nap`, agent *id* attributes) and notifies the local extensions that have subscribed to failure notifications for the destination place with a `fail` message. The activities that failed are contained in the `fail` message.

The timeout value used by the `monitor` extension can be configured, and by default the timeout is set to 30 seconds.

`nap` **Extension**

All places involved in a NAP computation must run the NAP extension. All inter-place communication between `nap` extensions is by means of TOS messages. All these TOS messages consist of the agent attributes that comprise the mobile agent state, as well as a `type` attribute that identifies the type of the message. The `nap` extensions handles seven different message types: `start`, `fail`, `update`, `ack`, `stop`, `turn`, and `stable`.

The `start` message is sent locally by the currently elected place when an ongoing protocol execution has completed. When receiving the `start` message, the `nap` extension creates an agent instance based on the value of the *code* attribute and determines which action (i.e., Python method) to execute from the *next* attribute of the message. Before the action starts executing, the `nap` extension stores the failover list in *rg* in the *rg0* agent attribute. During execution of the current action, the mobile agent specifies the next failover list in the *rg* attribute. The agent code is subsequently enqueued on a thread pool that performs the actual execution of the action. The thread that runs the action returns the operation to be executed next when the action terminates. If the returned operation is a **terminate**, then the protocol specified in Section 6.6.8 is started. If the returned operation is not a **move**, the agent is assumed to have failed. The activity representing the agent is flagged as failed, and the `monitor` extension of the next place in the failover list will eventually detect the activity or place as failed. This causes a `fail` message to be sent to the local `nap` extension with the activity that failed. The `nap` extension then executes the recovery action or resubmits messages as part of the FTF execution.

If an action instead terminates with a **move**, then all agent attributes are extracted by accessing the `__dict__` attribute of the agent class instance and put in a message of type `update`. If the **move** specifies a place, then this place is put first in *rg*. The `update` message is then sent to the first place in *rg0* of the terminating agent. Sending the `update` message to the first place of *rg0* thus starts FTF1 of the NAP2 protocol. When all places of *rg0* have received the `update` message, FTF2 is started by sending a `turn` message from the last place in *rg0* to the last place of *rg*. The last place in *rg0* also sends the `stable` message to the first place in *rg* to allow starting the next action optimistically. When the last place in *rg* receives the `turn` message, it sends an `ack` message to its preceding place in *rg*. When a place receives the `ack` or `stable` message, it sends `stop` messages to all places *rg0*. When a place in *rg* has received the

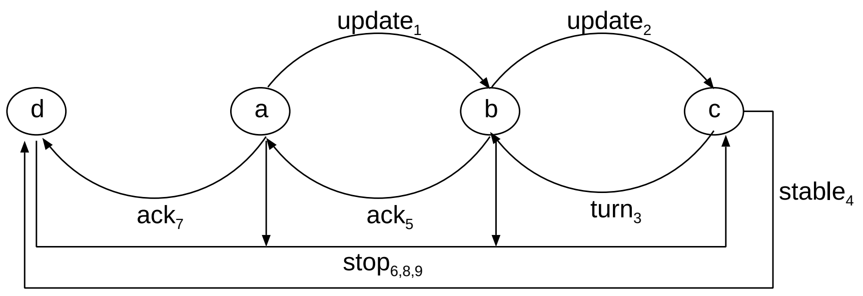


Figure 6.5: Message flow in NAP2.

`ack` message or `stable` message, whichever comes first, rule R2 is executed to determine whether it is elected. If a place in *rg* finds itself to be elected, a `start` message is sent the local `nap` extension.

Figure 6.5 shows an example of an invocation of the NAP2 protocol when *rg0* is $[a, b, c]$ and *rg* is $[d, a, b, c]$. The subscript of the messages depicts the message ordering. First, a sends an `update` message to b, which in turn sends an `update` to c. c has no successor so it sends a `turn` message to b and a `stable` message to d. Upon receiving the `turn` message, b then sends a `stop` message to c and an `ack` message to a. a sends an `ack` message to d and a `stop` message to c, whereas d has no previous place and sends a `stop` message to c. c, upon receiving a `stop` message, terminates the monitoring it previously did for b.

6.6.6 Starting Agents

When an agent has no previous failover list, there is no need for the `update` messages. Instead, the agent can be sent with a `turn` message to the last place in $r(1)$. This will cause `ack` messages to propagate up to the first place of $r(1)$, and the elected place starts executing the agent. This procedure, which is used to bootstrap a NAP2 agent into the initial state specified by rule NAP2 rule R1, implements the **init** operation. The **init** operation takes three parameters, corresponding to the attributes in Table 6.2:

$$\textbf{init}(\textit{agent}, \textit{failover}, \textit{action})$$

Here, *agent* is the agent source code, $failover$ the failover list for the first action, and *action* the name of the first action to execute. The agent source code in *agent* is stored in the *code* attribute, and the failover list in $failover$ is stored in the *rg* attribute. The *id* attribute is generated by the **init** operation to ensure that the agent identifier is unique, and the $version$ attribute is set to 1.

Starting Multiple Agents

When we implemented TOS agents for distributed software management, one of the recurring scenarios was to use a fan-out strategy for distributing the agents. In this scenario, the initiating agent typically **init** n agents, causing n invocations of the bootstrapping protocol specified in the section above. The **init** operation does not terminate the action and is non-blocking by virtue of the asynchronous message passing in TOS, so any number of **init** operations can be done within the scope of a single NAP2 action.

6.6.7 Deterministic Failures

Although not covered by the system model in Section 6.1.1, in practice there will be situations, for instance, programming bugs, where a recovery action $R(i)$ will fail deterministically and all places that attempt to execute $R(i)$ will fail. An approach to handle this problem is to **move** the agent to a specific place, which we call the *rally point*. The identity of the rally point is specified in the agent attribute *rally_point*. When a rally point is specified and there is only a single non-failed place p in $r(i)$ that finds itself about to execute $R(i)$, then place p will not execute $R(i)$ but instead execute a **move** to the place specified in the *rally_point* agent attribute. Doing so can aid identifying the cause of the deterministic failure by inspecting the mobile agent state at the rally point place.

6.6.8 Terminating Agents

When no more actions are specified or **terminate** is issued by the computation, the NAP2 protocol for this agent also terminates. Although the FTB protocol that NAP2 is based on cannot terminate [160], orchestrating termination of NAP2 is not complicated. Assume that the operation **exit** is the command that instructs a place to terminate execution for the corresponding mobile agent, and let the last user-defined action be:

FTA_ω: **action** A_ω **recovery** R_ω

FTA_ω is replaced by the following two actions:

action { A_ω; **checkpoint**} **recovery** R_ω;
action exit recovery exit;

Hence, after **checkpoint** terminates, all places in the failover list have their subsequent action and recovery action set to **exit**. When a place executes **exit**, it terminates executing NAP2 for the agent, resulting in the activity that specifies the agent being reported as failed with a subsequent failure detection by the next place. The election protocol in NAP2 chooses a place to execute the recovery action. The agent that executes the recovery action then terminates, causing the activity specifying that agent as failed to the next place where another failure detection is done and another place starts executing the recovery action. This sequence of actions continue until the last place in the failover list has executed **exit**.

Note that this behavior is similar to a deterministic failure of a recovery action. Thus, when a rally point is specified, the last place will **move** the agent to the place specified in the *rally_point* attribute of the agent. Hence, all executions end up at the rally point at termination. The reason for termination can be recorded as an attribute in the mobile agent.

6.7 Network Failures

The NAP2 protocol assumes the use of a perfect failure detector and that processes execute according to the fail-stop model. Results from the work of Christian and Fetzer [40] indicate that local area networks perform synchronously most of the time. The networks fluctuate between long stable periods where timing assumptions can be made and short unstable periods where timing assumptions no longer hold. In their work, the ratio between a stable versus an unstable period is measured to 341:1, which means that a perfect failure detector can work correctly most of the time within a local area network. However, in an asynchronous environment where places may execute slowly and network packets are lost, failure detection may cause false positives. For NAP2, false positives cause redundant actions to be executed, violating the exactly-once property of a stage. More specifically, this means that the places q in the second conjunct of the NAP2Elected property specified in Section 6.4.5 may be contained in F_p without having failed.

From the previous chapters, we have observed three kinds of network failures: i) partitions where two or more cliques cannot communicate with each other, ii) asymmetric communication, and iii) non-transitive communication. We now investigate what happens in the NAP2 implementation specified in Section 6.5 when such failures occur.

In the following sections, r is a failover list.

6.7.1 Partitions

A network partition divides the set of places in the failover list into two or more cliques that cannot communicate with one another. In this case, packets going between cliques will be lost, and the NAP2 failure detector in the `monitor` extension will eventually detect this.

Assume that the list of places in r are partitioned into sets $c_1, c_2, ..., c_{n-1}, c_n$, where n is the number of cliques. Then, all c_n, except the clique containing the head of r will execute a recovery action. A worst case for redundant recovery action executions is when each set c contains only one place (i.e., n is equal to the length of r). Then, the number of recovery actions for r will be $|r| - 1$.

6.7.2 Asymmetric Communication

When there is asymmetric communication, a place x is able to communicate with a place y, but y is unable to communicate properly with x.

The failure detector in the `monitor` extension in NAP2 works by having a thread periodically send a *ping* message to the place that is being monitored. Upon receiving a *ping* message, a place sends back a *pong* message to the place that sent the *ping* message. If a *pong* message is not received within a specific timeout value, in our case 30 seconds, the place being monitored is declared as failed.

If x monitors place y, then y may send a *pong* message, but x will not receive it and y is thus declared as failed by x. Similar, if y monitors x, the initial *ping* message is not received by x and thus no corresponding *pong* message is sent. Hence, x will detect y as failed, and y will detect x as failed, with the result being similar to that of x and y being partitioned away from each other. A worst case when the NAP2 failure detector is used occurs when all places in r have an asymmetric communication link with all other places. Given the symmetric failure detection property of the failure detector in NAP2, this scenario is similar to the scenario of partitioned cliques with a single member and results in a similar number of recovery actions being executed.

Asymmetric communication can also cause disagreement on the members of the failover list. Assume we have $r = [x, y, z]$, where z cannot communicate with y. In this case, y will detect z as failed and find itself to be the tail of r and send an `ack` message to x. Likewise, z will detect y as failed, and send an `ack` message to x. The problem now is that both y and z believe they are the tail of r and monitor x for failures and should x fail, the recovery action is executed by both y and z.

6.7.3 Non-Transitive Communication

Non-transitive communication happens when a place x can communicate with a place z only through place y. For instance, if $r = [x, y, z]$ and y fails, then x will fail to communicate with z and thus assume that both y and z failed although only y failed.

Consider the scenario where all communication goes through a single place, and this place is either at the head or the tail of r. Assume there are n places in r, $[p_1, p_2, \ldots, p_{n-1}, p_n]$, and that all $n-1$ places $[p_1, p_2, \ldots, p_{n-1}]$ communicate through p_n. This means that places p_1 through p_{n-2}, will observe all other places as failed, similar to the case where all places are partitioned away from each other. Place p_{n-1} will be able to communicate with p_n, which leads to one fewer execution of the recovery action compared to the worst case of all places being partitioned away from each other. If the communication graph later changes so that p_n can only communicate with p_{n-1} through a place in $[p_1, \ldots, p_{n-2}]$, then p_n will detect p_{n-1} as failed.

6.7.4 Observations

From the simulation results based on the RON traces in Section 3.4.2 (Table 3.1), we observed that when network partitioning occurs, the number of connected components are never more than two. This means that the worst case for partitions in the previous analysis is unlikely to occur frequently. However, we observed that the amount of non-transitive communication is significant, and thus more likely to trigger similar worst-case behavior.

In Section 5.6 we described that there are generally two approaches to avoid faulty communication links, i) using a static communication link setup, and ii) using an overlay network. Given the complexity and overhead of maintaining an overlay network as part of the mobile agent runtime, an approach inspired by static link configuration seems more appropriate for NAP. If failover lists are adapted to the underlying network topology during the computation, implying failover lists are *network aware*, the advantage of static link configurations can be gained for mobile agents.

Recall from Section 3.3.1 that non-transitive communication can occur when communication spans more than two ASes. Determining whether a failover list spans more than two ASes can be difficult, since the topology view depends on the mechanism used to establish the view. For instance, Mahadevan et. al. [117] discovered that the topology obtained using WHOIS was significantly different from the topologies obtained with `traceroute` and BGP. However, a conservative approximation of an AS can be done by discerning on local area networks. Thus, if the places are organized in such a way that a failover list never spans more than two local area networks, then the computation will never span more than two ASes and the probability of non-transitive communication is small.

Failover lists spanning two or more local area networks are also vulnerable to partitions. The vulnerability to partitions can be reduced by minimizing the time failover lists span two or more local area networks. In protocols with transparent backup management, such as first version of NAP [89], this requires the mobile agent to issue a sequence of $f + 1$ **move** operations each time it enters a new local area network to add at least f places within that local area network. Allowing multiple places to be added and removed with fewer than $f + 1$ **move** invocations may reduce the time a failover list spans more than one local area network, and thus reduce the exposure to network failures.

In the next chapter, we analyze the costs of adapting failover lists to avoid network failures.

6.8 Summary

We started this chapter by specifying a mobile agent fault-tolerance protocol called the Norwegian Army Protocol (NAP). NAP is based on a primary-backup approach and updates state to the backups at fixed points during the computation. The set of backups can be changed during the computation to accommodate resource changes in the environment. We presented the derivation of a specification of the NAP protocol, and continued with specifying how to map the derived protocol to an implementation. The chapter then presented an implementation of NAP using the TOS mobile agent platform, and ended with how NAP can be made robust against the network failures we identified in Chapter 3.3.1.

Chapter 7

NAP2 Performance Evaluation

In the previous chapter, we derived a specification for NAP2, and described an implementation of the protocol. In this chapter, the performance of the resulting NAP2 implementation is evaluated. The performance evaluation has two main purposes. First, evaluate the performance hypothesis we proposed in Section 1.3. Next, serve as basis for the discussion in Chapter 8 where we consider alternative implementations.

7.1 Purpose

The main purpose of this chapter is to evaluate the performance of our NAP2 implementation. By performance, we mean the *latency* of executing a mobile agent computation. The measured latency will be used to answer two questions.

The first question we ask is: *what is the latency of running a mobile agent application in TOS with NAP2?* To answer this question, the latency of sending and receiving a message from one computer to another with TOS must be measured. We then need to measure the latency of executing the **move** operation without NAP2, and compare this with the latency of executing **move** with NAP2. The latency of NAP2 is influenced by many parameters, such as the number of places in the failover list, which we vary.

One of the key design features of NAP2 is flexible reconfiguration of the failover list, and this allows moving a set of backups from one local area network to another to reduce exposure to communication failures. With transparent backup management, as specified in Section 6.3.3, similar reconfigurations can be done through use of multiple **move** operations. The next question we ask is: *is the latency of changing between a failover list R to a disjoint failover list R' larger with transparent backup management?* To answer this question, the latency of adding and removing multiple places to the failover list in NAP2 is measured and compared to the latency of doing a comparable number of **move** operations with transparent backup management.

The results will either confirm or falsify the hypothesis given in Section 1.3, where we propose that network aware mobile agent fault-tolerance will perform better than transparent mobile agent fault-tolerance in this case.

7.2 Experiment Procedure

In order to answer the two questions in the previous section, experiments must be performed. The purpose of this section is to establish that the procedure for performing these experiments is reasonable.

7.2.1 Test Environment

Our experience with NAP2 applications like the License Checker in Section 6.3.2 has so far shown that between 2 and 5 backups provide sufficient fault-tolerance. Some of the experiments, for instance the last in this chapter, are simpler to measure with more computers. Thus, our test environment consists of 10 identically configured Dell Precision WorkStation 360 PCs with P4 3GHz CPUs, 1GB RAM and 120GB disks, connected by a 100Mbit Ethernet. The computers run Red Hat Enterprise Linux Client release 5.0. NAP2 is implemented in the Python programming language, and we use Python 2.5.1 to execute TOS. Each place, as specified in Section 6.1, is a computer running a single TOS deployment that hosts a single instance of the NAP2 extensions.

Although the latency of NAP2 is expected to be largely dominated by network latency, some interference can be assumed from the Python interpreter. For instance, Python internally schedules threads to serialize execution of bytecodes among multiple threads. This means that only one thread executes bytecode at a time and the cost of scheduling threads increases with the number of threads [175]. The context switching is done either when a particular number of bytecodes have executed for a thread, or the thread explicitly yields control (e.g., before performing a blocking system call such as `sleep`). In addition, Python uses reference counting to manage memory, coupled with a generational garbage collector for handling circular references. The garbage collector runs asynchronously and freezes the execution of bytecodes when executed.

To avoid buffering in TCP to affect the latency of communicating messages, the Nagle algorithm has been disabled in TOS. The experiments are performed on multiuser computers, and users logging into the computers may affect the amount of network bandwidth and induce processor load on the computer. To limit the impact of the multiuser environment, we perform all experiments during the night, where we expect the network and computers to be lightly loaded. The same applies to reducing the impact of memory pressure. Memory pressure occurs if another users' application running on the computer forces our TOS deployment to be swapped out. Since the TOS deployment requires little memory, we assume that unless another user's application uses a lot a memory on the computer, memory pressure should not be a problem. None of our experiments directly rely on the performance of stable storage such as disks. A subtlety here is that the Python interpreter dynamically loads modules from disk when the `import` statement is issued in the code. To avoid this from influencing our experiments, we have removed all `import` statements from the critical paths of TOS and the `nap` and `monitor` extensions.

Although carefully managing our test environment this way can reduce the probability of influence from parameters such as users logging into the test computer, we cannot completely disregard them. Thus, we use statistical methods to remove samples from our results in cases where the test environment has a strong bias on the samples.

7.2.2 Samples

Samples are acquired by recording the time when the experiment begins and ends, and the difference then denotes the latency of the system. We obtain the time using the `gettimeofday` system call with the `time` function of the `time` Python module. We measured the performance of calling `gettimeofday` with the `time` function in Python. The experiment revealed that the mean cost was 9 microseconds for two `gettimeofday` calls. This cost is subtracted from all samples.

