CONCEPTION AND IMPLEMENTATION OF A TOOLKIT FOR BUILDING FAULT-TOLERANT DISTRIBUTED APPLICATIONS IN LARGE SCALE NETWORKS

THÈSE N^{O} 1557 (1996)

PRÉSENTÉE AU DÉPARTEMENT D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Christoph Peter MALLOTH

Ingénieur informaticien diplômé EPF

originaire de Rossa (GR)

acceptée sur proposition du jury:

Prof. A. Schiper, directeur de thèse
Prof. Ö. Babaoğlu, corapporteur
Prof. B. Hirsbrunner, corapporteur
M. T. Jell, corapporteur
Prof. C. Petitpierre, corapporteur

Lausanne, EPFL
1996

Abstract

Large scale systems are becoming more and more common today. Many distributed applications are emerging that use the capability of world-wide internetworking. Since many applications require availability and consistency in the presence of failures, an adequate support for fault-tolerance is necessary. This can be provided by different paradigms and their implementations. Unfortunately, most of these implementations consider only local area networks, whereas this thesis describes a system, called Phoenix, which aims at large scale networks where additional types of failure have to be taken into account.

This thesis gives a complete description of Phoenix, a toolkit for building fault-tolerant, distributed applications in large scale networks. Fault-tolerance in Phoenix is achieved using replicated process groups, and consistency within one process group is achieved by using *view synchronous communication*. The particularity of Phoenix is the provision of this fault-tolerance and consistency in a large scale environment, where large scale is two-fold: (1) the wide geographical distribution of the replicated processes, and (2) a high number of participating processes in the system.

The description of Phoenix given here is based on its architecture. Each layer of Phoenix focuses on a particular problem and proposes a solution. Lower layers are responsible for the geographical large scale aspects and their problems, whereas higher layers provide high order communication and deal with numerical large scale aspects.

In large scale networks, in addition to the increased unpredictable latency of messages, communication protocols have to deal with link failures, which are often only transient. The *dynamic routing* layer in the Phoenix architecture tries to mask these link failures by rerouting. This rerouting not only gives increased reliability of communication, but also a more stable and accurate image of the reachability of the processes.

On top of the *dynamic routing* layer, the *reliable communication* layer provides *eventually reliable channels*, i.e. messages sent are eventually delivered at the destination provided that the sender and the destination processes are correct. This layer takes into account different parameters of large scale networks, such as (1) increased, unpredictable latency, and (2) non-negligible packet desequencing and (3) important packet loss.

The consistency among the replicas is based on a new implementation of the *virtually synchronous communication* paradigm. The implementation is part of the *view synchronous communication* layer and is based on a modified consensus protocol together with the *eventually reliable channels* of the *reliable communication* layer. The modified consensus protocol itself is

based on an unstable suspicion model, where incorrectly suspected processes can be considered alive at a later point. This will be exploited to make the protocol alive whenever a majority of replicas can communicate with each other. The situation where a distributed system is cut into smaller subsystems, and none of these subsystems contains a majority, is not uncommon in large scale, but is often only transient. Further, the *dynamic routing* layer already does a maximum to avoid this situation.

Based on the *view synchronous communication* layer, the *ordered multicast communication* layer provides different ordering primitives based on solid, theoretical definitions, allowing the implementation of different total and uniform orders.

The numerical large scale is considered by assigning different roles to the processes of a distributed system without leaving the context of groups. The idea is to concentrate the fault-tolerant aspect to a small set of core processes, whilst still guaranteeing convenient and efficient access semantics to processes outside these core processes.

Résumé

Les systèmes répartis à grande échelle sont de plus en plus courants aujourd'hui, et un grand nombre d'applications exploitent déjà les possibilités des réseaux de communication mondiaux interconnectés. Parmi ces applications, certaines doivent proposer un comportement cohérent en cas de pannes, ainsi qu'une disponibilité permanente nécessitant un support adéquat pour la tolérance aux fautes. Cette tolérance aux fautes peut être fournie par l'intermédiaire de différents paradigmes. Malheureusement, la mise en oeuvre de la plupart de ces paradigmes ne considère que des réseaux locaux, ou affaiblissent considérablement les garanties fournies par le système pour une utilisation à grande échelle. L'originalité du système Phoenix, décrit dans cette thèse, est de tenir compte de la grande échelle et des nouveaux problèmes introduits dans ce contexte, sans affaiblir les garanties.

Cette thèse donne une description complète de Phoenix, une boîte à outils conçue pour le développement d'applications réparties tolérantes aux fautes à grande échelle. La tolérance aux fautes est réalisée au moyen de groupes de processus dupliqués, et la cohérence entre les processus d'un groupe est garantie par le paradigme de la *communication virtuellement synchrone*. La particularité de Phoenix consiste à proposer de la tolérance aux fautes et de la cohérence dans un contexte à grande échelle. La grande échelle est vue à travers les deux dimensions suivantes: (1) la répartition géographique des processus dupliqués, et (2) un grand nombre de processus au niveau de l'application.

La description de Phoenix dans cette thèse est basée sur son architecture. Chaque couche de Phoenix se concentre sur un problème particulier et propose une solution. Dans cette architecture, les couches basses sont responsables des aspects liés à la grande échelle géographique, tandis que les couches hautes proposent des primitives d'ordonnancement, ainsi qu'une couche qui s'occupe de la grande échelle du point de vue du nombre de participants.