All experiments in this chapter are executed 100 times, yielding 100 samples, and based on these samples we calculate the mean ($\overline{X}$) and standard deviation (SD) for the samples. The standard deviation is given by the square root of the variance, where N is the number of samples and $x_1, \ldots, x_N$ denotes the samples:

$$\overline{X} = \tfrac{1}{N} \textstyle\sum_{i=1}^{N} x_i \text{ and } SD = \sqrt{\tfrac{1}{N} \textstyle\sum_{i=1}^{N} (x_i - \overline{x})^2}$$

We also calculate the relative standard deviation (RSD) and coefficient of variation (CV), as given by the following formulas [174]:

$$RSD = CV * 100 \text{ and } CV = \tfrac{SD}{\overline{X}}$$

During the course of the experiments we experienced that the RSD factor was sometimes large. By sorting the list of samples for such experiments, we quickly discovered that some samples deviated significantly from the calculated mean. Such values are referred to as *outliers* [174]. We experienced that outliers for example occurred when the garbage collection in Python was executed. The procedure we use for removing the impact of outliers is known as Grubbs test (or extreme studentized deviate) [174]. The approach of Grubbs test is to rank samples by increasing value and calculate the mean and the standard deviation for the samples. The lowest and highest ranked samples (X_{low} and X_{high}, respectively) are then evaluated against the mean and standard deviation like this:

$$Z_{low} = \tfrac{(\overline{X} - X_{low})}{SD} \text{ and } Z_{high} = \tfrac{(X_{high} - \overline{X})}{SD}$$

Whether Z_{low} or Z_{high} is an outlier then depends on whether they exceed a threshold value that is determined by the confidence level. Hence, Grubbs test removes outliers for a given level of confidence, and we use 99.9% confidence for our experiments. We report the number of samples in the sample sets with the number of outliers removed. Unless stated otherwise, all figures in this chapter plot the mean values for each experiment with error bars based on the highest and lowest sample.

7.3 NAP2 Latency

The purpose of the first set of experiments is to investigate the cost of running NAP2. We start by determining the latency of communicating a message in TOS, both when establishing a new communication channel and when reusing an existing channel. We then measure the cost of executing mobile agents in TOS with **move** without NAP2 being enabled.

Based on the results of the experiments above, we develop three estimate functions α, β and γ, that predict the performance of NAP2 as a function of changes when going from the failover list $r(i)$ to $r(i+1)$ where i is the version of the list.

We then measure the actual latency of running NAP2 and compare the results with the results of the estimators for the same experiments. If the measured results are close to the estimators, then the estimators can be used to calculate the expected latency of executing the protocol for a given computation.

7.3.1 TOS Performance

TOS communication between TOS servers takes place on channels on top of TCP sockets. Communication is done with messages, where messages are Python dictionaries serialized with the `marshal` Python module. The `nap` TOS extension discussed in Section 6.5, sends all mobile agent state using a TOS message.

Message communication in NAP2 both establish new channels (e.g., when adding new places to the failover list) and reuse channels (e.g., when executing **checkpoint**), so we measure the latency of communicating a message in both cases.

The experiment uses an extension called `pingpong`. This extension handles messages with two attributes, *hops* and *path*. The *path* attribute states the sequence of places that the message should be sent along. When the first message arrives at the first place in *path*, the current time is recorded in the message. The *hops* attribute is decremented each time a messages arrives at the `pingpong` extension at a place. When *hops* reaches zero, the difference between the current time and the recorded time is calculated. For example, to sample the time it takes to do 100 message exchanges between host p_1 and p_2, *hops* would be 100 and *path* would be $[p_1, p_2]$.

We use 10 places to run the experiments, configure *path* to contain all places, and set *hops* to 10. The time difference between the start and the end of the experiment is thus calculated by the same place. Since the sample gives the value of performing 10 functionally identical message exchanges, we find the value of performing a single message exchange by dividing the sample value by 10.

The experiment involves two scenarios. The first scenario of the experiment measures the latency of sending and receiving a message between two places when the TOS communication channel has not been set up. The purpose of this experiment is to establish whether the latency of communicating a message increases linearly with the size of the message. Hence, we vary the additional message payload in from 256 bytes up to 512 kilobytes, which covers the payload requirements of our mobile agent applications so far. Payloads of 256 and 512 bytes may seem too small to have any practical use for mobile agents. However, the high-level Python programming language enables expressing useful computations at this size. Conversely, our experience with TOS written in Java was that similar computations required more payload due to larger serialized state [111].

P	0B	256B	512B	1	2	4	8	16	32	64	128	256	512
M (N)	1.8	1.9	2.0	2.1	2.2	2.4	2.9	3.7	5.3	8.6	15.2	28.3	55.5
S (N)	95	96	96	94	96	96	96	95	96	96	93	96	100
M (E)	0.6	0.6	0.7	0.9	0.9	1.1	1.7	2.6	4.2	7.4	14.1	27.0	53.9
S (E)	100	100	100	99	96	100	100	100	100	100	100	99	98
D	1.2	1.2	1.2	1.2	1.2	1.2	1.1	1.1	1.1	1.1	1.2	1.2	1.6

Table 7.1: Latency results in milliseconds for established (E) and non-established (N) channels.

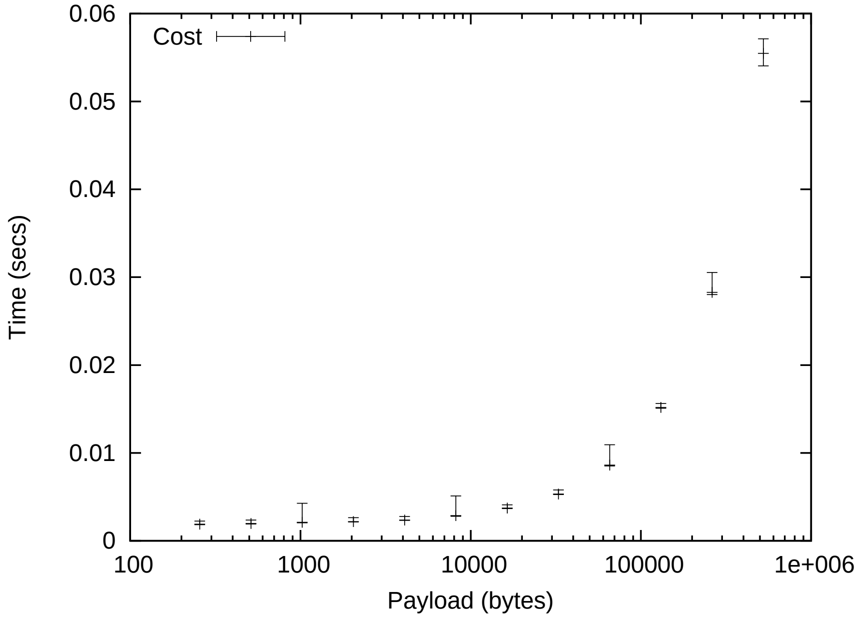
The results of the first scenario of the experiment are shown in Figure 7.1a. The payload of the message on the x-axis is shown in log-scale. The results show that the latency of communicating a message increases linearly with the size of the message, also for large messages. This indicates that TOS latency scales linearly as a function of message size.

The second scenario of the experiment measures the latency of sending and receiving a message between two places on an already established TOS communication channel. The purpose of this experiment is to measure the latency of communicating a message from one extension to another, and also calculate the cost of setting up a TOS communication channel. The cost of setting up a communication channel can be calculated by subtracting the result samples of this scenario from the result samples of the first scenario.

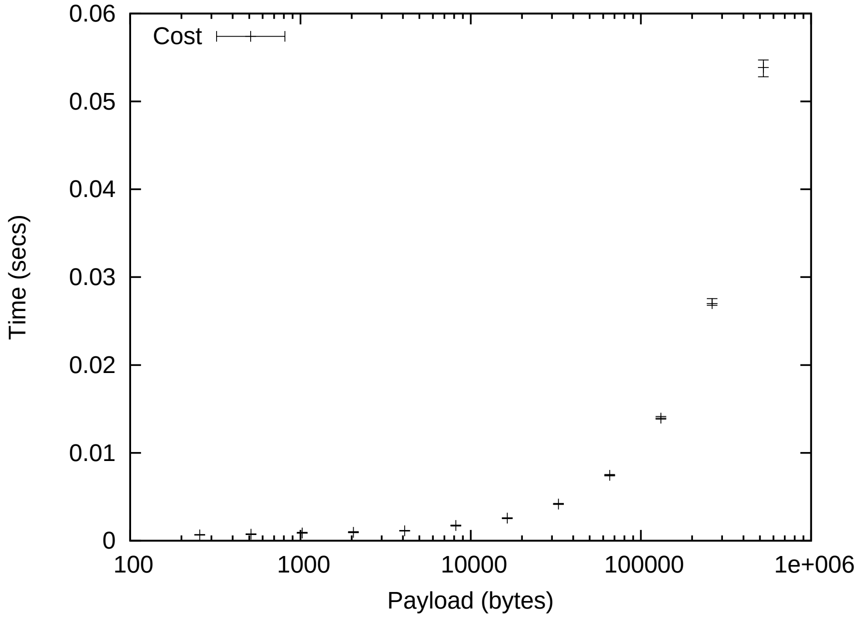
We expect the cost of setting up a communication channel to be similar regardless of payloads, since the communication setup procedure in TOS transfers the same amount of data for all payloads. Figure 7.1b shows the result of communicating a message on an established channel, and Figure 7.2 plots the difference between the mean latency of a message with and without the channel setup cost. The payload of the message on the x-axis is shown in log-scale.

As we expected, Figure 7.2 shows that the latency of establishing the channel is independent of the message payload. There is a marginally higher latency for the payloads larger than 64 kilobytes, mainly due to operations on larger amounts of data in the Python interpreter when setting up initial message used for channel authentication. We can thus infer that the cost of setting up the TOS communication channel is between 1.1 and 1.6 milliseconds.

The results for both scenarios of the experiment are summarized in Table 7.1, where all time values are given in milliseconds. The P row specifies the payload, the M rows the mean latency values for established (E) and non-established (N) channels, and the S rows the number of samples with outliers removed for established (E) and non-established (N) channels. The D row gives the difference between the mean for the latency with and without communication channel setup latency. Note that the values in the D row were calculated with greater precision than given in the table and rounded to the same precision.



(a) Latency results on non-established channels



(b) Latency results on established channels

Figure 7.1: Latency results of communicating a TOS message with and without communication channel setup.

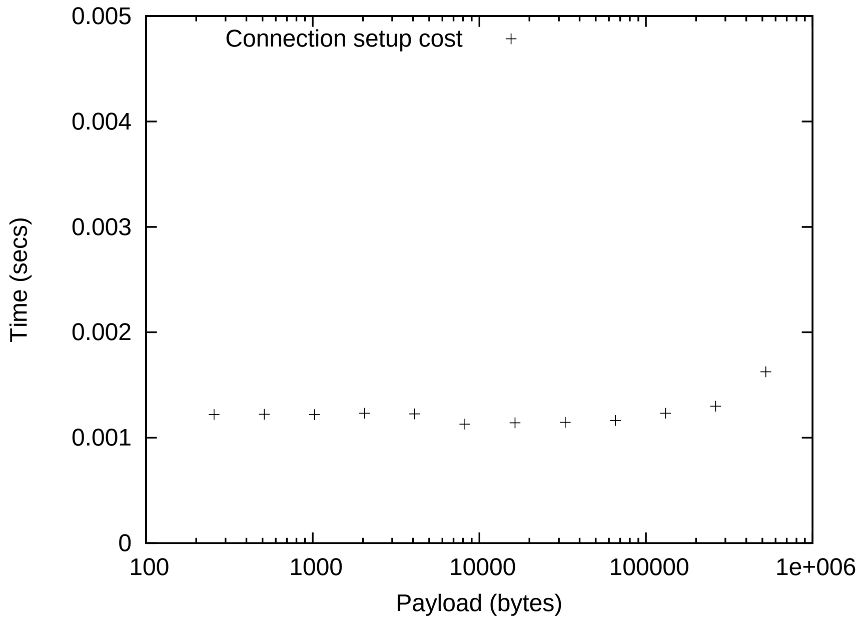


Figure 7.2: Mean communication channel setup cost.

Observations on TOS Performance

For 256 byte messages, communicating a message between two TOS extensions takes 0.6 milliseconds when the channel is established. Thus, 1666 messages can be transmitted per second, yielding a bandwidth of 426 kilobytes per second with 256 byte messages. Similar, with 512 kilobyte messages, 53.9 milliseconds were used, yielding a bandwidth of 9499 kilobytes per second. This means that TOS is able to utilize most of the 100 megabit bandwidth with 512 kilobyte messages. However, smaller messages are not able to fully utilize the bandwidth, which may be caused by our choice to disable the Nagle algorithm for TOS sockets. Another reason TOS does not utilize more bandwidth for smaller messages is the latency of the Python interpreter when processing incoming and outgoing messages to and from TOS extensions.

7.3.2 Mobile Agent Performance

The purpose of this experiment is to measure the latency of migrating a mobile agent in TOS. We use an extension called `next` that implements the **move** operation used by NAP2, but without invoking the NAP2 protocol for changing the failover list. More specifically, the `next` extension performs the following steps during execution at a place:

- Extract the mobile agent source code from the TOS message and instantiate the mobile agent class.
- Enqueue the mobile agent instance on a thread pool where a thread executes the action of the mobile agent.

P	0B	256B	512B	1	2	4	8	16	32	64	128	256	512
M	2.4	2.5	2.6	2.7	2.8	3.4	3.9	4.8	6.4	9.6	16.1	30.3	58.1
S	90	94	91	97	97	100	99	99	99	95	99	98	98
D	0.6	0.6	0.6	0.6	0.6	1.0	1.0	0.9	1.1	1.0	0.9	2.0	2.6

Table 7.2: The cost in milliseconds of executing **move** including the cost of setting up communication channels.

- When the action and thread terminates, extract the mobile agent state , prepare the message consisting of the mobile agent state and mobile agent source code, and submit the resulting message to the next place.

When the cost of running the `next` extension has been established, the results can be combined with the results from the previous TOS experiments to estimate the expected cost of executing **move** with NAP2.

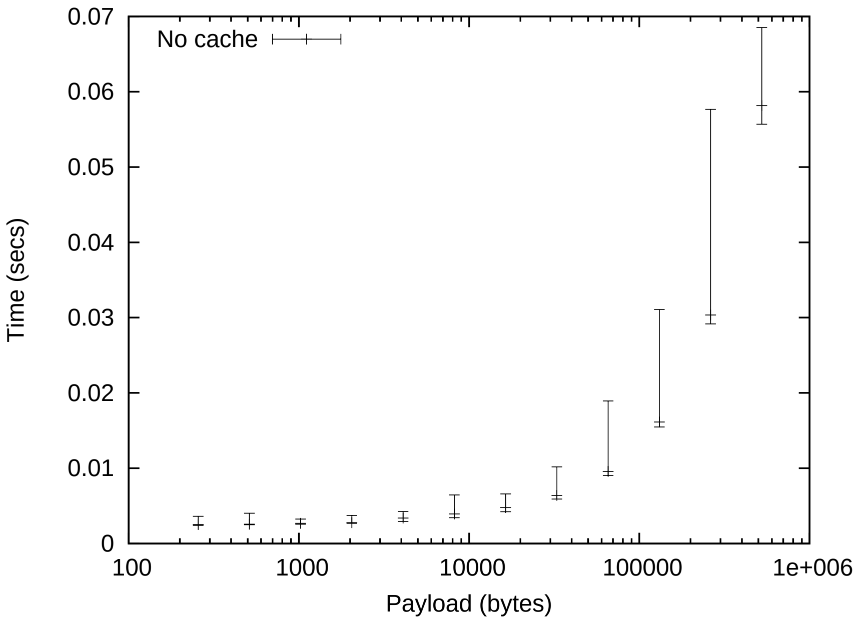
This experiment uses 10 places, where the mobile agent executes the **move** operation as its only action at each place. We measure the latency of completing all the 10 **move** operations, and since the action performed at each place is identical, the measured value is divided by 10 to give the latency of a single **move** operation. The `next` extension does not affect the code path that involves setting up a communication channel in TOS. The difference in the latency of doing **move** with and without communication channel setup will thus be the same as for the TOS performance experiments[1] We only give the results for the latency including the communication channel setup, since the common case is that a mobile agent visits a new place for each **move** operation.

The latency of **move** with communication setup included is visualized in Figure 7.3a. As expected, latency follows the trend of the TOS performance experiments, and Figure 7.3b plots the difference in mean latency between executing **move** and the previous TOS experiments for the corresponding agent payloads. There is an increase in latency for the larger messages, which is caused by the increased cost of handling the larger agent state in the `next` extension with larger payloads. The results are summarized in Table 7.2, where all time values are given in milliseconds. The P row denotes the payload, the M row shows the mean latency values, and the S row the number of samples with outliers removed. The D row gives the difference in mean latency versus the TOS mean latency results for non-established channels in Table 7.1.

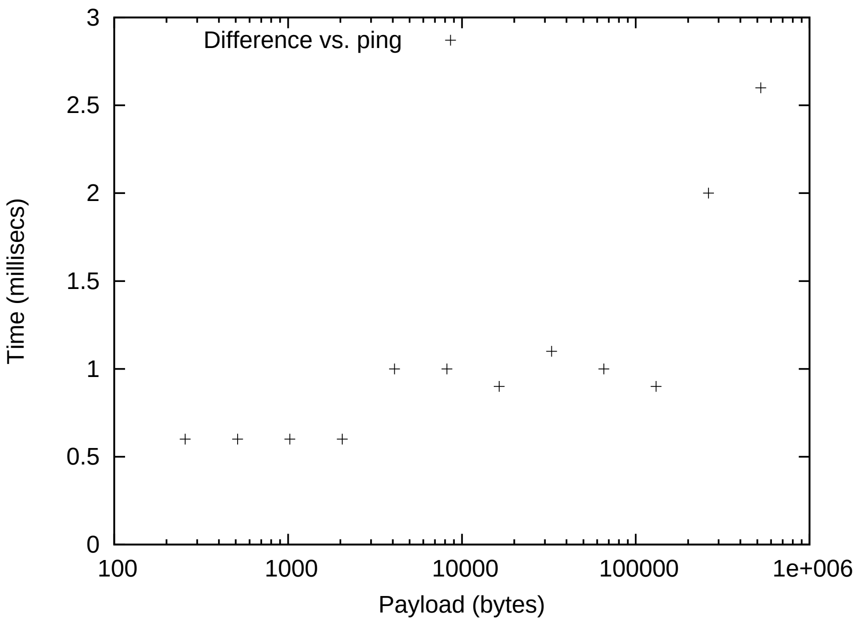
An Estimator for NAP2 Performance

Given knowledge of the latency of doing **move** without NAP2 as well as the latency of communicating messages with and without communication channel setup in TOS, we can estimate the expected latency of going from failover list $r(i)$ to failover list $r(i+1)$ in NAP2 by summing the following individual latencies:

[1]We actually measured the difference in latency to ensure this was the case, but the results are not included since they did not convey any new insights.



(a) Latency of migrating an agent using **move** including communication channel setup.



(b) Difference in mean latency of **move** vs. TOS message communication latency.

Figure 7.3: Cost of migrating an agent using **move**.

$$C_{setup} + C_{est} + C_{next} + X$$

C_{setup} is the sum of the latency of all TOS messages that require communication channel setup (Table 7.1), C_{est} the sum of the latency of all TOS messages without communication channel setup (Table 7.1), and C_{next} the latency of performing the steps for executing **move** operation without NAP2 (Table 7.2). X is the unknown computational overhead of executing the NAP2 protocol at a place, which involves executing the actual logic of the protocol. For each of the subsequent NAP2 experiments, we will measure the difference between the estimated latency and measured latency in an attempt to establish X.