Dans des réseaux à grande échelle, mise à part le délai plus grand et imprévisible de la transmission de messages, un protocole de communication doit également s'occuper des pannes de liens, celles-ci n'étant souvent que de caractère temporaire. La couche de *routage dynamique* dans l'architecture de Phoenix tente de masquer ces pannes de liens en reroutant des messages. Ce reroutage ne propose pas seulement une communication plus fiable, mais aussi une information plus stable et précise de l'accessibilité d'autres processus.

La couche de *communication fiable*, mise en oeuvre au-dessus de la couche de *routage dynamique*, propose des canaux *finalement fiables*, qui assurent qu'un message est reçu par la destination à condition que l'émetteur et le récepteur soient corrects. En plus de la fiabilité, cette couche tient compte des différents paramètres influençant cette fiabilité, comme (1) un délai plus grand

et imprévisible, (2) un déséquencement non-négligeable de messages, et (3) la perte importante de paquets.

La cohérence des duplicas d'un service est basée sur une nouvelle mise en oeuvre du paradigme de la *communication virtuellement synchrone*. L'implémentation de ce paradigme est basée sur un protocole de consensus modifié, utilisant les canaux finalement fiables de la couche de *communication fiable*. Le protocole de consensus modifié se base sur un modèle de suspicion instable, c-à-d un processus qui a été suspecté à tort, peut être reconsidéré correct plus tard. Ceci est exploité par le protocole, garantissant le progrès à une majorité de processus pouvant communiquer. La situation où un système réparti est partitionné en plusieurs sous-systèmes et où aucun des sous-systèmes ne contient une majorité, est possible, mais en général temporaire. La couche de *routage dynamique* tente d'ailleurs d'éviter au maximum de telles situations.

La couche de *communication ordonnée* propose différentes primitives pour l'ordonnancement de messages. Ces primitives, toutes basées sur la couche de *communication virtuellement synchrone*, mettent en oeuvre des ordres totaux, uniformes ainsi que des combinaisons des deux.

Dans un système à grande échelle, le nombre de participants peut être considérable et sans support adéquat, la vivacité et la performance d'un tel système peut se dégrader rapidement. En affectant différents rôles aux participants, le problème peut être géré de manière efficace. L'idée de base consiste à concentrer les aspects de la tolérance aux fautes sur un petit sous-ensemble de processus, tout en garantissant aux autres processus un accès adéquat et rapide à l'information gérée par le sous-ensemble.

Acknowledgements

I am pleased to have this opportunity to express my gratitude to several people who have helped me in various ways during this work.

First I wish to thank my supervisor, Professor André Schiper, who has always been patient and open to discussions and questions. This study has been greatly influenced by his sense of accuracy, clarity and precision.

I am grateful to the members of the jury, Professor Özalp Babaoğlu, Professor Beat Hirsbrunner, Dipl. Ing. Thomas Jell and Professor Claude Petitpierre for the time they have spent applying their expert knowledge to examining this thesis.

I wish to thank Professor Boy Faltings for having presided the exam.

I have greatly appreciated working with current and former colleagues at the Operating Systems Laboratory, including our secretary Kristine Verhamme. I particularly wish to thank:

- Pascal Felber, Jean-Claude Lugeon and Uwe Wilhelm, who contributed to the development of the Phoenix toolkit;
- Dr. Rachid Guerraoui, who patiently read through a great number of versions of the present chapters, for his constructive remarks;
- Guillaume Nivat, Michel Schinz and Assia Doudou, for their contributions to the improvement of the Phoenix toolkit.

I am grateful to Dr. Hilary Martin and Jean-Pierre Exsteen for proof-reading the English manuscript.

As always I am thankful to my parents for their constant encouragement, confidence and support.

Finally I am particularly grateful to those who share my life: my wife Florence and my children Damian and Charline. Thank you for encouraging and supporting me in many ways throughout this work, and for your patience, love and understanding especially during my frequent absences (either physical or mental).

To Florence, Damian and Charline

Contents

List of Figures

Chapter 1

Introduction

Distributed systems are commonplace in the domain of local area networks. On the other hand with emerging world-wide interconnection of computers, e.g. using the Internet, large scale distributed systems become more and more important. Some examples are: cooperative editing systems that allow different sites throughout the world to work on the same document, air traffic control systems supervising a large territory, world-wide flight ticket reservation systems, etc. Common to all these systems is the fact that they provide some service to clients. If one thinks of a centralized implementation of such a system, where the service is running on one machine, the service becomes unavailable to clients as soon as this machine fails. In some cases, this unavailability is not problematic, as the machine will be repaired sooner or later. But, in some applications, e.g. air traffic control systems, availability is of great importance.

Availability is achieved by replicating a service over several sites rendering it fault-tolerant against the failure of single sites. The replication of the service also implies the replication of the information of the service, in order that the replicated service provides the same information as the centralized service. Consistency among the replicas of a service is achieved using adequate communication primitives in order to apply changes to every single copy of the replicated service. Consistency among the copies of a replicated service gives a client the possibility to send its request to the service as a whole. Changes in the configuration of the service, due to e.g. failures, are transparent to the client, leading to the permanent availability of the service.

Another source for unavailability are link failures. Although, link failures are almost absent in local area networks, they are common in large scale networks, as e.g. the Internet. For a participant it is not possible to distinguish a link failure from a link congestion, neither to distinguish the link failure from the failure of the destination, when communication is the only means to determine the failure. The only thing that a participant can notice is, whether it can communicate with the destination or not. This leads to another problem, which can be defined as the impossibility to determine how long one should wait for a given answer. Particularly in

large scale, communication can be extensively delayed and there is no bound on these delays. As consistency is guaranteed by communication, link failures can imply that this consistency can not always be guaranteed.

Another aspect of large scale, which a distributed system has to deal with, is the number of participants. The greater the considered network is, the greater the number of participants can be. Scalability is the key word, and is important in order to deal with huge numbers of participants.

The Phoenix[1] toolkit, described in this thesis, gives an answer to all the problems and questions stated above by letting a programmer build up a distributed system, including tolerance to process, site and link failures, support for large scale network and huge number of participants. The task of the programmer is facilitated by providing an object-oriented programming interface letting the programmer see its system as composed of distributed, fault-tolerant objects.

1.1 Thesis Contributions

This thesis shows the different problems occurring in large scale asynchronous distributed systems. The thesis identifies two aspects of large scale, the geographical large scale and the numerical large scale. Geographical large scale can influence the liveness of protocols guaranteeing consistency because of the asynchrony of communication and link failures. On the other hand, numerical large scale can influence scalability of such a system: without adequate concepts the system hardly scales to a high number of participants.

Consistency in the Phoenix system is provided through the means of the *view synchronous communication* paradigm. The thesis presents a new implementation of this paradigm based on a consensus protocol. This consensus protocol itself behaves well even in presence of increased latency or link failures, making the Phoenix system operable in geographical large scale. Other similar platforms are either not able to run in large scale (liveness is not guaranteed), or had to put restrictions on the consistency guarantees of the services provided. Phoenix goes in the way between providing at the same time (1) consistency and (2) ability to run in large scale, without imposing more restrictions than in a local area network context.

As the consistency is based on protocols with well-defined liveness guarantees, it gives Phoenix a solid theoretical framework.

Another problem of geographical large scale systems, is the problem of link failures and the

[1] In ancient Egypt, people associated the fabulous bird Phoenix with immortality; after the death and the immolation of an ancient Phoenix, a young Phoenix rose out of the ashes. The immortality can be translated to the system as the permanent availability of a service independent of the *death* of one of its processes.

resulting non-transitivity of communication. The thesis identifies the key problems and presents a simple solution which masks many of the problems that might occur.

The thesis contribution for numerical large scale systems consists in a framework, where not all processes have the same role with respect to a given group. The Phoenix system provides support for (1) core members, (2) clients and (3) sinks of a given group.

The thesis describes a complete architecture where each problem and its solution correspond to a layer. The architecture is completed by an object-oriented programming interface, allowing to write distributed, fault-tolerant applications. This platform, called Phoenix, has been implemented and a working prototype has shown the validity, the usefulness and the effectiveness of the design.

1.2 Thesis Organization

Chapter 2 presents the context in which the Phoenix toolkit is situated and shows the main aspects where Phoenix excels. This allows to compare the Phoenix toolkit with related work described in Chapter 3. In Chapter 4, an overview of the Phoenix architecture is given. The different layers in this architecture are described, and their interface specified. Chapters 5 to 10 are each dedicated to one of the layers of the Phoenix architecture, in the bottom-up order. Each chapter shows the basic ideas, concepts and algorithms for the implementation of the considered layer. Chapter 5 describes the *socket interface layer*, the lowest layer, in the architecture which is used for communication. It introduces also the event-driven programming paradigm which is pushed through all the overlying layers of the architecture. The next higher layer is the *routing layer*, described in Chapter 6. It deals with the communication problems occurring when link failures are common, but can be masked by rerouting messages. This builds the basis of the *reliable communication layer*, described in Chapter 7. Chapter 8 describes the most important layer in the Phoenix toolkit, the *view synchronous communication* layer. This layer guarantees consistency of a replicated service in case of failures by providing the adequate communication primitives. The protocols used in this layer guarantee a consistent state in spite of site and link failures. The *ordered multicast communication layer* (Chapter 9) provides different ordering criteria on top of the *view synchronous communication layer*, and this even in presence of failures. In Chapter 10, the application programming interface, including all underlying primitives provided with an object-oriented interface, is presented. Chapter 11 concludes the thesis.

Chapter 2

Phoenix Outline

2.1 Introduction

As introduced in Chapter 1, Phoenix is a toolkit[1] for programming fault-tolerant, large scale applications. The aim of this chapter is to describe the context of Phoenix and to show what fault-tolerance means in this context. In order to understand the next sections and chapters it is important to define the system model (Section 2.2) on which Phoenix is based. Then Section 2.3 describes the distributed aspects of Phoenix followed by Section 2.4 presenting the fault-tolerant aspects of Phoenix. Fault-tolerance is not a new research area, but Phoenix takes into account the large scale aspect not considered by other platforms. The large scale aspect is two-fold in Phoenix. We distinguish the geographical large scale and the numerical large scale, where the geographical large scale is the basis of the problems discussed in Section 2.5. Section 2.6 describes the numerical large scale and its influence on the availability and efficiency of fault-tolerant applications in the presence of huge groups. This chapter also introduces the discussion of the next chapter about the related work. All necessary items are introduced here in order to be able to compare Phoenix with other similar platforms.