Our evaluation of NAP2 performance now proceeds with three experiments to answer the question of determining the cost of executing NAP2: i) the latency of adding new places, ii) the latency of updating an existing set of places, and iii) the latency of removing places. These cases cover the regular operations performed on failover lists.

Based on the latency results from the previous experiments on the cost of TOS communication and executing **move**, we expect the next three experiments to show a linear increase in latency as a function of the number of places in the failover list. We expect the value of X to vary according to the number of places, and expect that our estimators are able to predict the trend of the latency. To test the accuracy of our estimators further, we use three different payloads: 0 bytes (only the agent code and state required to do the experiment), 64 kilobytes, and 512 kilobytes. We use the mean values of the previous experiments to establish the estimators. Also note that all of the following NAP2 experiments are executed using a timeout of 30 seconds in the failure detector to avoid interference from the messages sent by the `monitor` extension.

7.3.3 Adding Places

Adding new places to the failover list is typically used when bootstrapping a computation, or when the agent **move**s from a local area network to another.

The experiment uses a mobile agent that has two actions. The first action has a failover list $r(1)$ with a single place p that records the current time, and starts the second action with failover list $r(2)$ where p is the first place of $r(2)$ followed by $R-1$ additional places. The total number of places R in $r(2)$ is varied between 2 up to 9. The second action records the current time and calculates the latency. Going from 1 place in failover list $r(1)$ to R places in failover list $r(2)$ requires the following set of messages to be sent:

- One `turn` message from p in $r(1)$ to the last place in $r(2)$.
- $R-1$ `ack` messages from the last place in $r(2)$ to the first place in $r(2)$. The first place p in $r(1)$ and $r(2)$ is the same, so the final `ack` message does not contain any payload.

The places added to failover list $r(1)$ resulting in $r(2)$ have not previously been involved in the computation, so the latency includes the cost of setting up the

R	2	3	4	5	6	7	8	9	Avg(X)
0 kB	5.2	7.2	9.1	11.7	13.7	15.8	18	20	1.15
S	96	98	94	96	95	95	94	94	-
64 kB	12.4	21.1	29.6	39.4	48	56.6	65.3	73.8	1.5
S	94	98	99	98	99	98	95	95	-
512 kB	61.7	116.2	171.8	225.6	280.1	333.5	390.5	449.5	2.8
S	99	99	97	98	98	100	100	100	-

Table 7.3: The cost in milliseconds of adding new places, with the average value of X in milliseconds for each payload.

communication channel. The size of the mobile agent and its state is comparable to a TOS message with a payload of 512 bytes.

The estimators, α, for this experiment where $R > 1$ are thus:

$$\alpha_0 = C_{setup} + C_{est} + C_{next} + X = ((R-1)*2.0) + 1.8 + 0 + 0.6 + X$$
$$\alpha_{64} = C_{setup} + C_{est} + C_{next} + X = ((R-1)*8.6) + 1.8 + 0 + 1.0 + X$$
$$\alpha_{512} = C_{setup} + C_{est} + C_{next} + X = ((R-1)*55.5) + 1.8 + 0 + 2.6 + X$$

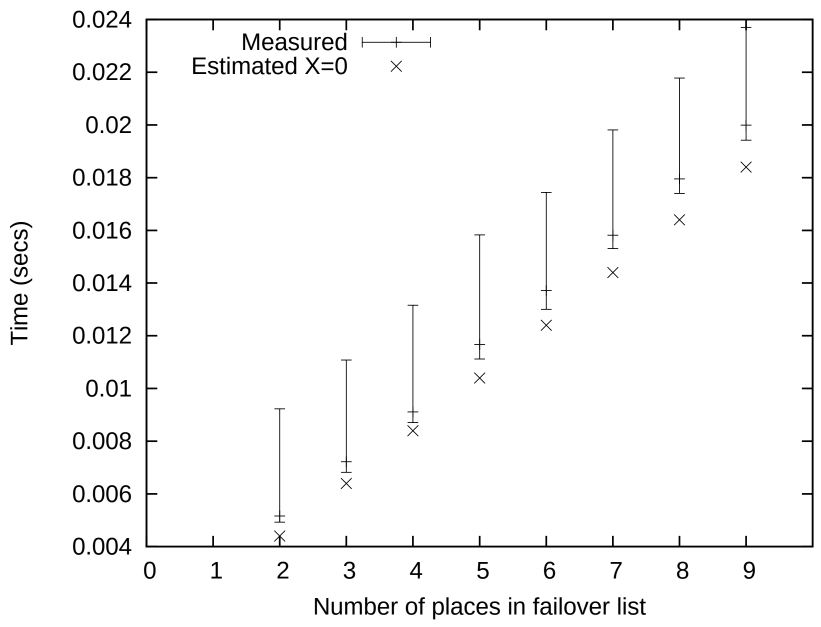
The results for 0 and 64 kilobyte payloads are shown in Figure 7.4 and 512 kilobyte payload in Figure 7.5. An estimator with $X = 0$ is plotted along with the mean values. We observe that the estimator is quite accurate in predicting the trend of the measured results and that the value of X increases with the number of places for all payloads.

Table 7.3 gives the sample results as well as the average value of X for each payload. All time values are given in milliseconds, where R denotes the number of places. For 0 and 64 kilobyte payloads, the average is 1.15 and 1.5 milliseconds. The cost for 512 kilobyte payload is a little higher, which conforms to the results for performing **move** without NAP2 in Table 7.2. The S rows specify the number of samples with outliers removed for the corresponding latency values of the previous row.

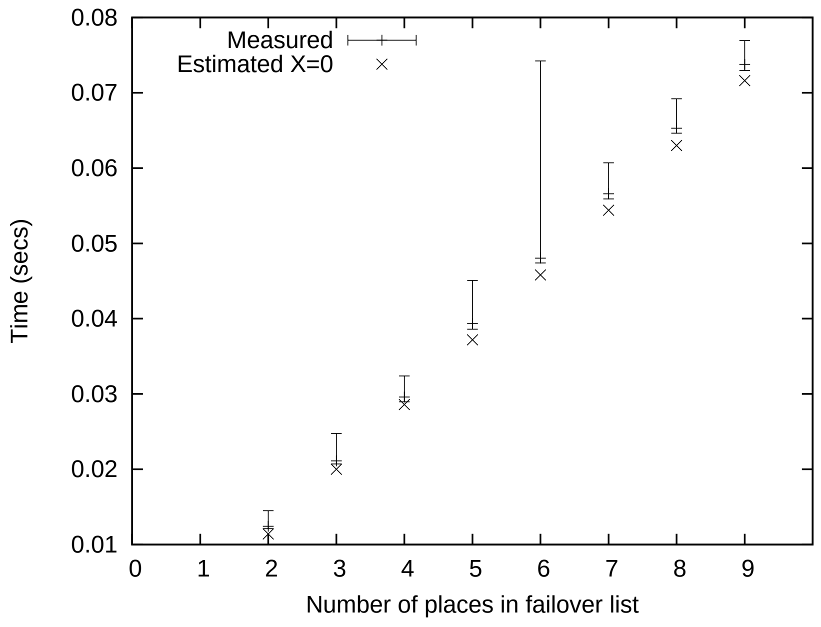
7.3.4 Checkpointing

During the computation, the agent may issue **checkpoint** to ensure that the state of the agent is updated at the current set of places in the failover list. The experiment uses an agent that has two actions. Both actions have the same R places in the failover lists $r(1)$ and $r(2)$, respectively. The first action is started with $r(1)$, records the time, and starts the second action. The second action is started with $r(2)$, records the time and calculates the latency. Updating R places without changing the failover list requires the following messages to be sent:

- $R-1$ `update` messages from the first place in $r(1)$ to the last place in $r(1)$.
- 1 `turn` message from the last place in $r(1)$ to the last place in $r(2)$ (i.e., itself).
- $R-1$ `ack` messages from the last place in $r(2)$ to the first place in $r(2)$.



(a) Latency for 0 byte payload.



(b) Latency for 64 kilobyte payload.

Figure 7.4: Latency of adding new places.

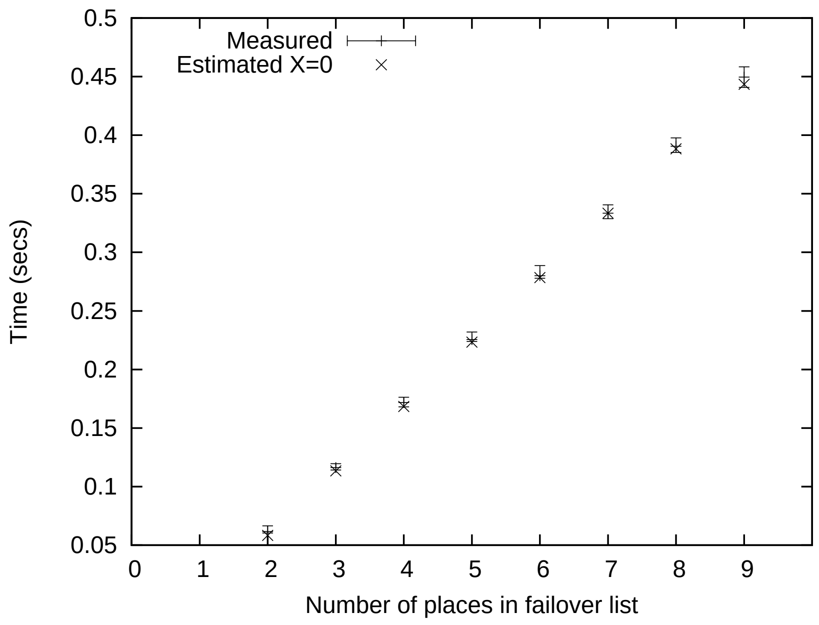
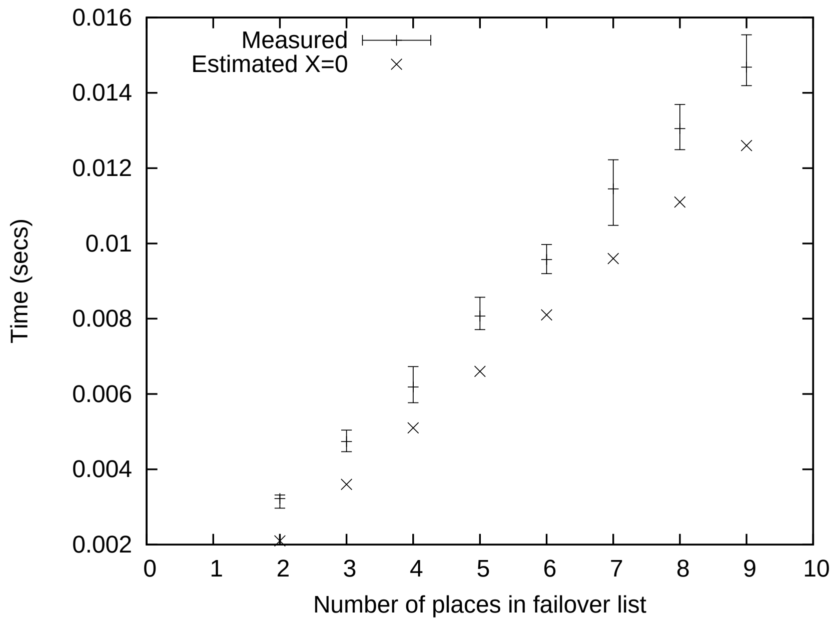


Figure 7.5: Latency of adding new places with 512 kilobyte payload.

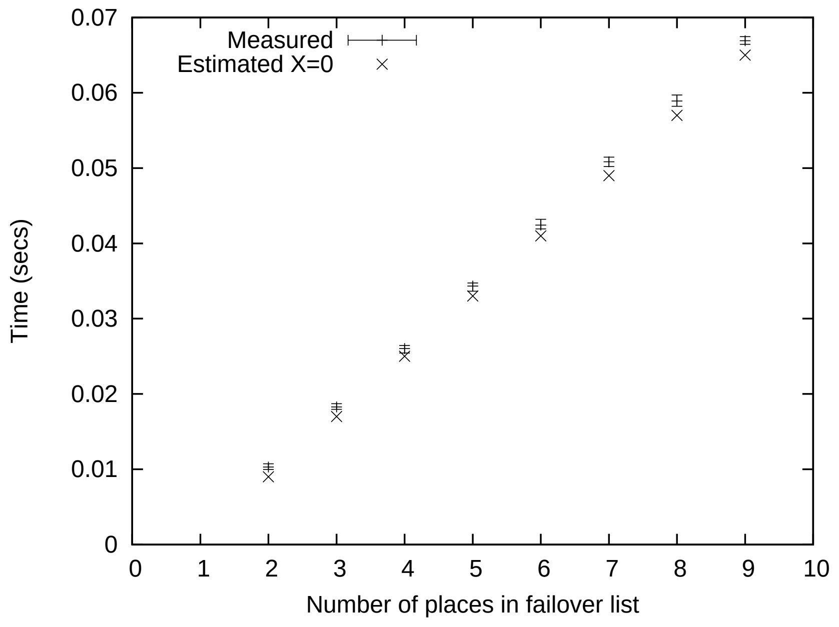
Since the order of the places in failover lists $r(1)$ and $r(2)$ are equal, all messages are sent on established communication channels, so there is no cost for setting up channels. Note that when a place sends the `ack` and `turn` message to a place in $r(2)$, only the modified mobile agent attributes since the `update` message needs to be included. This means that the $R-1$ `ack` messages and the `turn` message are comparable to TOS messages that have zero payload, so we used the latency for the zero payload TOS messages when estimating the latency of these messages. The mobile agent code and its state is comparable to a payload of 512 bytes. The estimators, β, for this experiment are:

$$\beta_0 = C_{setup} + C_{est} + C_{next} + X = 0 + (R-1) * 0.9 + (R-1) * 0.6 + 0.6 + X$$
$$\beta_{64} = C_{setup} + C_{est} + C_{next} + X = 0 + (R-1) * 7.4 + (R-1) * 0.6 + 1.0 + X$$
$$\beta_{512} = C_{setup} + C_{est} + C_{next} + X = 0 + (R-1) * 53.9 + (R-1) * 0.6 + 2.6 + X$$

The measured results for 0 and 64 kilobyte payloads are shown in Figure 7.6 and 512 kilobyte payload in Figure 7.7. An estimator with $X = 0$ is plotted along with the mean values. Like in the previous experiment, we observe that the estimator is quite accurate in predicting the trend of the measured results and that the value of X increases with the number of places for all payloads. Table 7.4 gives the average value of X for each payload. All time values are given in milliseconds. For all payloads, the average of X is 1.52 milliseconds or less. The S rows specify the number of samples with outliers removed for the corresponding latency values of the previous row.



(a) Latency for 0 byte payload.



(b) Latency for 64 kilobyte payload.

Figure 7.6: Latency of updating state to existing place.

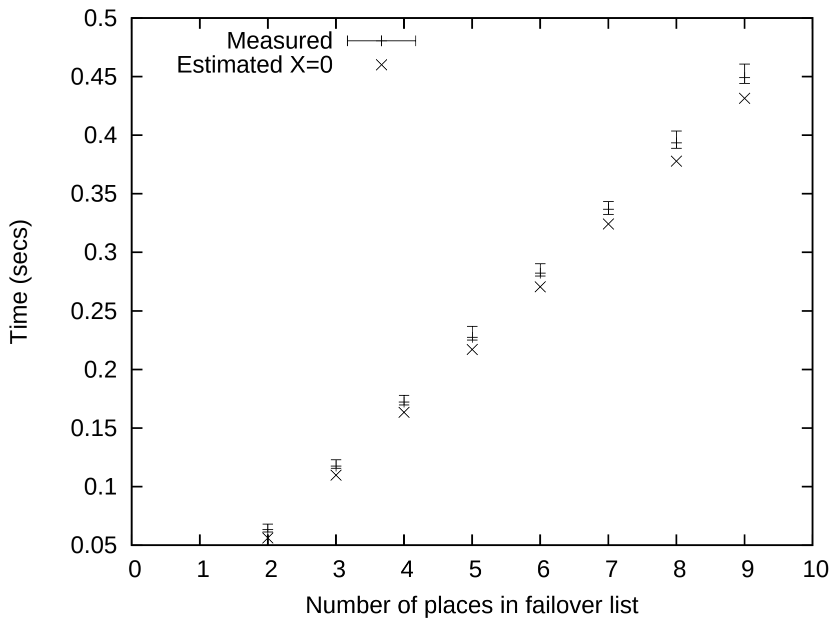


Figure 7.7: Latency of updating state of existing places for 512 kilobyte payload.

R	2	3	4	5	6	7	8	9	Avg(X)
0 kB	3.2	4.7	6.2	8.1	9.6	11.4	13	14.7	1.52
S	100	100	100	100	100	100	100	100	-
64 kB	10.3	18.3	26	34.3	42.4	50.8	58.9	66.9	1.50
S	100	100	100	100	100	100	100	100	-
512 kB	63.4	117.5	172.1	227.5	282.3	336.8	393.5	449.1	1.14
S	100	100	100	98	100	100	100	100	-

Table 7.4: The cost in milliseconds of updating places, with the average value of X in milliseconds for each payload.

7.3.5 Removing Places

When a mobile agent **move**s from one local area network to another, it typically adds a set of new places in the destination network and removes a set of places in the network that it left. The purpose of this experiment is to measure the latency of removing a set of places from the failover list.

The experiment uses an agent with two actions. The first action is executed with a failover list $r(1)$ with R places and records the time. Before starting the second action, $R-1$ places are removed from the failover list, resulting in $r(2)$. The second action starts with $r(2)$, records the time and calculates the latency. Removing $R-1$ places from a failover list consisting of R places requires the following messages to be sent:

- $R-1$ `update` messages from the first place in $r(1)$ to the last place in $r(1)$.

R	2	3	4	5	6	7	8	9	Avg(X)
0 kB	2.8	5.1	5.8	7	7.8	8.8	9.6	10.6	1.01
S	99	98	98	99	98	98	99	99	-
64 kB	9.9	18.5	25.7	33.3	40.7	47.8	55.1	62.4	0.82
S	99	99	99	99	100	99	98	99	-
512 kB	59.9	114.2	166.9	221.1	274	327.2	379.5	433.8	0.4
S	100	98	100	100	100	100	97	100	-

Table 7.5: The cost in milliseconds of removing places, with the average value of X in milliseconds for each payload.

- 1 `turn` message from the last place in $r(1)$ to the remaining place in $r(2)$.
- $R-1$ `stop` messages from the remaining place in $r(2)$ to the removed places in $r(1)$.

All `update` messages are sent on established communication channels, but the `turn` message sent from the last place in $r(1)$ to the remaining place in $r(2)$ requires a channel to be set up, except in the case where we go from 2 places in $r(1)$ to 1 place in $r(2)$. Only the `update` messages carries payload, since the `turn` message is sent to a place present in $r(1)$. The latency is recorded after the `stop` messages have been sent by the first place in $r(2)$. This means that the latency does not include the cost of handling `stop` messages at the receiving places in $r(1)$. The mobile agent code and its state is comparable to a TOS message with a payload of 512 bytes.