2.2 System Model

The system model consists of a set of processes P. Processes of P can fail by crashing and later recover. The processes are entirely connected by a set of communication links L which will be described in Chapter 6. Communication over these links is asynchronous (i.e. there are no bounds on transmission delays), and unreliable (i.e. message can get corrupted or lost). Like to the processes, the communication links L can fail and be repaired after some time. This will be

[1] The toolkit actually consists in a daemon process and an application library.

further discussed in Section 2.5.

A group g is a subset of P, fulfilling some particular task. This group g can be addressed, independant of its composition, as a whole by other processes of P. As members of a group g can fail or get partitioned, and other processes of P can join the group, the group composition evolute in time. In order to be able to identify the composition of the group g at any moment, a new view V_i^g is defined each time the composition changes. As only one group at a time is considered in the following, the group identifier g has been omitted and the notation V_i is used as a unique identifier for the view as well as the ith set of processes defining the ith view of group g. Further, the installation of a view at the members of a group can be identified by the delivery of the *new view message* at these members.

A typical configuration suiting this system model are workstations connected through an Ethernet. The example for a large scale network is the Internet, where link failures commonly occur. Thus, the Internet is the ideal testbed for Phoenix.

2.3 Distributed System and Group Communication

A Phoenix distributed system is composed of several cooperating sites, where a site can host one or more processes. In order to coordinate some distributed task, these processes exchange messages using the underlying, asynchronous communication infrastructure. As this communication infrastructure is unreliable, reliability has to be build on top of it; reliability guarantees that, if there are no failures, a message sent from some process is eventually delivered at the destination.

A typical message exchange in distributed systems consists of the sending of a message to all or to a subset of the participating processes. Figure 2.1 illustrates the distribution of update information to a set of servers.

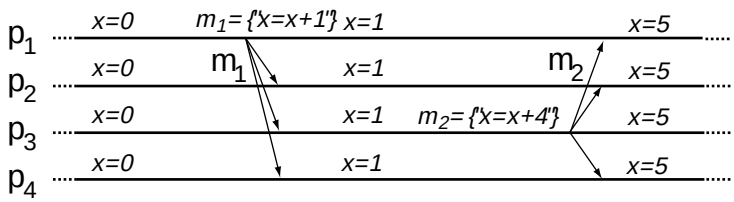

Figure 2.1: Group Communication

Subsets of processes exchanging messages using *one-to-many* semantics are called groups, and the processes in a group are called members of a group. The *one-to-many* send of some messages is called a *multicast*, and it is issued to a group. As it is the case for point-to-point communication, this multicast should be reliable, i.e. in the absence of failures, every process in the group

eventually receives the message[2]. However, in contrast to point-to-point communication, the reliability of a multicast is difficult to define, as shown in the next section.

2.4 Group Based Fault-Tolerance

Consider a centralized process providing some service. If the process or the site on which the process is running fails, the service becomes unavailable. The solution to keep the service available is to replicate the server process on several machines. The replicated server processes define a group. In contrast to the centralized case, the failure of one server does not imply that the service is unavailable. The replication of the server also implies the replication of the information provided by the server. Further, all updates to this information have to be applied to all servers, in order to guarantee that all servers provide the same consistent state as one single server would. In the case where no failures occur, the servers can inform each other about updates using multicasts to the group. These multicasts have to be reliable in order to guarantee that the updates are eventually delivered to every server of the group[3].

Consider the case where one of the servers fails during the multicast of some update information.

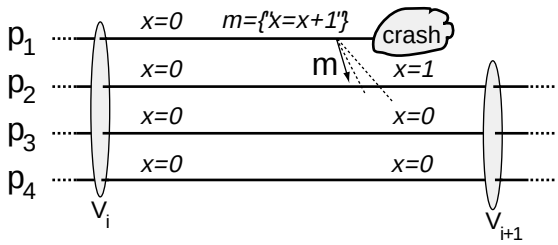

Figure 2.2: Crash of the Sender of a Reliable Multicast

Figure 2.2 shows the case where the reliable multicast is no longer sufficient for fault-tolerance and a stronger and more powerful primitive is needed. The update information from p_1 is only received by p_2, but not by p_3 or p_4. In order to continue the service, the remaining processes should further agree that p_1 is no longer part of the service; this allows one to disregard p_1 for further updates. The reliable multicast also lacks precise guarantees here. Once the set of remaining processes, called the *new view* V_{i+1}, have agreed on the exclusion of p_1, they also have to agree on the messages to deliver before continuing to serve other processes. In particular, p_3 and p_4 have to receive the message m received by p_2 in order to be consistent with p_2. The *virtually synchronous communication* paradigm, first introduced by [Birman 87], includes these conditions by considering views, defining sets of processes, and considering sets of messages

[2]We will see that this definition of *reliable multicast* is not sufficient. Chapter 8 will give an exact definition of *reliable multicast* including failures.

[3]Further, for consistency, it could be necessary that every process does the updates in the same order.

delivered in each view. The following informal definition summarizes the paradigm:

"Every process in the new view, has delivered
the same set of messages in the previous view"

The architecture of Phoenix is layered (see Chapter 4) and one of these layers provides a *view synchronous*[4] multicast primitive, allowing a process to multicast a message into a group with the *view synchronous* guarantee. This gives precise semantics to a reliable multicast in the presence of failures. A protocol, launched in case of problems, will guarantee that this *view synchronous communication* paradigm is satisfied when the protocol terminates. This is explained in detail in Chapter 8. Further, Phoenix allows processes to join and leave groups dynamically always guaranteeing consistency among the members of the group (view synchrony) using the same protocol. Besides increasing and decreasing the availability of a service this dynamic behavior of groups allows for example the migration of the service to some other machines if the shutdown of some machine is necessary. This avoids costly periods of down time and allows the service to permanently be available. Another use of the dynamic character of the groups is software upgrades. When a new version of a piece of software is available (assuming the same interface), a new process executing the new version can join the group. After the join of the new version, it is possible to shutdown a process executing an older version without any interruption of the service.

Besides the traditional fault-tolerance related to the crash of processes, Phoenix also considers link failures and partitions. Especially in large scale networks such as the Internet, link failures commonly occur. This is discussed in the next section.

2.5 Geographical Large Scale

A geographical large scale system can be defined as a distributed system whose components are distributed over more than one LAN or MAN. Often, components are in different towns, countries or even continents. In contrast to a local area network, built up with one sole fast and very reliable Ethernet, the large scale system can be composed of several local area networks. These local area networks are interconnected by sets of communication links, routers and gateways. The performance and the quality of these connections can vary from a public phone line with a modem to the most advanced Ethernets using FDDI and ATM [Fink 92, Leslie 93]. In any case, the multiple intermediate systems increase the probability of failure. For example, the common

[4]We replaced *virtually* by *view* as it suits better the paradigm defined. Moreover *virtual synchronous communication* and its definition in [Birman 87] includes causal and total order. *View synchronous communication* does not associate any order on messages besides the ordering of messages with respect to view changes.

interconnections are often shared with other services, and traffic cannot be foreseen. Thus, these interconnections are often subject to congestion and failure, which temporarily partition parts of large scale systems. As these failures are transient, after some time a congestion could have been absorbed or the failed link repaired or replaced, and normal traffic is reestablished.

Phoenix tries as much as possible to mask these link failures and link congestions, using other participating processes for relaying communication; this is discussed in Chapter 6. If masking is not possible, Phoenix always provides consistent state information despite partitions, called the *primary partition* property. This means there is always, for every group, only one set of processes representing the state of the group. The *primary partition* can be typically defined by considering that set of processes which contains a majority (there is always only one majority). Unfortunately, guaranteeing the *primary partition* can lead to a temporary blocking of the system, when no majority of processes able to communicate with each other can be defined. However, this blocking lasts only up to the point where a majority can be defined. Many other platforms, discussed in Chapter 3, do not deal with partitions, i.e. they block permanently when the *primary partition* is lost, or they circumvent this problem by not guaranteeing the *primary partition* property. In both cases, a group-wide consistent state can no longer be defined or has to be reestablished when the partition failure is repaired. This reestablishment of consistency is often a tedious task if not impossible, and is heavily dependent on the application and its semantics.

2.6 Numerical Large Scale

The second dimension of large scale in Phoenix is the numerical large scale. Phoenix provides support for distributed systems composed of a large number of participants. In order to understand better how Phoenix supports a large number of participants, the different roles of the processes in a distributed system are explained. In order to understand these roles the meaning of fault-tolerance has to be explained.

A system is fault-tolerant if it continues providing its functionality in spite of failures. Suppose that there are n copies of a server, which means that the service is fault-tolerant to $n-1$ failures. To ensure fault-tolerance 3, 4 or 5 servers are often enough to guarantee the availability of the service.

A process wishing to use a service provided by a group of processes is often only interested in part of the state information. It would be inefficient to add the process as a member to the group of the service in order for the process to receive the whole state information (allowing it to extract the information in which it is interested in). A process should have other ways to access a service, e.g.:

1.) the process is not a member of the group and communicates with the group by getting the identity of one or several of its members and by sending them requests;

2.) as in point 1.) above, but the process receives composition information of the group each time the composition of the group changes.

In solution 1 the process has absolutely no information about the group. Each time a process wants to access to the group's state information, it has to invoke some mechanism (e.g. a name server) in order to know who the members of a certain group are. The advantage of this solution is that it scales without any problems to a large number of processes, as no information is held inside the group about the processes requesting state information. The drawback of this solution is that the composition of the group can change over time, thus the process has to do the membership inquiry each time it wants to send a request to the group.

Solution 2 proposes a compromise between the solution of becoming a member and solution 1. In this solution the process subscribes to the group without joining it. Once subscribed, the process automatically receives updates of the composition of the group. This helps the process to keep track of the composition of the group. This information, coming from the group, allows the process at any moment to designate one of the members for issuing a request. This solution improves on the weaknesses of solution 1, but is still scalable, as the only information retained by the group is the set of processes which have subscribed. Phoenix provides support for this last solution and in this context the requesting processes are called *clients*.

Phoenix also provides a more primitive form of *clients*, called *sinks*, which can also subscribe to a group, but, in contrast to *clients*, they cannot directly issue requests to the group as they do not receive any composition information of the group. Group members keep a list of sinks which have subscribed and it is up to the group members, to send information to the sinks. Figure 2.3 gives a graphical illustration of the *members*, *clients* and *sinks* of a group.

A more detailed description of these roles can be found in [Babaoğlu 94] and the proposal for an object-oriented implementation in [Felber 95]. In Chapter 10, we will come back to these different *process types* when describing the application programming interface; sample applications are given in Appendix C.

2.7 Phoenix and the Consensus Problem

One important aspect of Phoenix, besides the numerical large scale, is the exploitation of recent results related to the solution of the consensus problem in asynchronous systems.