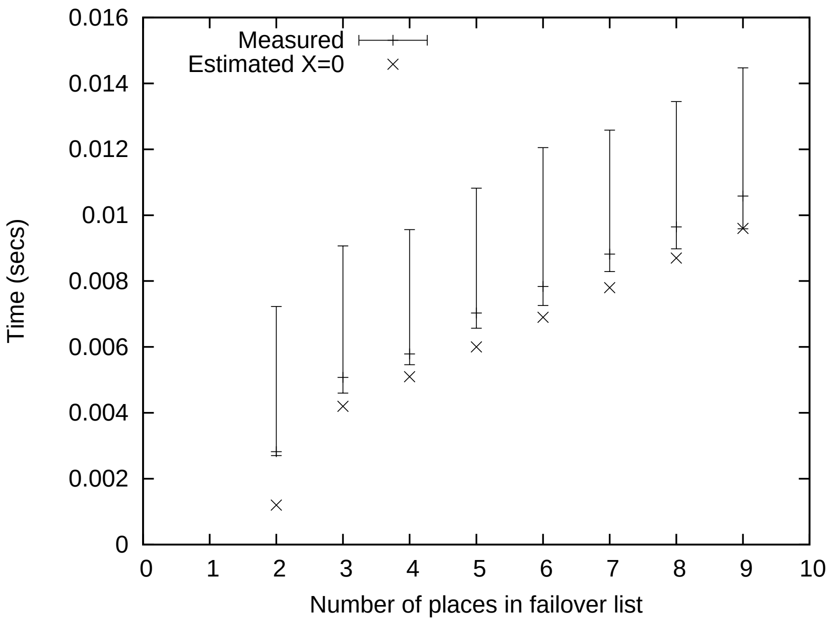
The estimators, γ, for this experiment are thus:

$$\gamma_0 = C_{setup} + C_{est} + C_{next} + X = 0 + (R-1) * 0.9 + 1.8 + 0.6 + X$$
$$\gamma_{64} = C_{setup} + C_{est} + C_{next} + X = 0 + (R-1) * 7.4 + 1.8 + 1.0 + X$$
$$\gamma_{512} = C_{setup} + C_{est} + C_{next} + X = 0 + (R-1) * 53.9 + 1.8 + 2.6 + X$$

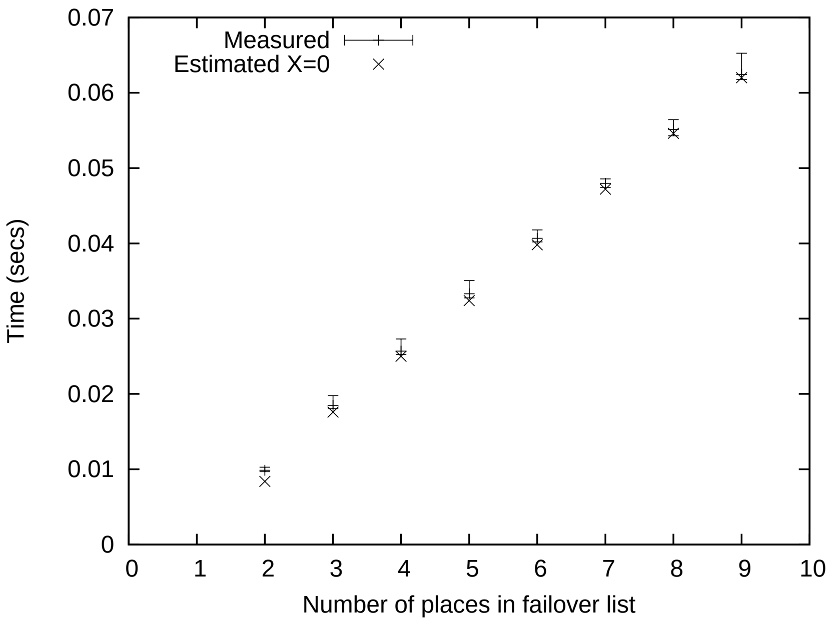
The measured results for 0 and 64 kilobyte payloads are shown in Figure 7.8 and 512 kilobyte payload in Figure 7.9. An estimator with $X = 0$ is plotted along with the mean values. As in the previous experiments, we observe that the estimator is quite accurate in predicting the trend of the measured results. Table 7.5 gives the average value of X for each payload. All time values are given in milliseconds. For all payloads, the average of X is 1.01 milliseconds or less. The S rows specify the number of samples with outliers removed for the corresponding latency values of the previous row.

7.3.6 Estimators Revisited

The significant result from the previous three experiments is that we have established that the estimators are able to predict the trend of the measured values. This means that the latency of our NAP2 implementation scales fairly accurate with our predicted model, and that the average value of X is generally small, less than 2.9 milliseconds in all our experiments, and for all but 1 experiment less than 1.53 milliseconds. However,



(a) Latency for 0 byte payload.



(b) Latency for 64 kilobyte payload.

Figure 7.8: Latency of removing existing places.

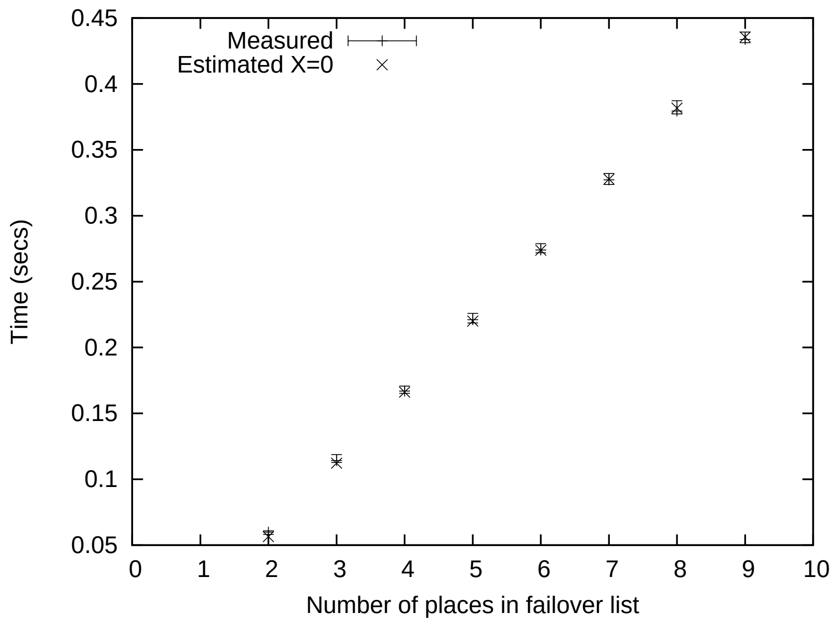


Figure 7.9: Latency of removing existing places for 512 kilobyte payload.

looking at the values of X as a function of the number of places shows that there is no clear correlation. This indicates that factors such as thread scheduling and lock contention in the TOS runtime and Python interpreter are likely to cause perturbations in the trends of X as a function of places.

During the performance evaluation of NAP2, the estimators proved useful when identifying performance problems in TOS that did not show up in the TOS and **move** experiments. Our first run of the experiments had values of X ranging between 10 and 25 milliseconds, an order of magnitude larger than the current result.

7.4 Two Backup Management Strategies

The previous experiments have evaluated the latency of NAP2 for variations of failover list operations. We now evaluate how NAP2 performs with the following two backup management strategies:

1. The itinerant-style mobile agent strategy with transparent backup management, as specified in Section 6.3.3. This strategy is identical to a strategy used in an earlier version of NAP [89] that only supported transparent backup management.

2. The network aware mobile agent strategy where the backup management is explicitly handled by the mobile agent.

These two strategies will be evaluated by a mobile agent computation whose purpose is to change the places of a failover list to from one disjoint set of places to another. This is a strategy used when a mobile agent **move**s from one local area network to another to reduce the time the failover list spans more than one local area network. The purpose of this experiment is thus to answer whether transparent backup management has smaller or larger latency for this scenario. The mobile agent and its state is for both strategies 1400 bytes. We run each experiment with three different additional payloads, 0 bytes, 64 kilobytes and 512 kilobytes.

7.4.1 Transparent Backup Management

The strategy used with itinerant computations for transparent backup management works as follows: when the agent **move**s from a place p to a place q, the last place in the failover list is removed, and the new place q is added to the head of the list. The order of the list is preserved, and with a failover list length of f places, the computation can sustain $f-1$ failures. We vary f between 1 and 5 and calculate the latency between two successive actions for each value of f.

The mobile agent has two actions. The first action is started at a place p with f places in the failover list, and it records the current time at p. The failover list is then updated by removing the last place in the list, and the next action a is started on a place q that is added to the head of the list. The action a checks whether f **move** operations have been performed. If so, a message is sent back to place p which records the time and calculates the latency. Otherwise, the agent performs another **move** operation.

The set of messages sent in this experiment for each invocation of **move** is thus comparable to the messages sent in the experiment with the β estimator. The difference is that one place is added, requiring an additional `ack` message with communication channel setup latency, and one is place is removed, requiring f `stop` messages to be sent by the remaining places. An additional latency is incurred by the first `update` message since this message includes the communication channel setup latency. Also, since the total latency of the experiment is calculated by place p, the latency includes the cost of sending the message to p from the place where the last action was executed.

7.4.2 Network Aware Backup Management

The network aware backup management strategy used when going from one set of places R to a disjoint set of places R' can be implemented by a mobile agent with two actions. The first action adds f places from R' to the failover list, while still keeping the f places from R. The second action removes f places from R from the tail of the failover list. One might be tempted to do both operations in a single action, but this would violate assumption F1S, as specified in Section 6.5.3, where $r(i) \cap r(i+1) = f$ to tolerate $f-1$ failures when going from failover list version i to $i+1$.

The set of messages sent during this experiment can be attributed all three estimators: when adding new places from R' in the first action, the α estimator can be used to estimate the messages for the new places, in combination with the β estimator that estimate the messages for the places in R. When removing the places in R in the second action, the γ estimator can be used. Thus, we expect the measured latency of the two actions to be close to $\alpha+\beta+\gamma$. We vary the value of f from 1 to 5, and compare the results with the latency of the transparent backup management computation in the previous section that invokes **move** f times to achieve the same effect. Similar to the transparent backup management strategy, the first action executed at place p records the time and the place executing the last action then sends a message to p to calculate the latency.

7.4.3 Results

We do not use estimators for these two experiments because the message size for the mobile agent falls between the payload of 1024 and 2048, which makes an estimation for the zero payload computation difficult to interpret. Several actions are executed in sequence, and this also causes influence from the handling of receiving `stop` messages, which are not covered by the γ estimator. In addition, none of the existing estimators can be directly applied to the transparent strategy.

However, we expect the transparent strategy to be faster for small values of f because less messages are sent than with the network aware strategy since $\beta < \alpha + \beta + \gamma$. However, the disadvantage with the transparent strategy is that the number of messages with the complete payload for N places is $O(N^2)$ where the network aware strategy is $O(N)$. Thus, at some number of places, we expect to find a cross-point where the latency for the transparent strategy exceeds the network aware strategy.

Figure 7.10, 7.11 and 7.12 depict the latency for 0, 64 kilobyte and 512 kilobyte payloads, respectively, and Table 7.6 summarizes the results. The R row specified the number of places in the failover list, and the S rows the number of samples with outliers removed for the transparent (T) and network aware (N) strategies. The remaining rows specify the mean latency with the specified payload for transparent (T) and network aware (N) strategies. For zero payload, the latencies are fairly equal at 3 places, while the network aware strategy performs significantly faster at 4 and 5 places. For 64 kilobyte and 512 kilobyte payloads, the network aware strategy performs faster at 3 places.

The current results indicate that when the payload increases, the cross-point for the latency converges towards fewer places in the failover list. The two experiments are performed within a local area network. This also means that if the computation was performed across two local area networks connected by one or more weak links with poor bandwidth, the results would favor the network aware strategy more, and probably move the cross-point back even further.

7.5 Reducing Latency

None of the experiments in this chapter use the `stable` message optimization. The main reason for this that we wanted the experiments to reveal the cost of running the entire NAP2 protocol. We can, however, tweak our existing estimators to gain insight on the expected performance gain from using the stable optimization. Recall that the `stable` message is sent by the sender of the `turn` message in the protocol.

For the α estimator, the impact of the `stable` message depends on the number of places in the failover list and can be significant since the entire payload is sent in the `ack` messages. As an example, consider α_{512} with 9 places. The latency without `stable`

R	1	2	3	4	5
0 kB (T)	10.5	22.7	40.4	64.9	94.7
0 kB (N)	12.4	25.0	38.0	54.9	73.1
S (T)	97	90	98	97	99
S (N)	94	99	94	96	97
64 kB (T)	18.4	57.6	115.3	194.0	292.4
64 kB (N)	27.0	65.9	105.2	159.9	198.8
S (T)	98	97	97	97	95
S (N)	99	98	96	98	97
512 kB (T)	71.4	299.2	640.6	1093.0	1667.1
512 kB (N)	125.1	348.5	573.0	818.8	1052.8
S (T)	99	100	97	100	98
S (N)	98	99	98	98	97

Table 7.6: The cost in milliseconds of executing transparent (T) and network aware strategies (N).

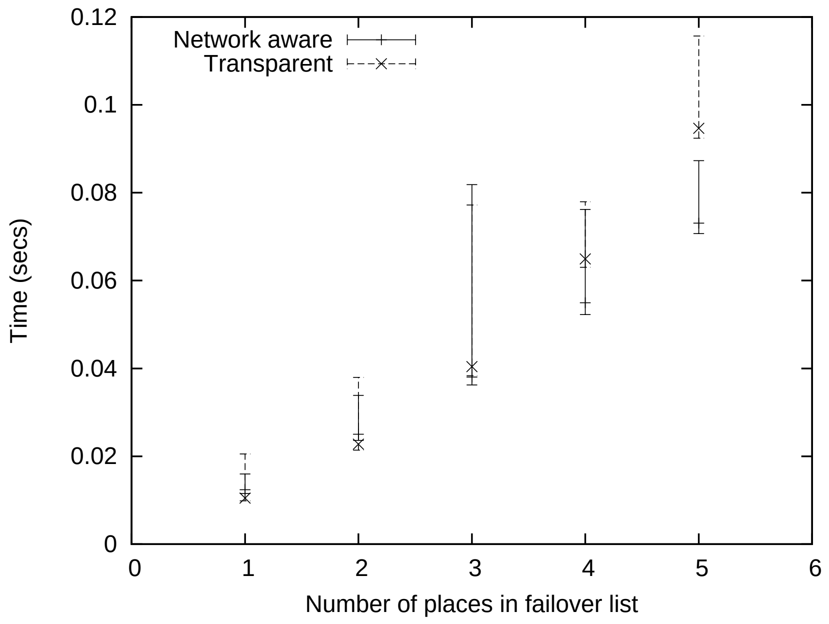


Figure 7.10: Latency of transparent vs. network aware strategy for zero payload.

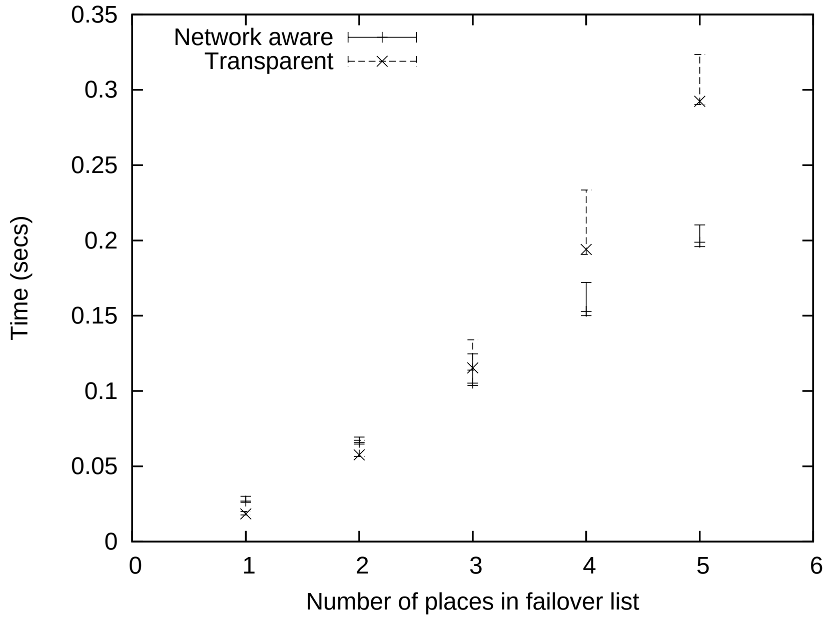


Figure 7.11: Latency of transparent vs. network aware strategy for 64 kilobyte payload.

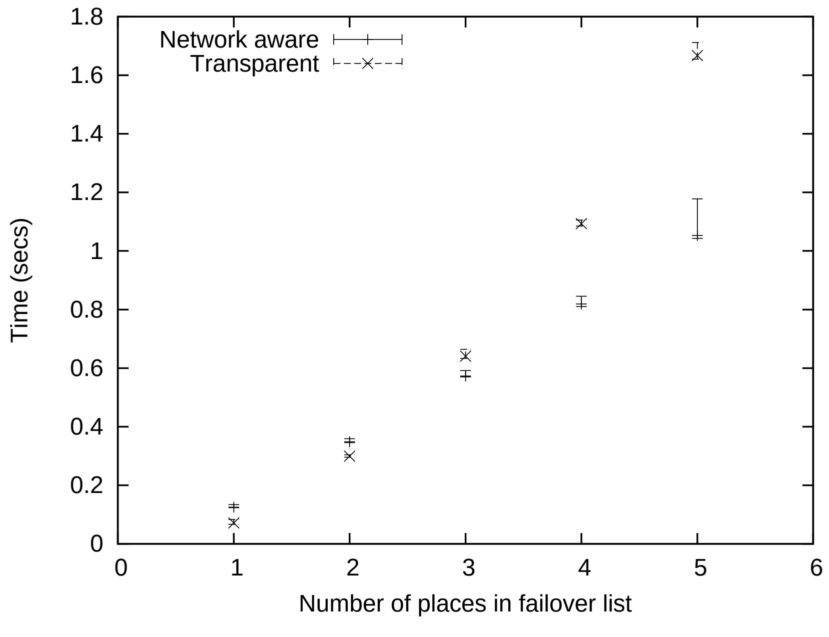


Figure 7.12: Latency of transparent vs. network aware strategy for 512 kilobyte payload.

was estimated to 443 milliseconds. An estimate with the `stable` message would for the same number of places be 59.9 milliseconds, which is a reduction in latency by almost an order of magnitude.

The latency saved for the β estimator can also be significant. However, since all places in $r(2)$ are in $r(1)$, the `ack` messages do not need to carry any payload and the latency saved by using `stable` is thus not a function of the payload. Consider β_0 with 9 places. The latency without `stable` was estimated to 12.6 milliseconds. An estimate with the `stable` message would in this case be 5.6 milliseconds with 9 places, reducing the latency by over 50 percent.