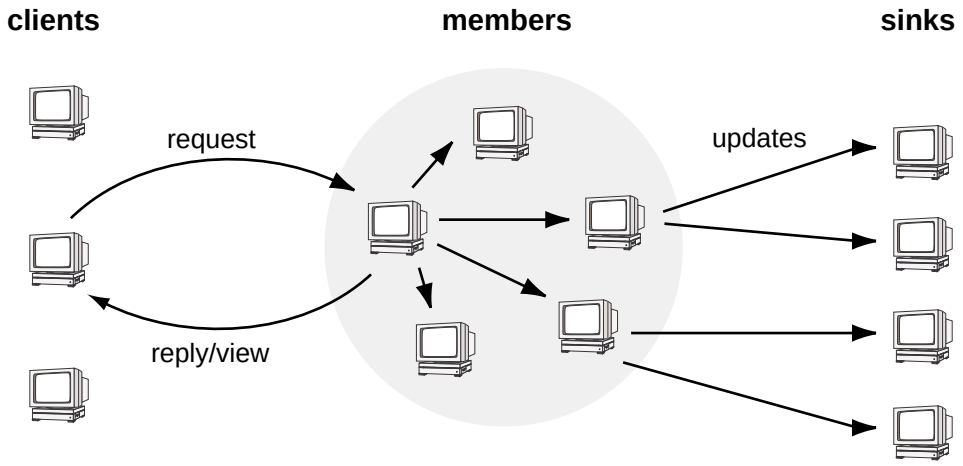

Figure 2.3: Phoenix Process Types

The Impossibility Result of Fischer, Lynch and Paterson

The impossibility result described in [Fischer 85] showed that it is not possible, in an asynchronous system, to solve consensus with a deterministic algorithm with only one faulty process. A priori it seems that the implementation of Phoenix is impossible, as an important part of Phoenix is based on consensus.

Unreliable Failure Detectors for Reliable Systems

In 1991, Chandra and Toueg [Chandra 95] show that by augmenting an asynchronous system with the unreliable failure detector $\diamond\mathcal{S}$ consensus becomes solvable. This result is not in contradiction with the Fischer-Lynch-Paterson impossibility result, as $\diamond\mathcal{S}$ is not implementable. However, the properties of $\diamond\mathcal{S}$ can be ensured with a high probability.

Contribution of the Phoenix toolkit

The main part of Phoenix, as will be shown in Chapter 8, is the protocol guaranteeing *view synchrony*. This protocol is based on the consensus protocol of [Chandra 95] which provides a solid, theoretical base to the Phoenix system. In spite of the impossibility result of [Fischer 85], it is possible with a high level of probability to have a live and operational system running by using a reliable communication layer providing eventually reliable channels. Further, a considerable effort is made in Phoenix to take partitions into account and to mask them as far as possible.

2.8 Conclusions

This chapter has given a short outline of the fault-tolerant and distributed aspects of Phoenix. The basic system model has been defined. Phoenix provides the infrastructure to implement

fault-tolerance by replication and is aimed at large scale distributed systems. Large scale is two-fold: the geographical large scale poses new kinds of problems related to link failures and the numerical large scale poses the problem of scalability.

Chapter 3

Related Work

3.1 Introduction

Fault-tolerance is not a new field of research and considerable development has already been done. In this chapter, Phoenix is compared to related work. One feature that is poorly supported by existing platforms is the suitability for large scale as described in the previous chapter. This is the context where Phoenix excels and where other platforms only provide weak support or no support at all.

This chapter describes work related to the Phoenix platform. First, reliable communication subsystems are described in Section 3.2, followed by group communication platforms in Section 3.3.

3.2 Reliable Communication Subsystems

3.2.1 TCP/IP

TCP/IP [Comer 88] is the standard for reliable point-to-point communication on today's networks, where reliable means *best-effort* communication, i.e. lost packets are retransmitted, but not indefinitely. Numerous services and protocols such as TELNET, FTP [Comer 88], RPC [Birrell 84], NFS [Sun Microsystems Inc 89] and HTTP [Marshall 95] are based on TCP/IP.

TCP/IP provides stream-oriented *first-in-first-out* delivery. Stream-oriented means that message boundaries are not respected and a single *message* could be split up and delivered in several packets at the destination. This can be a problem if message boundaries have to be respected, requiring a higher level mechanism to guarantee these boundaries.

TCP/IP is based on IP, which provides no reliability: messages are transmitted only once and

can get lost or corrupted. In order to provide reliability, TCP/IP retransmits lost messages, but it uses a rather simple mechanism to implement the retransmission. The acknowledgements, generated by the destination, contain the information stating up to which packet the sequence of received packets is complete. The sender retransmits packets, starting at the first non-acknowledged packet, no matter whether one or more of the packets to retransmit have already been transmitted and received by the destination. A sliding window protocol, responsible for flow and congestion control by varying the size of the window, allows more than one packet at the same moment in transmission.

Considering link failures or congestion, TCP/IP abandons the non-acknowledged packets after some timeout and kills the connection.

Considering connections, TCP/IP uses the socket interface of the underlying operating system. In typical systems, e.g. UNIX, the number of those sockets and their associated descriptors is limited by the system. Dynamically opening and closing connections would be a solution, but the heavy-weight character of the opening and closing of a connection would be a significant performance bottleneck in the system, when the number of active connections is large. This is further discussed in Chapter 7, where the reliable communication layer of Phoenix is presented.

3.2.2 RDP

The reliable datagram protocol (RDP), described in [Velten 84], tries to avoid some of the lacks of TCP/IP. The main difference compared with TCP/IP is that RDP keeps data boundaries; data is presented in the form of packets and packets are delivered as a whole; in the stream-oriented TCP/IP the flow of information can be cut at an arbitrary place in case of failure. Furthermore, RDP can allow delivery of packets without preserving order. Intended applications using RDP are bulk data transfers over networks with moderate loss rates.

Similarly to TCP/IP, RDP provides *best-effort* communication with abandon of the connection and the non-acknowledged messages. Thus, RDP addresses the performance issue of TCP/IP, but still does not consider the case of failure and recovery of a process. No useful semantics are defined in case of permanent or transient failures, a problem which is addressed by Phoenix (see Chapter 8).

3.2.3 NETBLT

The Network Bulk Transfer protocol [Clark 87] addresses the problem of the distribution of large amounts of data over communication channels where message loss is frequent. This protocol performs better than TCP/IP over these unreliable channels. NETBLT is not required to guarantee *first-in-first-out* delivery, but rather guarantees that eventually some bulk of data is

received at some destination. The key idea is to transfer only lost packets, where the receiver determines and informs the sender which packets have to be retransmitted.

The protocols NETBLT and RDP are very similar, and differ only in the flow control strategy. Similarly, NETBLT does not address explicitly the case of failure, but aims at improving performance of TCP/IP.

3.2.4 AFDP

Up to now, only protocols supporting point-to-point communication have been mentioned. The following protocols include support for one-to-many communication. AFDP [Cooperstock 95], standing for *Adaptive File Distribution Protocol*, addresses the need for efficient and reliable one-to-many communication, exploiting current multicast and broadcast possibilities. This protocol is mainly used for updating files on huge sets of machines in a local area network. The protocol is based on a publisher/subscribers model: one particular subscriber is designated as the secretary, responsible for managing group membership and reliability. The flow control depends on the slowest subscriber, limiting the throughput of the fast subscribers.

AFDP is the first protocol considering group communication by providing one-to-many communication. This one-to-many communication is based on multicast and broadcast primitives, if available, making the protocol faster than multiple point-to-point communications. In the absence of such one-to-many primitives, the protocol automatically switches back to point-to-point communications. Although it can be used in large scale, throughput will be poor if there are very slow sites involved in communication; the slowest link to a subscriber can be very slow in large scale[1], and this would slow down all overlying protocols.

Similar, to the previous protocols, AFDP does not really care about failures. The reader may identify the inherent problem of this protocol in case of the failure of the *secretary*.

3.2.5 MUTS

MUTS [van Renesse 92a] (acronym for Multicast Transport Service) is actually the communication subsystem used by Horus [van Renesse 92b, van Renesse 93] described in Section 3.3.4. MUTS provides a multitude of services to an overlying system: (1) memory management, (2) communication management, and (3) threading management. The most important part, for the concerns of Phoenix, is the communication management part, which implements reliable communication with *best effort* semantics. When communication problems occur, the commu-

[1] Values about 100 to 1000 times slower are usual when comparing local area network communication with communication in large scale.

nication management does not decide itself how proceed, but informs an overlying layer about these problems.

The threading management is defined by a generic interface which has been matched to different existing threading packages. MUTS has been successfully ported to common operating systems, e.g. SunOS, Solaris, and to micro-kernel architectures such as Mach [Accetta 86] and Chorus [Rozier 88].

3.3 Group Communication Platforms

3.3.1 RMP

The Reliable Multicast Protocol (RMP) [Whetten 94] provides view synchronous and total order multicast[2] to a process group, where for each of these multicast types a resilience value K can be specified, which requires that at least K destinations have received the message before delivering the message to a higher layer. Setting K at greater than the half of the participating processes provides primary partition semantics.

Moreover, RMP provides processes outside a group a means to issue so-called multi-RPCs to the group and to receive replies. This allows the building of scalable applications, where not all participating processes have to be members of the group in order to access some of its information.

Concerning group membership, RMP exploits the totally ordered multicast primitive to multicast membership changes, either for joins, leaves or failures. An adequate resilience value K, e.g. more than half of the processes of the group, allows that a new membership is only delivered if a majority has received it. Setting K at less than a half, allows concurrent views, but these cannot necessarily be merged again later (see next paragraph).

The failure detection mechanism is based on timeouts, and faulty suspected processes are removed. In case of a temporary partition, partitioned members cannot rejoin the initial group, but have to join as new members.

Reliability and total order are achieved by an extended rotating token protocol, initially introduced by Chang and Maxemchuck [Chang 84]. The protocol is based on a negative acknowledgement scheme. Each process can multicast messages, but only the token holder can order messages, by piggy-backing ordering information on its own multicast. A process receiving ordering information for a message that it has received, delivers that message. If the process receives

[2] RMP also provides multicast semantics with less *quality of service* (best-effort, unreliable), but these multicast semantics are no longer reliable.

ordering information for a message that has not been received, a request for retransmission of the message concerned is issued to the process which ordered the message.

RMP extensively exploits the IP multicast service [Deering 89], which provides good and scalable performance as multiple point-to-point communication is replaced by one single multicast[3]. On a LAN, throughput is almost independent of the number of participating processes.