For the γ estimator, there is actually no reduction in latency since all but one place are removed from the list. This implies that there is only one `ack` message sent from the last stopped place to the remaining place, effectively the same as the `turn` message would do and makes the use of the `turn` message redundant.

For the transparent strategy, the latency reduction is at least as good as with the β estimator. In addition, since the first place in the list is new after a **move**, the `stable` message with the entire payload is sent in parallel with the propagation of the `ack` messages among the set of places in the previous list. The network aware strategy also benefits from the `stable` message: when adding the new places, the next action can be started early while the `ack` messages are propagating, with results similar to the α estimator.

Although the stable optimization decreases the latency for starting an action, there are two minor drawbacks. First, if the first place in the failover list is not in the previous failover list and the payload large, then the stable message incurs additional bandwidth utilization. However, if the first place in the failover list is not new, the `stable` message does not contain the entire payload of the agent and incurs little extra bandwidth usage. Second, since the subsequent action cannot start until all `ack` messages have been communicated, short computations with actions that terminate quickly (such as the ones used for the experiments in this chapter) may not benefit significantly since actions are typically finished before all `ack` messages have been communicated.

There are other optimizations that could be applied to reduce latency. Agent payloads can be compressed and `update` messages do not need to contain mobile agent attributes that have not changed since the last action executed. To minimize the latency contributed by the communication channel setup, a *scout* agent can be executed prior to the computation. The scout **move**s along the expected itinerary of the agent computation to ensure communication channels are established before the actual computation starts.

7.6 Summary

We started the chapter by asking two questions:

1. What is the latency of running a mobile agent application in TOS with NAP2?

2. Is the latency of changing between a failover list R to a disjoint failover list R' larger with transparent backup management?

To answer the first question, we measured the basic performance of communicating messages in TOS and the cost of performing a mobile agent migration without NAP2. Based on these results we developed three estimators that were able to predict the performance of NAP2 with good accuracy. We thus established that the latency of our NAP2 implementation scales according to our predictions.

To answer the second question, and evaluate the performance hypothesis proposed in Section 1.3, the latency of adding and removing multiple places in NAP2 was measured and compared to the latency of doing a comparable number of **move** operations. The results indicate that for fewer than 3 places, transparent backup management was faster and rejecting our hypothesis, but with 3 or more places and larger payloads, NAP2 was faster, confirming our hypothesis. The experiments were conducted within a local area network, and the results show that if bandwidth is not as good, NAP2 latency is likely to be faster than transparent backup management for fewer than 3 places in the failover list and agents with small payload.

We ended the chapter by estimating how much latency can be saved by using the `stable` message optimization. The savings can be significant: for large payloads when adding places, the latency for starting an action can be reduced by an order of magnitude when using `stable` messages.

Chapter 8

Discussion

In this chapter we discuss issues with our assumptions, design, and implementation that have not been previously addressed in this dissertation. We start by discussing our assumption on linear broadcast and how the performance of NAP2 can be improved. Next, we discuss the applicability of our solution for both mobile and non-mobile application domains. We then end this chapter by discussing the tradeoff between providing exactly-once semantics and blocking the computation.

8.1 Performance

The performance of NAP2 depends on the latency of maintaining the failover list. We now discuss our choice of broadcast strategy, optimizations related to state dissemination, and how replica placement affects performance.

8.1.1 Broadcast Strategy

The NAP2 implementation evaluated in Section 7 uses a linear strategy with Fault-Tolerant Forwarding for updating state among the places in the failover list. We chose to use a linear strategy since it minimizes the amount of messages sent across a single weak network link. When the set of places spans more than one local area network, minimizing communication on weak links is attractive. However, within a local area network, linear broadcast may cause longer latency than a tree-based strategy [160]. Recall from Section 6.5.3 that when changing from failover list $r(i)$ to $r(i+1)$, the places in $r(i+1)$ that are changed to $< r(i+1), i+1 >$ with `ack` messages by the second FTF do not require the `ack` message to use a linear broadcast strategy. However, the elected place for $i+1$ can only start the next action $i+2$ when it has received an `ack` message from all non-failed places in $r(i+1)$ so that no places are in the transitional state, as required by property NAPTransElected specified in Section 6.5.3. An approach to improve local area network performance is to flood the `ack` messages to all members of $r(i+1)$ that we have not already communicated the `ack` message with. Flooding has quadratic message complexity and requires more messages than linear broadcast, but may result in lower broadcast latency. However, a problem for sender-initiated flooding algorithms is that large broadcast groups can results in message implosion [139].

For larger number of places in the failover list, a reliable best-effort multicast protocol such as Bimodal Multicast [26] or constrained flooding [112] could be appropriate for sending the `ack` messages. Bimodal Multicast uses a limited form of flooding to reduce the amount of messages, and the protocol runs in two stages. The first stage disseminates the message to all members using a network level (broadcast) protocol, for instance IP multicast. The second stage executes a gossip based anti-entropy protocol where members of the broadcast exchange message history summaries and request missing messages from other members. While Bimodal Multicast is a best-effort protocol, much like WAMW, experiments show that it has stable throughput and high reliability when message loss is uncommon [26]. When message loss is more common, a protocol like Slingshot [14], that uses Forward Error Correction for faster packet recovery, can be more reliable. Slingshot uses fewer packets than Bimodal Multicast for recovering packets, and recovers lost packets an order of magnitude faster than the established SRM multicast protocol [14, 54]. Regardless, best-effort protocols still have weaker delivery guarantees than required by the NAP2Change definition specified in Section 6.4.5.

While a flooding or tree-based protocol may perform well with few places and within a local area network, increasing the number of messages crossing a weak communications link may cause the broadcast latency to increase. An approach to avoid this problem is to deterministically choose a place from each local area network to act as a network proxy for communication with other local area networks, and have such proxies forward all messages between them using a point-to-point protocol. A problem with managing multiple broadcast strategies this way is the increased complexity compared to the current NAP2 implementation that only requires the FTFs specified in Section 6.5.2. It is also not clear that latency will be significantly reduced compared to NAP2 with `stable` messages. In addition, network-level broadcast protocols such as IP multicast are not generally available in the wide-area Internet [45].

8.1.2 The F1S Property

In Section 6.5.3 we decided to strengthen property F1 into F1S to avoid having places in $r(i+1)$ monitor the places in $r(i)\backslash r(i+1)$ after $A(i+1)$ or $R(i+1)$ has terminated. Since this optimization requires that $r(i) \cap r(i+1)$ contains at least one non-failed place, the latency of the protocol increases when $r(i) \cap r(i+1)$ is smaller than f. This latency increase is because two **move** operations must be performed, similar to the network aware backup management strategy that we used in the performance evaluation in Section 7.4.2.

There are, however, other practical ways to terminate the monitoring while satisfying F1. One approach is to set an upper bound Δ on the time a place can be in the transitional state. When a place p in $r(i)\backslash r(i+1)$ enters $<<>, \mathsf{true}>$, as specified by step In2 in Section 6.5.3, a timer is started. If this timer exceeds Δ time units, p enters $<<>, \mathsf{false}>$, as specified by step Out2 in Section 6.5.3. The problem is determining an appropriate value for Δ. If the agent payload is large and bandwidth low, the execution of the NAP2 protocol can take significant time to complete, and cause Δ to expire before the protocol has completed, thus violating property F1. In most cases, however,

Δ could be set to a large value or be encoded as part of the agent in an agent attribute *delta* to allow mobile agents to calculate or configure an appropriate value based on the computation.

8.1.3 State Optimizations

Recall two process migration optimizations from Section 2.1.1: *pre-copy* [35, 38] and *lazy-copy* [125, 145, 185]. We now discuss whether similar optimizations can be applied to reduce the cost of state dissemination in NAP2.

The pre-copy approach was first used by the V distributed system [35]. The general idea is to execute the computation concurrently with copying state to the destination place for migration. More recently, this approach was also used to migrate virtualized machines [38]. Since migration in V is implicit, applying this optimization to NAP2 is not straightforward since NAP2 uses explicit migration. Thus, the failover list for the next action may not be chosen until the **move** operation is about to be invoked. Without the next failover list, the pre-copy protocol cannot execute since there is no knowledge of the destination places. If, however, the computation follows a predefined itinerary where the set of places to execute the actions is known at start of the computation, the NAP2 runtime can start sending the agent state to the place for the second action when the first action starts. All subsequent updates to agent attributes are then trapped by overriding the `__setattr__` class method in the agent and either sent immediately to the place for the second action or buffered until the action terminates. If the set of places on the itinerary of an agent changes during the execution of the action, then the pre-copied data needs to be communicated to the new places in the itinerary.

The copy-on-reference [185], or lazy-copy approach, was first used by the Accent system [145] and later also by Mach [125]. The approach is to migrate the minimal state required to allow the process to start executing at the destination place and copy required pages on demand from the source place. Unlike the pre-copy approach, lazy-copying is simpler to integrate effectively with explicit migration. For instance, a lazy-copy approach in NAP2 can have the `stable` message only include the minimal agent state required to start the action. The NAP2 runtime then traps accesses to agent attributes by overriding the `__getattr__` class method in the agent. When access has been trapped, the runtime forwards accesses to these attributes to the place at the tail of the failover list. The `ack` messages carry the entire state so eventually all the state will be present at the place executing the action before the next action is started. This also means that should the place that provides the complete set of agent attributes fail, the next place in the failover list can be accessed instead.

We have also considered the lazy-copy approach as a mechanism to enforce access control on specific agent state attributes by the Principle of Least Privilege [154]. For instance, if an agent is sent to a place that is suspected as malicious, only a minimal set of agent attributes is sent with the agent. The execution of the agent is then monitored by a (trusted) runtime environment that enforces access control on the remaining agent state when the agent subsequently tries to lazy-copy state.

8.1.4 Replica Placement

The appeal of optimizing replica placement to increase performance or robustness has also been observed and applied by other systems, which we now discuss.

Chain Replication [176] is a protocol that focuses on providing strong consistency with high throughput and availability. The protocol organizes a set of replicas in a logical chain where read operations are submitted to the head of the chain while write operations are submitted to the tail and propagate towards the head. Thus, the result of write operations are not available for read operations until all replicas have received the value, which ensures strong consistency. Chain Replication is similar to NAP in that it uses a linear monitoring relation between the replicas. Upon failures, the failed place is removed from the chain. Optionally, a new replica can be added to maintain sufficient fault-tolerance. If Chain Replication is deployed in a wide-area setting, then the replica placement becomes crucial to optimize towards access patterns, network proximity, and the reliability of a place [176]. While reordering the chain is possible in theory, this area is not explored further in the paper.

The work in [2] offers a read/write storage abstraction where replicas are placed according to predictions of future accesses based on the history of accesses so far. Data is reordered, replicated, or deleted from places. The purpose is to reduce the latency of communicating messages sent by clients to replicas during read and write operations. An optimal replica strategy has places that are frequently read located close to the reader. Writes, however, are performed on multiple places that benefit from being close to each other. A similar prediction method would be beneficial for NAP computations that, for instance, implement a server interface. The head, h, of the failover list that executes the server interface should migrate close to the clients. The rest of the failover list, r, should consist of places that are topologically close to each other (and, if possible, to the head) to make the replication perform faster. The challenge here is the tradeoff between optimizing for h or r, since communication between h and r could become a bottleneck to the computation.

The work in [183] presents a dynamic object replication approach for distributed databases. The network topology is modeled as a graph. For each read and write operation, an adaptive data replication (ADR) algorithm tries to optimize latency for executing reads and writes by altering the graph. It is shown that the resulting graph is convergent-optimal, meaning it eventually stabilizes with at least as good performance as the initial graph. A similar approach would be beneficial for use with NAP, since an algorithm like ADR can be used to optimize the placement of places within a failover list. Exploiting the benefits of an algorithm like ADR, however, requires the computation to be long-running and assumes a failover list that is relatively static.

Distributed hash table services (DHT) like Pastry [152], Tapestry [186], or Chord [170] are also concerned with replica placement and are designed for wide-area operation. A DHT is a data structure where key-value pairs are stored and replicated within a network of connected nodes. All nodes participating in the DHT are typically organized in a logical ring [152, 170] where a given position in the ring maps to a part of the key namespace. Due to the number of possible participating nodes, each node cannot know which nodes map to a specific key for all keys, which is why DHTs route packets to nodes which is known to be *closer* to the destination. DHTs are generally able to route packets to its destination using *O(log N)* hops, where each node only maintains as little as *O(log N)* links to its neighbors.

A lot of research on DHTs and peer-to-peer networks has focused on how DHTs perform when deployed in networks with high *churn* rates [23, 37, 74, 144]. Churn is the process where nodes continuously arrive and depart from the table. A study on churn by Rhea et al. [148] revealed that Pastry failed on the majority of lookups (i.e., getting the value for a key) under heavy churn. Chord behaved better as all lookups completed successfully, although with significant latency. An attempt to avoid the effects of churn is to use proximity neighbor selection, where neighbors are chosen based on the measured network latency, and the routing table is re-tuned as the network changes. The same study shows that by measuring the latency of random key lookups, latency can be reduced with as much as 24% with only a small bandwidth overhead. Also, the study shows that with significant bandwidth overhead, the latency can be reduced with 42%. Other techniques, such as the nearest-neighbor algorithm in Pastry [31], have also been shown to exhibit good reduction in lookup latency during heavy churn. While DHTs are designed for a very large number of participants, the results are relevant for long running wide-area NAP computations. For instance, techniques such as proximity neighbor selection can be used to choose places for the failover list based on the measured network latency.

A weakness of the network adaption in NAP2 is that distinguishing local area networks is not always possible by inspecting the IP addresses of places. For instance, places may communicate indirectly using a roaming protocol such as Mobile IP, where a home agent forwards inbound communication to a remote place [136]. Hence, methods based on actual predictions or communication latency measurements may give better results than our approach in practice.

Our approach for avoiding exposure to weak communication links is to avoid failover lists that span multiple local area networks. A problem with this approach is that it may make a mobile agent computation more prone to geographically correlated failures. Assume that a mobile agent application is initiated by a place p in local area network s, and that the computation then **move**s to a local area network s' and uses a failover list based on the places in s'. In this case, if s and s' are partitioned away from each other, there are no places in s to make the computation progress on behalf of p. To cope with this, a replica placement strategy inspired by the rally point is frequently used by NAP2 computations: The mobile agent keeps the originating place p that initiated the agent as the last place on all subsequent failover lists. This way, the place p can monitor the progress of the agent through the agent state updates when NAP2 executes a **move**. If the originating place is the only remaining place in the failover list, the computation is started over again. While this can cause duplicate computations if s' partitions away from p, the computation at least makes progress.

8.2 Applicability

Code mobility is not new and is a technology that constantly resurfaces. The paradigm started with process migration [124] and worms [163], and continued with remote evaluation [168] and mobile agents [60]. The currently emerging uses of code mobility are mobile agents in grid computations [61, 78, 178] and migration of virtualized computers [38, 155]. Hence, code mobility appears worth studying just from the point of being a technology that will be reused with high probability.

The use of mobile agents in grid environments influenced our decision to make NAP2 adaptive to network topology. Other mobile agent systems have followed similar approaches with stronger constraints: In the Mobile Agent Framework [41], each local area network has a checkpoint manager that monitors the progress of agents within that local area network. The checkpoint manager is not replicated and if it fails agents can no longer execute within that local area network. By changing from one disjoint set of places to another, NAP2 computations achieve behavior that similar to the Mobile Agent Framework when moving between subnets. However, by having two or more backups, NAP2 computations are not vulnerable to the failure of a single component such as the checkpoint manager. Allowing places to be explicitly organized during the computation thus expands the set of possible uses of NAP2.

While applicability in the field of mobile agents is important, we have also found use of NAP2 in systems where code mobility is not used. The main reason why this is possible is that NAP2 does not require the computation to start the next action on a different place to ensure f failures can be tolerated. A system using a transparent fault-tolerance protocol, as specified in Section 6.3.3, would require that the mobile agent constantly **move**s to ensure f failures to be tolerated. Conversely, NAP2 does not require that the next action starts on a different place to maintain the Bounded Crash Rate, but can instead execute computations as a conventional primary-backup system. As an example of this, we now present and discuss a revised version of WAMW that uses NAP2 to ensure sufficient masters are replicated within a local area network.

8.2.1 Wide-Area Master Worker

The WAMW algorithm uses a static configuration of masters, where there is at least one master per local area network. If all masters crash, then the computations stops even if all workers are available. A way to increase robustness for master failures is to use more than one master per local area network. A problem with this strategy is that the broadcasts between the masters become more costly and the bandwidth requirements of the protocol increase significantly. Hence, determining how many masters to use per local area network is difficult. Too many masters and the increased wide-area traffic becomes a problem, too few masters and the risk of all masters failing within a network increases. An alternative solution is to use a separate protocol for replicating masters within a local area network. We now discuss how NAP2 can be used to implement such a solution.

The general approach is to replicate the WAMW task allocations and task results among a set of places. Doing so requires the master loop in Pseudocode 5.2.3

(Section 5.2.4) to be structured in terms of fault-tolerant actions. Pseudocode 8.2.1 shows the master loop structured as two actions. The first action, `action1` initializes the data structures and terminates by starting the second action, `action2`. Each invocation of `action2` corresponds to the execution of an iteration of the main loop in Pseudocode 5.2.3, and ensures that task allocations and results are periodically replicated among the places in the failover list.

The task of maintaining the places in failover list is performed by the `check_backups` function in `action2`. All workers are executing TOS with the `nap` and `monitor` extensions enabled. Thus, `check_backups` inspects the failover list, and if the list is shorter than f, an idle worker is picked from the *idle_workers* list and added to the failover list. If there are no idle workers, a worker is randomly picked from the list of active workers. The recovery action is the same as the regular action for `action1`. The recovery action for `action2` performs `update_master_state(true); check_backups();` **checkpoint**(*action*2). This ensures that local state is replicated to remote masters immediately and causes a new place to be added before resuming operation of `action2`.

In the original WAMW, masters know about every other master when the computation starts and the set of masters does not change. When NAP2 is used, the set of masters changes in response to masters failing and new masters being added. Thus, the entire failover list is sent as part of the computation state to remote masters when `action2` terminates. Doing so ensures that remote masters can map (and thus retrieve) the results from a failed master to the places in its failover list.

Bootstrapping

Mobile agents are beneficial for resource discovery both before and during a grid computation [97, 178], since what resources are required and how to extract them may depend on the computation. In WAMW, before `action1` starts, the algorithm assumes that there exists a list of places for each local area network that are designated masters and workers. Establishing this list is a suitable task for a mobile agent. Given a list of candidate places, *candidates*, a mobile agent can **move** from place to place on the list as an itinerant computation. A backup place is used to detect that a place on the list has failed, and records this in a list, $failed$, adds a new place from the list of visited places and tries to **move** to the next place on the list. When all candidate places have been visited, the difference between the initial *candidates* list and the $failed$ list gives the places that can be used for the computation. If the initial *candidates* list is large, the list can be split into segments and a mobile agent computation started (using **init**) on a separate place for each segment.

An alternative approach to using an itinerant style computation is to have an initiating place i start agents in parallel at all places $c_1, ..., c_{n-1}, c_n$ in the initial *candidates* list and have the failover lists be of the form $< c_k, i >$ for all k in *candidates*. This way, i will be notified about all failing places in *candidates*, and unlike the itinerant style approach, the completion time of this approach does not depend on the number of failed entries in *candidates*. However, if the number of entries in *candidates* is large, i may suffer from message implosion when the mobile agents terminate their actions.

Pseudocode 8.2.1 Actions for WAMW masters.

```
def action1:
  $pmax\_interval$ = interval between each master state update
  $pmax$ = current_time() + $pmax\_interval$
  $idle\_workers$ = []
  $f$ = desired number of backups
  checkpoint($action2$)

def action2:
  $state\_update$ = false
  $type$, $workerid$, $result$ = get_worker_request()
  if $type$ == RENEW_LEASE:
    set_worker_lease($workerid$)
  if $type$ == TASK_REQUEST:
    $idle\_workers$.append($workerid$)
    if $result$ != NULL:
      $state\_update$ = true
      mark_task_as_done($result$)
  while len($idle\_workers$) > 0 and num_unallocated_tasks() > 0:
    $task$ = allocate_task()
    $workerid$ = $idle\_workers$.pop()
    set_worker_lease($workerid$)
    send_task_to_worker($task$, $workerid$)
    $state\_update$ = true
  update_master_state($state\_update$)
  check_leases()
  check_backups()
  if all_tasks_completed():
    terminate()
  checkpoint($action2$)
```

Splitting the initial *candidates* list into segments and starting an agent on a separate place for each segment reduces the amount of messages required to be processed by the initiating place.

8.3 Other Issues

Our practical approach to preserving the NAP2 properties specified in Section 6.4.1 is to adapt the failover list to the underlying network topology to decrease exposure to asynchrony. We now discuss alternative approaches to handle asynchrony, by using failure detector mechanisms that approximate the fail-stop model, by using consensus algorithms, and by using wide-area group communication.

8.3.1 Failure Detection

NAP2 is a protocol that assumes a failure detector that eventually detects all failures and never suspects a non-failed place as failed. Since the Internet is normally modeled as an asynchronous system, such semantics are impossible to implement, so an implementation of a failure detector may in practice make mistakes when deployed on the Internet. There are, however, failure detectors that make weaker assumptions on synchronous behavior. We now discuss two approaches to approximate fail-stop semantics [51, 153]. These approaches enable the implementation of a failure detector with semantics similar to a perfect failure detector in environments with weaker assumptions on synchrony.

Approximating the Fail-Stop Model

The work in [153] specifies a failure model that is *indistinguishable* from the fail-stop model in an asynchronous system. The approach is to simulate a fail-stop failure detector in an asynchronous system by extending the *happens-before* relation [102]. If a process p suspects a process q, then this suspicion must not happen-before a local event in q, meaning q can execute events after being suspected, but not events that are causally related to the suspicion of q. In simulated fail-stop, the happens-before relation defines that the crash of q happens-before the event where p suspects q. The approach works by having a process exchange a round of suspicion messages with a group of other processes. All non-failed processes declare the suspected process q as failed. If q has failed, it never receives the suspicion message. If q has not failed, the receipt of the suspicion message causes q to voluntarily crash. Simulated fail-stop thus hides that a process is suspected before it crashes by enforcing the happens-before relation on message channels. However, doing so requires that all messages are communicated on channels that are known to the system.

The work of [51] avoids the requirement of simulated fail-stop where all messages must be sent on channels that are known to the system. Instead, the approach is to *ensure* that a computer is crashed before being suspected, by using a hardware watchdog. The hardware watchdog resets the host computer when a threshold timeout has expired. Thus, if the threshold is not periodically increased, the computer crashes. Participants of the protocol grant each other leases [68] that define the threshold value. If a participant p fails to renew its lease with other participants, the computer that p executes on crashes. The participants execute a distributed snapshot protocol that ensures a participant is not suspected unless it has crashed. A similar approach is also described using process watchdogs instead of hardware watchdogs. The use of leases, like in WAMW, requires the timed-asynchronous model where clocks have rates close to real time.

Both these approaches would be useful in NAP2 to approximate the fail-stop model and thus help ensure that the NAP2 properties hold in a deployment of the implementation. Since all messages in NAP2 are communicated using TOS, the extended happens-before relation can be implemented, and the use of process watchdogs is probably not required for the mobile agent applications we have encountered so far. Both approaches, however, require a majority of correct processes to agree on the suspected process (i.e., $2f + 1$ processes are required for tolerating f failures). Hence, if failure detection is performed exclusively by places in the failover list, additional places are required to tolerate the same number of failures as the NAP2 implementation specified in the previous chapter. Additional places will also increase the cost of NAP2 in failure-free runs.

8.3.2 Consensus

By satisfying certain properties, failure detectors can enable solutions to problems such as consensus or atomic broadcast [33]. More specifically, consensus can be achieved even if the failure detector makes mistakes. Consensus involves two primitives *propose* and *decide*. A process invoking *propose*(x) proposes the value x and a process invoking *decide*(y) decides the value y. Consensus algorithms must enforce the following properties:

- No two correct processes decide on different values (agreement).
- A decided value must have been proposed (validity).
- Every correct process eventually decides on a value (termination).

Consensus has been employed by mobile agent systems before. A variation of consensus called DIV consensus is used to decide on the result from executing a mobile agent stage in the Fatomas mobile agent system [140]. For a protocol like NAP2, consensus can be used to reach agreement on the failover list and the corresponding mobile agent state. We discuss such an approach below, but before doing so we describe how consensus protocols operate with *u*nreliable failure detectors.

A failure detector that makes mistakes but is known to enable a solution for consensus is $\diamond\mathcal{P}$, an eventually perfect failure detector. Eventually perfect means that it satisfies strong completeness and eventual strong accuracy. Consensus with $\diamond\mathcal{P}$ works in rounds, where a coordinator process proposes a value that it tries to have accepted by other processes. If the coordinator is suspected as crashed, a new value may be proposed by another coordinator. To ensure agreement on a value, the majority of the deciding processes must be correct and decide. Thus, if there is a network partition then only the partition including the majority of the processes can decide, which means that the computation fails if the majority of the processes fail.

Paxos

Paxos [103] is a consensus algorithm that always guarantees safety, and satisfies liveness with very weak synchrony assumptions (e.g., using the Ω unreliable failure detector [32]). The protocol has been employed in several real systems, for instance the Chubby distributed lock service [29].

Like the $\diamond\mathcal{P}$-based consensus algorithm, Paxos works in rounds. The first round starts with a *proposer* choosing a unique number b, and sending this number to a group of *acceptors*. The acceptors receiving the number b check whether they have already received a higher number. If so, the proposal is rejected, and the proposer must restart the protocol with a higher number. If b is the highest numbered proposal seen by an acceptor, the acceptor sends a response back to the proposer with the highest numbered proposal the acceptor has accepted, and promises not to accept proposals less than b. If a majority of the acceptors accept the proposal, the proposer chooses a value v. This value is either the highest value from the acceptor responses, or if no value was provided, an arbitrary value chosen by the proposer. The next round then starts with the proposer sending b and v to the group of acceptors. The acceptors respond to the proposer by accepting or rejecting the proposal. An acceptor accepts the proposal unless it has already received any message with a proposal number greater than b. If an acceptor accepts b, it sends a message with b and v to a group of *learners*. A learner decides on the value v of the proposal if it receives such a message from a majority of the acceptors.

Paxos and NAP

We now outline a high-level approach for a protocol like NAP2 using Paxos. Assume that for a given failover list, each backup place in the list runs a *controller state machine*. Assume that n places comprise the controller state machine. All possible places that can execute an agent act as clients to this state machine, and execute in an initial state waiting for a reply from the controller state machine. The reply has two values: *execute* tells a client that it should execute the stage encoded in the reply message, and *failed* tells a client that a place has failed and that the recovery action encoded in the reply should be executed. Generating the *failed* reply requires that the controller state machine implements a failure detector.

The state of the controller state machine is the place p executing the current action, and the set of backups for p. Paxos is used to determine the membership of the controller state machine. Each place that runs the controller state machine plays the role as an acceptor, and the place p that executed action i acts as the proposer for the next action $i+1$, the place q for executing action $i+1$ and the backups for q. Hence, when an action terminates, p is assigned the consensus instance for action $i+1$, and proposes the membership for $i+1$ and q to the backups for i and $i+1$. If consensus is achieved, q is sent an *execute* reply to start executing action $i+1$.

Although detailed analysis of the approach just given is beyond the scope of this dissertation, we highlight some issues here. Paxos is based on a weaker model (i.e., the Ω failure detector [32]) than the fail-stop model of NAP2. For instance, proposed values are accepted if a *majority* of acceptors accept the value, which means that n must be $2f+1$ to tolerate f failures. If there is a network failure and there is a majority of non-failed controller state machine places within a connected component, there will be consensus on the controller state machine membership. If communication is non-transitive, two or more places may disagree on the failover list resulting in multiple proposals being sent. However, if there is a controller state machine place in common for all network components, consensus can still be achieved. A problem, though, is that if an unreliable failure detector such as Ω is used to generate the *failed* reply, the resulting protocol cannot satisfy property NAP2Elected, as specified in Section 6.4.5. For example, consider an execution of the protocol where the controller state machine consists of places $< a, b, c >$ and a place p is executing the current action. Ω states that there is a time where all correct processes always trust the same process. However, over time, places a, b and c may suspect p as failed although p has not failed. Hence, if one of the places in the controller state machine sends a *failed* response to a client when p has not failed, a redundant action will be executed.

Another problem with this approach is that it requires more messages to be sent than NAP2 does. However, the number of messages containing the entire payload of the agent will be the same as in NAP2 (e.g., by sending a checksum of the consensus value instead of the actual value in subsequent messages to the same place [104]), unless multiple proposals are required for consensus. Thus, for computations where the mobile agent state is sufficiently large, the latency overhead of the extra rounds of communication in Paxos may be insignificant compared to the NAP2 latency. An alternative would be to externalize the mobile agent state from the consensus algorithm and run a reliable broadcast protocol with the mobile agent state once consensus has been reached on the failover list. However, as observed in [140], separating reliable broadcast from the consensus algorithm adds complexity to the protocol. Additional complexity can make Paxos difficult to implement correctly. While the basic algorithm is simple to express at a high level, an implementation of a Paxos-style protocol may require several thousand code lines [29].

8.3.3 Wide-Area Group Communication

A protocol like NAP2 can be implemented by abstractions with weaker assumptions than consensus, for instance by a wide-area group communication system [95, 120, 127, 149, 150]. Group communication systems have different approaches to handling network instability, but a common problem is that during network instability, the group communication system will output views that are obsolete shortly after delivery. Processing obsolete views is expensive for the group communication system and for an application that rely on virtual synchrony if the application requires computational state to be communicated upon view changes. The Moshe group communication system [95] minimizes the amount of obsolete views when the network is unstable by delaying view delivery until the network is stable. Recall from Section 3.2.1 that Moshe uses membership servers that communicate the current network topology view among them. When membership servers disagree on the topology, Moshe enters a state where no subsequent view updates are sent to group members until the membership servers agree on the network topology. While this causes significant blocking, as we experienced in Section 3.4.2, agreement on views allows us to design a protocol like NAP2 that helps preserve the exactly-once property during times of asymmetric or non-transitive communication.

We will now outline how a protocol like NAP2 can be realized with a view synchronous group communication system like Moshe. Assume that initially all places that will potentially be visited by a mobile agent are running Moshe and that these places are members of a group called *NAP*. When an agent starts, a set of places must be added to its first failover list. To do so, an *init* message is reliably broadcast by the agent to the *NAP* group. The message contains a failover list $r(1)$, the agent code and its state, the number of crash failures to tolerate f, and a globally unique computation identifier g. Assume that the agent does not crash, so all non-failed members of NAP receive the *init* message. When a place p in the failover list $r(1)$ receives the *init* message, p joins a group g named by the computation identifier. As members of the NAP group join g, view updates will be sent to the existing members of g. When all non-failed places on the failover list $r(1)$ have joined g, the first non-failed place in $r(1)$ starts the action to be executed. When the action terminates, a new failover list $r(2)$, the agent code and its state, f, and the computation identifier g are reliably broadcast to NAP as a new *init* message, and the procedure is repeated. When the new failover list $r(2)$ is received by the places in $r(1)$, places in g that are not in the failover list $r(1)$ leave g when f members of failover list $r(2)$ have joined the group g. The first place in $r(2)$ can then start executing the current action when all places in $r(1)$ that are not in $r(2)$ has left g (to preserve property NAP2Elected, as specified in Section 6.4.5).

If the first place of a failover list crashes, a group leave message will be sent to the remaining members of g, and the recovery action must be executed by the first non-failed place in the failover list when the subsequent group view is installed.

Although detailed analysis of the protocol just given is beyond the scope of this dissertation, we highlight some issues here. First, a benefit of using a group communication system is that the reliable broadcast strategy, if supported, is determined by the group communication system implementation. However, if the NAP group contains a large number of places, then broadcasting the state of the agent to all members when an action terminates is likely to be expensive. The algorithm can, however, be improved in several ways with respect to performance. Since all non-failed members of the NAP group receive each broadcast, only the state that has changed from the previous *init* message needs to be sent. A further improvement to this approach is to not include the agent and its state in the *init* message, but have existing members of g broadcast the state to new members as they join g.

Although an algorithm built on top of Moshe would help preserve the exactly-once property when there is asymmetric and non-transitive communication, a network partitioning will still cause redundant actions to be executed when each network clique form a new view since property NAP2Elected, as specified in Section 6.4.5, is violated. Avoiding redundant actions here would require either a primary-partition policy where one of the partitions decides to terminate the computation like in Isis [149] or block like in Phoenix [120]. Another approach is to require rollback support for the mobile agent actions that executed at the places running NAP, and abort the actions at the places that have been visited by the duplicate computation. This is similar to the approach taken by NetPebbles [128]. The latter approach, however, would also help ensure the exactly-once property in NAP2 computations without requiring a group communication system.

8.4 Summary

We started this chapter by discussing approaches to improve the performance of NAP2. We first discussed the assumption of linear broadcast, and observed that a flooding protocol that uses network-level multicast may perform better within a local area network. We then discussed the use of classic state optimization protocols such as pre-copy and lazy-copy. While pre-copy would only work for specific NAP2 computations, support for lazy-copy would be simpler to add as a general mechanism. We then discussed strategies for replica placement in distributed storage systems.

The next part discussed the applicability of NAP2. We showed that an algorithm that does not use mobile code, WAMW, with minor modifications can be executed in NAP2 and benefit from the added fault-tolerance without requiring mobility.

We ended the chapter by discussing other issues. First, we discussed our assumption on failure detection that does not make mistakes and discovered two alternatives that can be used by a NAP2 implementation. We then discussed the use of consensus protocols and weaker failure detectors, and discovered that an algorithm like Paxos can be used to implement a protocol like NAP2. Finally, we discussed the use of wide-area group communication such as Moshe as a mechanism for implementing a protocol like NAP2. While the Moshe group communication system blocks when the network is unstable, it can help enforce the exactly-once property by delaying obsolete views.

Chapter 9

Conclusion

The amount of computational resources available on the Internet is increasing. Effectively using these resources for distributed computations is challenging. *Computational grids* provide tools for structuring and deploying large-scale distributed computations on the Internet. One of the key problems in computational grids is *managing* the available computational resources, and tools based on mobile agents are being advocated to solve this problem. However, to be widely adopted, such tools must be robust towards failures in the grid environment and thus require effective mechanisms for mobile agent fault-tolerance.

9.1 Results

This dissertation has identified network awareness as a central property for mobile agent fault-tolerance in grid environments. In Section 1.3, we stated the thesis of this dissertation:

Network aware fault-tolerance provides similar benefits over transparent fault-tolerance that mobile agents provide over conventional distributed computing.

The benefit of mobile agents over conventional distributed computing is that a mobile agent computation can adapt to changes in the network topology by migrating to another place. Consequently, we have investigated whether network aware fault-tolerance schemes that allow adapting replicas to changes in the network are beneficial over fault-tolerant schemes with transparent replica management.

To evaluate our thesis, we started by investigating how network communication behaves in grid environments. More specifically, we studied the performance of executing two master-worker algorithms when deployed on the Internet. The first algorithm, AX, is optimal in terms of redundant task executions and requires a group communication system. We ran simulations of a wide-area group communication system called Moshe on top of Internet communication traces and experienced that the performance of Moshe depends on the amount of asymmetric and non-transitive communication. Running AX on the same traces revealed that while AX is optimal in terms of redundant task executions, the frequency of asymmetric and non-transitive communication causes

Moshe, and thus AX, to block significantly. We developed a second algorithm, WAMW, that instead of group communication uses message flooding combined with leases for synchronization of global state. Although WAMW has significantly higher message complexity than AX, simulation results show that it completes the same computations faster and does not execute more redundant tasks than algorithm AX. The first insight from this study is that non-transitive and asymmetric communication occurs frequently on the Internet and grows with the number of participating networks. The most important insight, however, is that our experience with WAMW shows that a network aware master-worker algorithm can perform better than Moshe and AX despite not being optimal in terms of redundant task executions.

We then devised a mobile agent fault-tolerance protocol called NAP2. NAP2 is based on the primary-backup approach where a set of backups monitors the progress of the place that executes the agent. Upon detecting that the agent has failed, a backup executes a user specified recovery action. In NAP2, backups are explicitly managed by the computation in a failover list. More specifically, NAP2 allows computations to change the failover list to better accommodate resource changes in the execution environment. NAP2 assumes the fail-stop model. Routing instability in the network may cause violations of the assumed timing bounds, leading to redundant invocations of recovery code. However, we observed that by adapting failover lists to the network topology we could minimize the communication that spans more than one local area network. Minimizing the time the failover list spans more than one local area network decreases asynchrony and communication latency among the places in the list. In addition, by ensuring failover lists do not span more than two local area networks, non-transitive communication can be avoided.

We then specified the properties of the NAP2 protocol and derived a specification of NAP2 that we implemented on the TOS mobile agent platform. Our NAP2 implementation uses a linear message forwarding approach, but the specification allows certain parts of the protocol to use any reliable message forwarding scheme, including flooding. Our implementation of NAP2 led to several optimizations of TOS, for instance, better handling of large messages. We evaluated the latency of TOS message communication and developed three estimator functions that predict the performance of different operations performed during execution of NAP2. Comparing the estimators against actual latency measurements showed that the estimators were accurate, within 2.9 milliseconds of the measured value.

In Section 1.3 we listed two properties that together determine the strength of the dissertation thesis: performance and applicability. We now discuss to what extent these properties have been fulfilled, and whether the thesis is confirmed, rejected or requires modification.

Performance. Network aware fault-tolerance as implemented by NAP2 allows a mobile agent computation to adapt fault-tolerance according to changes in the environment. However, the latency overhead of executing NAP2 may outweigh the performance benefits compared to performing the same operation in a transparent protocol. Our performance hypothesis was: *network aware fault-tolerance allows adapting to changes* *in the topology faster than transparent fault-tolerance.* In Section 7.4, we measured the performance of changing between two disjoint failover lists, which is a procedure that NAP2 computations use to adapt fault-tolerance when moving from one local area network to another. We discovered that the transparent fault-tolerance approach was faster than NAP2 when the number of places were fewer than 3 and the agent state is small. When the agent state size exceeds 64 kilobytes, NAP2 was faster at 3 or more places in the failover list. Thus, our performance hypothesis is not unanimously confirmed. However, the experiments were performed within a local area network. We conjecture that network aware fault-tolerance is faster than transparent fault-tolerance for fewer than 3 places in the failover list and small agent payload if communication spans more than a single network.

Applicability. The first requirement was to establish how communication performs in the environment we consider for grid computations. In Chapter 3, 4 and 5, we studied the behavior of wide-area group communication and two master-worker computations when executed on Internet communication traces. The results of this study guided our strategy for minimizing exposure to network partitions, asymmetric communication and non-transitive communication in NAP2, as shown in Section 6.7.

The second requirement was to show that NAP2 could be used for computations that do not migrate. More specifically, show that by explicitly managing failover lists, NAP2 can satisfy fault-tolerance requirements for computations that do not migrate. In Section 8.2.1, we modified the original WAMW algorithm to use fault-tolerant actions, and showed that the changes to do so are modest. The resulting WAMW algorithm uses NAP2 to replicate masters within a local area network. If a master fails, then a new backup for the remaining masters is added to the failover list, and the required level of fault-tolerance is maintained. By this, we have satisfied both applicability requirements of the thesis.

Thesis Fulfillment. To summarize, we have satisfied the two applicability requirements and confirmed scenarios where the performance is better with the network aware fault-tolerance protocol than with the transparent fault-tolerance protocol. The validity of our performance hypothesis depends on the number of places in the failover list and the size of the agent state, and the thesis statement thus needs to include these variables upon evaluation to be valid. However, as we have shown with both group communication and itinerant computations, it is difficult to provide transparent fault-tolerance that scales well with the number of networks. Our experience with NAP2 and WAMW shows that network aware fault-tolerance is likely to perform better.

9.2 Limitations

There are limitations to our solution. NAP2 requires that parts of the protocol uses a linear message forwarding protocol, which causes suboptimal performance compared to network-level broadcast protocols such as IP multicast. In addition, the latency of linear forwarding grows linearly with the size of the agent state and the number of places in the failover list.

We have only used NAP2 to adapt failover lists to the network topology when migrating from one local area network to another. Determining the local area network of a place is done by statically comparing the IP-addresses of the places in the list. There are, however, other replica placement policies based on latency predictions or measured communication latencies that may give better performance than our approach in practice.

Our assumptions on synchronous behavior may be problematic when failover lists span more than one local area network for longer periods, and cause redundant recovery actions to be executed. While adapting to the network topology reduces the probability of asynchronous behavior, NAP2 still only gives probabilistic guarantees on exactly-once behavior.

9.3 Future Work

The Dynamic Enterprise Bus [87] is a prototype enterprise information middleware developed to investigate issues related to autonomous behavior such as self-configuration, self-optimization and self-healing. One of the use-cases for this middleware is a data processing system based on our experience in enterprise search. The computational model operates with a data flow going through a sequence of stages, where stages are interconnected as a directed acyclic graph. The project is currently evaluating the use of NAP2 as a mechanism to deploy and perform lifecycle management of graphs and stages in a fault-tolerant manner.

We are also further investigating the use of network aware fault-tolerance for applications that are not mobile, but rather require dynamic replica placement to increase read/write performance of servers [2, 183]. Our goal is to develop a library of replica placement strategies that developers using NAP2 can include as part of their computations.

More recently, *cloud computing* [7] has emerged as a complement to grid computing. Hence, we are also investigating the applicability of mobile agents in cloud environments and specifically whether NAP2 can be leveraged in cloud computing.

References

[1] Anurag Acharya, M. Ranganathan, and Joel H. Saltz. Sumatra: A language for resource-aware mobile programs. *Selected Presentations and Invited Papers for the 2nd International Workshop on Mobile Object Systems - Towards the Programmable Internet*, pages 111–130, 1997.

[2] Swarup Acharya and Stanley B. Zdonik. An efficient scheme for dynamic data replication. Technical Report CS-93-43, Brown University, Providence, RI, USA, 1993.

[3] Gul Agha and Carl Hewitt. Concurrent programming using actors. In *Object-oriented concurrent programming*, pages 37–53. MIT Press, Cambridge, MA, USA, 1987.

[4] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient overlay networks. *SIGOPS Operating System Review*, 35(5):131–145, 2001.

[5] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home: An experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[6] James P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, USAF Electronic Systems Division, Hanscom Air Force Base, October 1972.

[7] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, February 2009.

[8] Yeshayahu Artsy and Raphael Finkel. Designing a process migration facility: The Charlotte experience. *IEEE-CS Computer*, 22(9):47–56, 1989.

[9] Algirdas Avižienis, Jean-Claude Laprie, and Brian Randell. Dependability and its threats: A taxonomy. In *Proceedings of the 18th World Computer Congress*, pages 91–120. IFIP, Kluwer Academic Publishers, August 2004.

[10] Ozalp Babaoglu, Alberto Bartoli, and Gianluca Dini. Enriched view synchrony: A paradigm for programming dependable applications in partitionable distributed sytems. In *IEEE Transactions on Computers*, volume 46(6), pages 642–658, June 1997.

[11] Ozalp Babaoglu, Alberto Bartoli, and Gianluca Dini. Programming partition-aware network applications. In *Advances in Distributed Systems, Advanced Distributed Computing: From Algorithms to Systems*, pages 182–212. Springer-Verlag, 1999.

[12] Ozalp Babaoglu, Renzo Davoli, Alberto Montresor, and Roberto Segala. System support for partition-aware network applications. *SIGOPS Operating System Review*, 32(1):41–56, 1998.

[13] Omar Bakr and Idit Keidar. Evaluating the running time of a communication round over the internet. In *Proceedings of the 21st Symposium on Principles of Distributed Computing*, pages 243–252. ACM Press, 2002.

[14] Mahesh Balakrishnan, Stefan Pleisch, and Ken Birman. Slingshot: Time-critical multicast for clustered applications. In *IEEE Network Computing and Applications*. IEEE Computer Society, 2005.

[15] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th Symposium on Operating Systems Principles*, pages 164–177. ACM Press, 2003.

[16] Daniel J. Barrett and Richard E. Silverman. *SSH, The Secure Shell: The Definitive Guide*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[17] Forest Baskett, John H. Howard, and John T. Montague. Task communication in DEMOS. In *Proceedings of the 6th Symposium on Operating Systems Principles*, pages 23–31. ACM Press, 1977.

[18] J. Baumann, F. Hohl, K. Rothermel, and M. Strasser. Mole—concepts of a mobile agent system. In *Mobility: processes, computers, and agents*, pages 535–554. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.

[19] Judy I. Beiriger, Hugh P. Bivens, Steven L. Humphreys, Wilbur R. Johnson, and Ronald E. Rhea. Constructing the ASCI computational grid. In *Proceedings of the 9th International Symposium on High Performance Distributed Computing*, page 193. IEEE Computer Society, 2000.

[20] Francine Berman, Richard Wolski, Henri Casanova, Walfredo Cirne, Holly Dail, Marcio Faerman, Silvia Figueira, Jim Hayes, Graziano Obertelli, Jennifer Schopf, Gary Shao, Shava Smallen, Neil Spring, Alan Su, and Dmitrii Zagorodnov. Adaptive computing on the grid using AppLeS. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):369–382, 2003.

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

[22] Lorenzo Bettini. A Java package for transparent code mobility. In *Proceedings of the International Workshop on scientific engineering of distributed Java applications*, volume 3409 of *LNCS*, pages 112–122. Springer, 2004.

[23] Ranjita Bhagwan, Stefan Savage, and Geoffrey Voelker. Understanding availability. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems*, February 2003.

[24] Walter Binder, Giovanna Di Marzo Serugendo, and Jarle Hulaas. Towards a secure and efficient model for grid computing using mobile code. In *Proceedings of the 8th ECOOP Workshop on Mobile Object Systems: Agent Application and New Frontiers*, Malage, Spain, June 2002.

[25] Kenneth P. Birman. *Building secure and reliable network applications*. Manning Publications Company, Greenwich, CT, USA, 1997.

[26] Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal multicast. *ACM Transactions Computer Systems*, 17(2):41–88, 1999.

[27] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984.

[28] Navin Budiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Primary-backup protocols: Lower bounds and optimal implementations. Technical Report UW-CS-TR-1346, Cornell University, Computer Science Department, 1992.

[29] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 24–24, Berkeley, CA, USA, 2006.

[30] Luca Cardelli and Andrew D. Gordon. *Formal methods for distributed processing: a survey of object-oriented approaches*, chapter Mobile Ambients, pages 198–229. Cambridge University Press, New York, NY, USA, 2001.

[31] Miguel Castro, Peter Druschel, Y. Charlie Hu, and Anthony Rowstron. Exploiting network proximity in distributed hash tables. Technical Report MSR-TR-2002-82, Microsoft Research, 2002.

[32] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43:685–722, July 1996.

[33] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[34] Andred Chasin. The Gnutella protocol specification version 0.41. In *Clip2 Distributed Search Solutions*, June 2001.

[35] David Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, 1988.

[36] David M. Chess, Colin G. Harrison, and Aaron Kershenbaum. Mobile agents: Are they a good idea? In *Selected Presentations and Invited Papers Second International Workshop on Mobile Object Systems - Towards the Programmable Internet*, pages 25–45. Springer-Verlag, 1997.

[37] Jacky Chu, Kevin Labonte, and Brian Neil Levine. Availability and locality measurements of peer-to-peer file systems. In *Proceedings of the ITCom: Scalability and Traffic Control in IP Networks II Conference*, volume SPIE 4868, pages 310–321, July 2002.

[38] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation*, pages 273–286, May 2005.

[39] D. E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young. Computing as a discipline. *Communications of the ACM*, 32(1):9–23, 1989.

[40] Flaviu Cristian and Christof Fetzer. The timed asynchronous distributed system model. In *Proceedings of the IEEE Transactions on Parallel and Distributed Systems*, pages 642–657, June 1999.

[41] M. Dalmeijer, E. Rietjens, D. Hammer, A. Aerts, and M. Soede. A reliable mobile agents architecture. In *Proceedings of the 1st IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 64. IEEE Computer Society, 1998.

[42] Raphael Y. de Camargo, Fabio Kon, and Renato Cerqueira. Strategies for checkpoint storage on opportunistic grids. *IEEE Distributed Systems Online*, 7(9):1, 2006.

[43] X. Defago, A. Schiper, and N. Sergent. Semi-passive replication. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, page 43. IEEE Computer Society, 1998.

[44] Peter J. Denning. Is computer science science? *Communications of the ACM*, 48(4):27–31, 2005.

[45] Christophe Diot, Brian Neil Levine, Bryan Lyles, Hassan Kassem, and Doug Balensiefen. Deployment issues for the IP multicast service and architecture. *IEEE Network*, 14(1):78–88, 2000.

[46] Mark W. Eichin and Jon A. Rochlis. With microscope and tweezers: An analysis of the Internet virus of november 1988. In *Proceedings of the IEEE Computer Society Symposium on Security and Privacy*, Oakland, Ohio, 1989.

[47] Elmootazbellah N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Survey*, 34(3):375–408, 2002.

[48] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. Manetho: Transparent roll back-recovery with low overhead, limited rollback, and fast output commit. *IEEE Transactions on Computing*, 41(5):526–531, 1992.

[49] Nick Feamster, David G. Andersen, Hari Balakrishnan, and M. Frans Kaashoek. Measuring the effects of Internet path faults on reactive routing. In *Proceedings of the SIGMETRICS international conference on measurement and modeling of computer systems*, pages 126–137. ACM Press, 2003.

[50] Alan Fekete, Nancy Lynch, and Alex Shvartsman. Specifying and using a partitionable group communication service. *ACM Transactions on Computer Systems*, 19(2):171–216, 2001.

[51] Christof Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Transactions on Computers*, 52(2):99–112, 2003.

[52] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[53] Gary W. Flake. *The Computational Beauty of Nature: Computer Explorations of Fractals, Chaos, Complex Systems, and Adaptation*. The MIT Press, January 2000.

[54] Sally Floyd, Van Jacobson, Steve McCanne, Ching-Gung Liu, and Lixia Zhang. A reliable multicast framework for light-weight sessions and application level framing. *SIGCOMM Computer Communication Review*, 25(4):342–356, 1995.

[55] Ian Foster. Internet computing and the emerging grid. *Nature Web Matters (http://www.nature.com/nature/webmatters/grid/grid.html)*, December 2000.

[56] Ian Foster, Jerry Gieraltowski, Scott Gose, Natalia Maltsev, Edward N. May, Alex Rodriguez, and Dinanath Sulakhe et al. The Grid2003 production grid: Principles and practice. In *Proceedings of the 13th International Symposium on High Performance Distributed Computing*, pages 236–245. IEEE Computer Society, 2004.

[57] Ian Foster, Nicholas R. Jennings, and Carl Kesselman. Brain meets brawn: Why grid and agents need each other. In *Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 8–15. IEEE Computer Society, 2004.

[58] Ian T. Foster. The anatomy of the grid: Enabling scalable virtual organizations. In *Proceedings of the 7th International Euro-Par Conference on Parallel Processing*, pages 1–4. Springer-Verlag, 2001.

[59] Roy Friedman and Alexey Vaysburd. Fast replicated state machines over partitionable networks. In *Proceedings of the 16th Symposium on Reliable Distributed Systems*, page 130. IEEE Computer Society, 1997.

[60] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[61] Munehiro Fukuda and Duncan Smith. UWAgents: A mobile agent system optimized for grid computing. In *Proceedings of the 2006 International Conference on Grid Computing and Applications*, pages 107–113, 2006.

[62] Munehiro Fukuda, Yuichiro Tanaka, Naoya Suzuki, Lubomir F. Bic, and Shinya Kobayashi. A mobile-agent-based PC grid. *Proceedings of the Autonomic Computing Workshop*, pages 142–150, 2003.

[63] Lixin Gao and Jennifer Rexford. Stable Internet routing without global coordination. *IEEE/ACM Transactions on Networking*, 9(6):681–692, 2001.

[64] Eugene Gendelman, Lubomir F. Bic, and Michael B. Dillencourt. An application-transparent, platform-independent approach to rollback-recovery for mobile agent systems. In *Proceedings of the 20th International Conference on Distributed Computing Systems*, page 564. IEEE Computer Society, 2000.

[65] Chryssis Georgiou, Alexander Russell, and Alex A. Shvartsman. The Complexity of Distributed Cooperation in the Presence of Failures. In *Proceedings of the 4th International Conference on Principles of Distributed Computing*, pages 245–264, Paris, France, 2000.

[66] Chryssis Georgiou and Alex A. Shvartsman. Cooperative computing with fragmentable and mergeable groups. *Journal of Discrete Algorithms*, 1(2):211–235, 2003.

[67] Graham Glass. Objectspace voyager - the agent ORB for Java. In *Proceedings of the 2nd International Conference on Worldwide Computing and Its Applications*, pages 38–55. Springer-Verlag, 1998.

[68] Cary G. Gray and David R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 202–210, 1989.

[69] Robert S. Gray. Agent Tcl: A flexible and secure mobile-agent system. In M. Diekhans and M. Roseman, editors, *Proceedings of the Fourth Annual Tcl/Tk Workshop*, pages 9–23, Monterey, CA, 1996.

[70] Karl Taro Greenfeld. Meet the napster. In *TIME Magazine*, volume 156(4), October 2000.

[71] Timothy G. Griffin and Gordon Wilfong. An analysis of BGP convergence properties. In *Proceedings of the Applications, technologies, architectures, and protocols for computer communication*, pages 277–288. ACM Press, 1999.

[72] Andrew S. Grimshaw, Jon B. Weissman, Emily A. West, and Edmond C. Loyot Jr. Metasystems: An approach combining parallel processing and heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 21(3):257–270, 1994.

[73] Rachid Guerraoui and André Schiper. Software-based replication for fault tolerance. *IEEE-CS Computer*, 30(4):68–74, 1997.

[74] Krishna P. Gummadi, Richard J. Dunn, Stefan Saroiu, Steven D. Gribble, Henry M. Levy, and John Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proceedings of the 19th Symposium on Operating Systems Principles*, pages 314–329. ACM Press, 2003.

[75] Lei Guo, Songqing Chen, Zhen Xiao, Enhua Tan, Xiaoning Ding, and Xiaodong Zhang. Measurements, analysis, and modeling of BitTorrent-like systems. In *Proceedings of the 2005 Internet Measurement Conference*, pages 35–48, 2005.

[76] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In *Distributed systems (2nd Ed.)*, pages 97–145. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[77] E. Hammer-Lahav. The OAuth 1.0 protocol. RFC 5849, Internet Engineering Task Force, April 2010.

[78] Salim Hariri, C.S. Raghavendra, Yonhee Kim, Muhamad Djunaedi, Rinda P. Nellipudi, Ashok Rajagopalan, Prasad Vadlamani, and Yeliang Zhang. CATALINA: A smart application control and management. In *Proceedings of the NSF Active Middleware Services Workshop*, pages 43–55, 2000.

[79] Kevin Holley and Ian Doig. *Digital cellular telecommunications system (Phase 2+) (GSM); Use of Data Terminal Equipment - Data Circuit terminating Equipment (DTE - DCE) interface for Short Message Service (SMS) and Cell Broadcast Service (CBS) (GSM 07.05 version 6.0.0).* The European Telecommunications Standards Institute (ETSI), 1997.

[80] John B. Horrigan. Home broadband adoption 2006. *Report, PEW Internet & American Life Project*, May 2006.

[81] Christian Huitema. *Routing in the Internet (2nd ed.).* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[82] Geoff Huston. Interconnection peering and settlement - part I. *Internet Protocol Journal*, 2(1), March 1999.

[83] Geoff Huston. BGP routing trends in 2006. In *Proceedings of the 68th Internet Engineering Task Force Meeting*, March 2007.

[84] Jeremy Hylton, Ken Manheimer, Fred L. Drake, Jr., Barry Warsaw, Roger Masse, and Guido van Rossum. Knowbot programming: System support for mobile agents. In *Proceedings of the 5th International Workshop on Object Orientation in Operating Systems*, pages 8–13, Seattle, WA, USA, 1996.

[85] Kjetil Jacobsen and Dag Johansen. Ubiquitous devices united: enabling distributed computing through mobile code. In *Proceedings of the 1999 ACM Symposium on Applied Computing*, pages 399–404. ACM Press, 1999.

[86] Dag Johansen and Gunnar Hartvigsen. Convenient abstractions in stormcast applications. In *Proceedings of the 6th ACM SIGOPS European Workshop*, pages 11–16. ACM Press, 1994.

[87] Dag Johansen and Håvard Johansen. The dynamic enterprise bus. In *Proceedings of the 4th International Conference on Autonomic and Autonomous Systems*. IEEE Computer Society Press, Guadeloupe, France, 2007.

[88] Dag Johansen, Kåre J. Lauvset, Robbert van Renesse, Fred B. Schneider, Nils P. Sudmann, and Kjetil Jacobsen. A TACOMA retrospective. *Software: Practice and Experience*, 32(6):605–619, 2002.

[89] Dag Johansen, Keith Marzullo, Fred B. Schneider, Kjetil Jacobsen, and Dmitrii Zagorodnov. NAP: Practical fault-tolerance for itinerant computations. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, pages 180–189. IEEE Computer Society Press, 1999.

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

[91] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. An introduction to the TACOMA distributed system—version 1.0. Technical Report 95-23, University of Tromsø, June 1995.

[92] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. Operating System Support for Mobile Agents. In *Proceedings of the 5th Workshop Hot Topics in Operating Systems (HotOS)*, pages 42–45, Washington, USA, 1995.

[93] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. Supporting Broad Internet Access to TACOMA. In *Proceedings of the 7th SIGOPS European Workshop*, pages 55–58, Connemara, Ireland, 1996.

[94] Katarzyna Keahey and Von Welch. Fine-grain authorization for resource management in the grid environment. In *Proceedings of the 3rd International Workshop on Grid Computing*, pages 199–206. Springer-Verlag, 2002.

[95] Idit Keidar, Jeremy Sussman, Keith Marzullo, and Danny Dolev. Moshe: A group membership service for WANs. *ACM Transactions on Computer Systems*, 20(3):191–238, 2002.

[96] Carl Kesselman and Ian Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 1998.

[97] Klaus Krauter, Rajkumar Buyya, and Muthucumaru Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Software: Practice and Experience*, 32(2):135–164, 2002.

[98] Craig Labovitz, Abha Ahuja, Abhijit Bose, and Farnam Jahanian. Delayed Internet routing convergence. *IEEE/ACM Transactions on Networking*, 9(3):293–306, 2001.

[99] Craig Labovitz, Abha Ahuja, and Farnam Jahanian. Experimental study of Internet stability and backbone failures. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, page 278. IEEE Computer Society, 1999.

[100] Craig Labovitz, Abha Ahuja, Srinivasan Venkatachary, and Roger Wattenhofer. The impact of Internet policy and topology on delayed routing convergence. In *Proceedings of the 20th Joint Conference of the IEEE Computer and Communications Societies*, Anchorage, Alaska, April 2001.

[101] Craig Labovitz, G. Robert Malan, and Farnam Jahanian. Internet routing instability. *IEEE/ACM Transactions on Networking*, 6(5):515–528, 1998.

[102] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[103] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[104] Leslie Lamport and Mike Massa. Cheap paxos. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, pages 307–, Washington, DC, USA, 2004. IEEE Computer Society.

[105] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[106] Danny B. Lange and Mitsuru Oshima. Seven good reasons for mobile agents. *Communication of the ACM*, 42(3):88–89, 1999.

[107] Danny B. Lange, Mitsuru Oshima, Gunter Karjoth, and Kazuya Kosaka. Aglets: Programming mobile agents in Java. In *Proceedings of the International Conference on Worldwide Computing and Its Applications*, pages 253–266. Springer-Verlag, 1997.

[108] Stefan M. Larson, Christopher D. Snow, Michael R. Shirts, and Vijay S. Pande. Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology. *Computational Genomics*, 2002.

[109] Kåre J. Lauvset, Kjetil Jacobsen, Dag Johansen, and Keith Marzullo. Separating mobility from mobile agents. In *Proceedings of the 8th Workshop Hot Topics on Operating Systems (Position Summary)*, page 173, 2001.

[110] Kåre J. Lauvset, Dag Johansen, and Keith Marzullo. TOS: A kernel of a distributed systems management system. Technical Report 2000-35, University of Tromsø, Department of Computer Science, 2000.

[111] Kåre J. Lauvset, Dag Johansen, and Keith Marzullo. TOS: kernel support for distributed systems management. In *Proceedings of the 2001 ACM Symposium on Applied Computing*, pages 412–419. ACM Press, 2001.

[112] Meng-Jang Lin, Keith Marzullo, and Stefano Masini. Gossip versus deterministically constrained flooding on small networks. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 253–267. Springer-Verlag, 2000.

[113] Michael Litzkow and Miron Livny. Supporting checkpointing and process migration outside the UNIX kernel. In *Proceedings of the Winter USENIX Conference*, pages 283–290, San Francisco, CA, January 1992.

[114] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, June 1988.

[115] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.

[116] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufman, 1986.

[117] Priya Mahadevan, Dmitri Krioukov, Marina Fomenkov, Xenofontas Dimitropoulos, Claffy, and Amin Vahdat. The Internet AS-level topology: three data sources and one definitive metric. *SIGCOMM Computer Communication Review*, 36(1):17–26, January 2006.

[118] Grzegorz Greg Malewicz, Alexander Russell, and Alex Shvartsman. Optimal scheduling for disconnected cooperation. In *Proceedings of the 20th Symposium on Principles of Distributed Computing*, pages 305–307. ACM Press, 2001.

[119] Grzegorz Greg Malewicz, Alexander Russell, and Alexander A. Shvartsman. Distributed cooperation during the absence of communication. In *Proceedings of the 14th International Conference on Distributed Computing*, pages 119–133. Springer-Verlag, 2000.

[120] Christoph Malloth, Pascal Felber, Andre Schiper, and Uwe Wilhelm. Phoenix: A toolkit for building fault-tolerant, distributed applications in large scale. In *Proceedings of the Workshop on Parallel and Distributed Platforms in Industrial Products*, October 1995.

[121] Christophe Malloth and Andre Schiper. View synchronous communication in large scale networks. Technical Report TR95-92, University of Bologna, 1995.

[122] Z. Morley Mao, Lili Qiu, Jia Wang, and Yin Zhang. On AS-level path inference. In *Proceedings of the SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 339–349. ACM Press, 2005.

[123] Zhuoqing Morley Mao, Ramesh Govindan, George Varghese, and Randy H. Katz. Route flap damping exacerbates Internet routing convergence. In *Proceedings of the Applications, technologies, architectures, and protocols for computer communications*, pages 221–233. ACM Press, 2002.

[124] Dejan S. Milojicic, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Computing Surveys*, 32(3):241–299, 2000.

[125] Dejan S. Milojicic, Wolfgang Zint, Andreas Dangel, and Peter Giese. Task migration on the top of the Mach microkernel. In *USENIX MACH III Symposium*, pages 273–290. USENIX Association, 1993.

[126] Yaron Minsky, Robbert van Renesse, Fred B. Schneider, and Scott D. Stoller. Cryptographic support for fault-tolerant distributed computing. In *Proceedings of the 7th workshop on ACM SIGOPS European Workshop*, pages 109–114. ACM Press, 1996.

[127] Shivakant Mishra. *Consul: a communication substrate for fault-tolerant distributed programs*. PhD thesis, University of Arizona, Tucson, AZ, USA, 1992.

[128] Ajay Mohindra, Apratim Purakayastha, and Prasannaa Thati. Exploiting non-determinism for reliability of mobile agent systems. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks*, pages 144–156. IEEE Computer Society, 2000.

[129] George C. Necula. Proof-carrying code. In *Proceedings of the 24th Symposium on Principles of Programming Langauges*, pages 106–119, January 1997.

[130] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *IEEE-CS Computer*, 21(2):23–36, 1988.

[131] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. Modeling TCP throughput: a simple model and its empirical validation. In *Proceedings of the Applications, technologies, architectures, and protocols for computer communication*, pages 303–314. ACM Press, 1998.

[132] Holger Pals, Stefan Petri, and Claus Grewe. FANTOMAS: Fault tolerance for mobile agents in clusters. In *Proceedings of the 2000 Workshop on Parallel and Distributed Processing*, pages 1236–1247. Springer-Verlag, 2000.

[133] Vern Paxson. End-to-end routing behavior in the Internet. In *Proceedings of the Applications, technologies, architectures, and protocols for computer communications*, pages 25–38. ACM Press, 1996.

[134] Vern Paxson. *Measurements and Analysis of End-to-end Internet Dynamics*. PhD thesis, University of California Berkeley, 1997.

[135] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[136] Charles E. Perkins. Mobile networking through mobile IP. *IEEE Internet Computing*, 2(1):58–69, 1998.

[137] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of the HotNets–I*, Princeton, New Jersey, October 2002.

[138] Gian Pietro Picco. μcode: A lightweight and flexible mobile code toolkit. In *Proceedings of the 2nd International Workshop on Mobile Agents*, Lecture Notes in Computer Science, pages 160–171. Springer-Verlag, 1998.

[139] Sridhar Pingali, Don Towsley, and James F. Kurose. A comparison of sender-initiated and receiver-initiated reliable multicast protocols. In *Proceedings of the SIGMETRICS international conference on measurement and modeling of computer systems*, pages 221–230. ACM Press, 1994.

[140] Stefan Pleisch and Andre Schiper. FATOMAS - a fault-tolerant mobile agent system based on the agent-dependent approach. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks*, pages 215–224. IEEE Computer Society, 2001.

[141] Stefan Pleisch and Andre Schiper. Approaches to fault-tolerant and transactional mobile agent execution—an algorithmic view. *ACM Computing Survey*, 36(3):219–262, 2004.

[142] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. LOCUS: a network transparent, high reliability distributed system. In *Proceedings of the 8th Symposium on Operating Systems Principles*, pages 169–177. ACM Press, 1981.

[143] Michael L. Powell and Barton P. Miller. Process migration in DEMOS/MP. In *Proceedings of the 9th Symposium on Operating Systems Principles*, pages 110–119. ACM Press, 1983.

[144] Venugopalan Ramasubramanian and Emin Gün Sirer. Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, pages 8–8, Berkeley, CA, USA, 2004. USENIX Association.

[145] Richard F. Rashid and George G. Robertson. Accent: A communication oriented network operating system kernel. In *Proceedings of the 8th Symposium on Operating Systems Principles*, pages 64–75. ACM Press, 1981.

[146] Y. Rekhter and T. Li. A border gateway protocol 4 (BGP-4). RFC 1654, Internet Engineering Task Force, July 1994.

[147] Jennifer Rexford, Jia Wang, Zhen Xiao, and Yin Zhang. BGP routing stability of popular destinations. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 197–202. ACM Press, 2002.

[148] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. In *Proceedings of the 2004 Usenix Annual Technical Conference*, pages 10–10, June 2004.

[149] Aleta M. Ricciardi and Kenneth P. Birman. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, pages 341–353. ACM Press, 1991.

[150] Luis Rodrigues and Paulo Verissimo. xAMp: A multi-primitive group communications service. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 112–121, 1992.

[151] K. Rothermel and M. Strasser. A fault-tolerant protocol for providing the exactly-once property of mobile agents. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, page 100. IEEE Computer Society, 1998.

[152] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–350, 2001.

[153] Laura Sabel and Keith Marzullo. Simulating fail-stop in asynchronous distributed systems. In *Proceedings of the 13th Symposium on Principles of Distributed Computing*, page 399. ACM Press, 1994.

[154] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[155] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 377–390. ACM Press, 2002.

[156] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, 1983.

[157] Fred B. Schneider. Byzantine generals in action: implementing fail-stop processors. *ACM Transactions on Computer Systems*, 2(2):145–154, 1984.

[158] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[159] Fred B. Schneider. Towards fault-tolerant and secure agentry. In *Proceedings of the 11th International Workshop on Distributed Algorithms*, pages 1–14. Springer-Verlag, 1997.

[160] Fred B. Schneider, David Gries, and Richard D. Schlichting. Fault-tolerant broadcasts. *Science of Computer Programming*, 4(1):1–15, 1984.

[161] Aman Shaikh and Albert Greenberg. OSPF Monitoring: Architecture, Design and Deployment Experience. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, March 2004.

[162] Gary Shao, Francine Berman, and Rich Wolski. Master/slave computing on the grid. In *Proceedings of the 9th Heterogeneous Computing Workshop*, page 3. IEEE Computer Society, 2000.

[163] John F. Shoch and Jon A. Hupp. The "worm" programs — early experience with a distributed computation. In *Mobility: processes, computers, and agents*, pages 18–27. ACM Press/Addison-Wesley Publishing Company, New York, NY, USA, 1999.

[164] Flavio M. Assis Silva and Sven Krause. A distributed transaction model based on mobile agents. In *Proceedings of the 1st International Workshop on Mobile Agents*, pages 198–209. Springer-Verlag, 1997.

[165] Luis Moura Silva, Vitor Batista, and Joao Gabriel Silva. Fault-tolerant execution of mobile agents. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks*, pages 135–143. IEEE Computer Society, 2000.

[166] Larry Smarr and Charles E. Catlett. Metacomputing. *Communications of the ACM*, 35(6):44–52, 1992.

[167] Eugene H. Spafford. The Internet worm program: An analysis. *SIGCOMM Computer Communication Review*, 19(1):17–57, 1989.

[168] James W. Stamos and David K. Gifford. Remote evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–564, 1990.

[169] Rick Stevens, Michael E. Papka, and Terry Disz. Prototyping the workspaces of the future. *IEEE Internet Computing*, 07(4):51–58, 2003.

[170] Ion Stoica, Robert Morris, David Karger, Frans F. Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. *SIGCOMM Computer Communication Review*, 31(4):149–160, October 2001.

[171] Nils P. Sudmann and Dag Johansen. Adding mobility to non-mobile web robots. In *Proceedings of the ICDCS 2000 Workshop of Knowledge Discovery and Data Mining in the World-Wide Web*, pages F73–F79, 2000.

[172] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 1–14. USENIX Association, 2001.

[173] Jeremy Sussman and Keith Marzullo. The bancomat problem: an example of resource allocation in a partitionable asynchronous system. *Theoretical Computer Science*, 291(1):103–131, 2003.

[174] John K. Taylor and Cheryl Cihon. *Statistical Techniques for Data Analysis, Second Edition*. Chapman & Hall/CRC, 2004.

[175] Steffen Viken Valvåg, Åge Kvalnes, and Kjetil Jacobsen. POSH: Python object sharing. In *PyCon 2003*, 2003.

[176] Robbert van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 91–104. USENIX Association, 2004.

[177] Kannan Varadhan, Ramesh Govindan, and Deborah Estrin. Persistent route oscillations in inter-domain routing. *Computer Networks*, 32(1):1–16, January 2000.

[178] Sankararaman Venkatesh, Bapu Bindhumadhava, and Amrit Bhandari. Implementation of automated grid software management tool: A mobile agent based approach. In *Proceedings of the International Conference on Information & Knowledge Engineering*, pages 208–216. CSREA Press, 2006.

[179] C. Villamizar, R. Chandra, and R. Govindan. BGP route flap damping. RFC 2439, Internet Engineering Task Force, November 1998.

[180] Alan S. Wagner, Halsur V. Sreekantaswamy, and Samuel T. Chanson. Performance models for the processor farm paradigm. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):475–489, 1997.

[181] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *SIGOPS Operating System Review*, 27(5):203–216, December 1993.

[182] Jim White. Telescript technology: The foundation for the electronic marketplace. General Magic white paper. General Magic Inc. 1994.

[183] Ouri Wolfson, Sushil Jajodia, and Yixiu Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems*, 22(2):255–314, 1997.

[184] Jian Wu, Z. Morley Mao, Jennifer Rexford, and Jia Wang. Finding a needle in a haystack: Pinpointing significant BGP routing changes in an IP network. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation*, May 2005.

[185] Edward Zayas. Attacking the process migration bottleneck. In *Proceedings of the 11th Symposium on Operating Systems Principles*, pages 13–24. ACM Press, 1987.

[186] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications*, 2003.

Appendix A

Publications

This dissertation is based on the work presented in the following three publications:

Publication I

Kjetil Jacobsen, Dag Johansen. Ubiquotous Devices United: Enabling Distributed Computing Through Mobile Code. In *Proceedings of the 1999 ACM Symposium on Applied Computing (ACM SAC '99)*, pages 399–404.

In this paper we devise an architecture that enables cellular phones as interacting clients in a distributed system. Through GSM text messages, the cellular phone is used submit tiny programs called weather-alarms for execution at an extensible server running TACOMA. When weather alarms are triggered, a notification is sent back to the user with a GSM text message. The paper shows that mobile code is a convenient tool for integrating clients with extremely limited computational power and asymmetric network capabilities. The paper does not address mobile agent fault-tolerance, but motivates the need for such fault-tolerance given the limited possibility of the client performing recovery. The system is described in Section 2.2 as part of our background survey.

Publication II

Dag Johansen, Keith Marzullo, Fred B. Schneider, Kjetil Jacobsen, Dmitrii Zagorodnov. NAP: Practical Fault-Tolerance for Itinerant Computations. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS'99)*, pages 180–189.

In this paper we present the first version NAP. The paper was one of the first publications on mobile agent fault-tolerance based on the primary-backup approach. The work on the NAP protocol in this dissertation is based on the experiences from the first version of NAP.

Publication III

Kjetil Jacobsen, Xianan Zhang, Keith Marzullo. Group membership and wide-area master-worker computations. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS'03)*, pages 570–579.

Group communications systems have been designed to provide an infrastructure for fault-tolerance in distributed systems, including wide-area systems. In our work on master-worker computation for GriPhyN, which is a large project in the area of the computational grid, we asked the question *should we build our wide-area master-worker computation using wide-area group communications?* This paper explains why we decided doing so was not a good idea. The paper presents results from running the simulations of algorithm AX and WAMW, and shows that the degree of asymmetric and non-transitive communication is a significant factor for the performance of wide-area group communication. The main parts of this paper are described in Chapter 3, 4 and 5.

Other Publications

During the course of the study for this dissertation, the author also contributed to the following publications, which are related but not part of this dissertation:

- Dag Johansen, Kåre J. Lauvset, Robbert van Renesse, Fred B. Schneider, Nils P. Sudmann, Kjetil Jacobsen. A Tacoma Retrospective. In *Software: Practice and Experience*, Volume 32, Issue 6, 2002.
- Kåre J. Lauvset, Kjetil Jacobsen, Dag Johansen, Keith Marzullo. Separating Mobility from Mobile Agents. Position summary in *Proceedings of the Eight Workshop on Hot Topics in Operating Systems (HotOS-VIII'01)*, May 2001.
- Steffen Viken Valvåg, Åge Kvalnes, Kjetil Jacobsen. POSH: Python Object SHaring. In *PyCon DC 2003*, March 2003.

ISBN 978-82-8236-042-5