3.3.2 Totem

Totem [Agarwal 92] is a protocol providing a total order multicast primitive. Totem provides this total order multicast with two different delivery guarantees:

- **agreed delivery**: a message m is delivered only if all messages which are multicast before m have been delivered.
- **safe delivery**: a message m is delivered only if all processes in the current configuration[4] have received the message m.

The protocol is based on the token circulating in a ring similar to [Chang 84], but the difference consists in the use of the token. In Totem, the token holder multicasts a message and at the same time passes the token to the next process in the ring; this is in contrast to [Chang 84] or RMP, where every process can multicast messages at any moment, but only the token holder can order multicasts. The way in which Totem uses the token combined with one-to-many communication primitives as broadcast and IP multicast, defines a natural order on the messages; the authors call it *born-ordered* total order in contrast to *sequencer ordered*.

Reliability is guaranteed using a sequence number in the token which is incremented each time a process multicasts a message. The message contains this sequence number. Each process receiving messages and the token, should have received all messages with sequence numbers smaller than the number in the token. If a process has missed one or more messages, it adds the sequence numbers of the missing messages to the retransmission list, which is another field of the token. Each time they receive the token the processes verify whether they are requested to retransmit any messages.

Messages kept for possible retransmission, are discarded after the point where the token has executed a full round without a retransmission request for any particular message.

In order to be alive a process which has no message to multicast passes the token to the next process without multicasting a message. In case of failure of the token holder, a new token is

[3] RMP is also operable in networks where only point-to-point communication is possible, but with a considerable reduction of performance.

[4] Totem configurations can be compared to the notion of views in other systems.

generated and the remaining processes reach agreement on the new configuration. Agreement on this next configuration is achieved, when the token has done a full round within this new configuration.

A major drawback of the Totem protocol is its poor performance if the load is asymmetrical or if there is a slow process in the ring. If there is one process in a ring generating a lot more traffic than other process in the same ring, the considered process has to wait for the token before being able to multicast the message. Another drawback of this solution is the permanent generation of traffic, due to the circulating token, even if processes have no messages to send.

Totem also has advantages. Besides the total order, the token approach in Totem provides an ideal way to control flow and buffer congestion. As only the token holder can multicast a message, there are no collisions (besides collisions with unrelated traffic on the network) and as each process can multicast one message when it receives the token, every process receives the same amount of network resources. This also gives Totem the ability to provide relatively weak, real time guarantees.

Extension to Multiple LANs

Initially, the Totem protocol was designed for local area networks where one-to-many primitives are available. In later versions, this restriction was released by having several rings, each with its associated token, and gateways between the rings responsible for forwarding messages from one ring to the other rings. Messages with some sequence number are injected into other rings with a new sequence number which is determined when the injecting gateway receives the token. In order to guarantee total order among several rings, the protocol uses information about the number of rings and delivers a message as soon as it has a message of every ring, and the considered message has the smallest time stamp of all rings.

As it is possible that some ring does not multicast messages for some time, the gateway is generating so-called *guarantee vectors* whose functions are two-fold: (1) from time to time they provide information concerning any ring in order to be able to deliver messages, and (2) they contain topology information in order to deal with failures or partitions.

Having several rings, Totem had to be made aware of partitions. The approach taken by the authors to overcome the problem of partitions is to allow each single ring to operate upon itself. The gateways are responsible for detecting a possible partition between rings and act accordingly. If partition occurs inside a ring, each partition forms a new ring. This can lead to a ring with only one singleton process. As a consequence Totem might not always be able to provide and guarantee a primary partition.

In order to give guarantees on the order of messages in case of partition, the authors have proposed an *extended virtual synchrony model* [Moser 94]. This model can be summarized by

the following sentence:

> *"Extended virtual synchrony allows two processes in different components of a partitioned system to deliver different messages, but does not allow them to deliver the same messages in different orders."*

This *extended virtual synchrony* can make sense for some applications, but if a consistent state has to be guaranteed, each partition has, at least, to keep all messages that processes in other partitions have to deliver. It is often impossible to merge several total orders to one single total order respecting each single total order, or in other words, partition merging is a difficult task and there are often no predefined methods to provide this merging. Horus (see Section 3.3.4) and Transis (see Section 3.3.6) have also included the ideas of *extended virtual synchrony*, and Section 3.3.6 about Transis gives concrete examples of how this feature can be employed.

3.3.3 Isis

The Isis toolkit [Birman 90] is the first platform which has implemented the *virtually synchronous communication* paradigm [Birman 87]. It provides *view synchronous* multicast with FIFO, causal and total order [Birman 91]. Like many earlier platforms, Isis was designed to work in a local area network, where only process and site failures are considered and latency of communication is small.

The membership protocol of Isis [Ricciardi 91] provides primary partition semantics, i.e. there is always only one set of processes defining the state of the group. The implementation of view synchronous communication is based on the *flush protocol* [Stephenson 91]. The membership protocol is unfortunately not resilient to partitions with only-minority views, as suspected processes are excluded even if they have not crashed. This leads to the permanent blocking of the protocol when such a situation occurs. This is discussed in detail in Section 8.7.2.

Isis introduces the notion of clients which join the group. An Isis client has the possibility to multicast messages to the group[5]. However, this multicast has no *view synchronous* semantics as the client can fail during this multicast. Thus, in order to be sure that a message is received by all members of a group, a coordinator of the group has to multicast the message again, this time using a multicast with *view synchronous* semantics[6].

[5] RMP designates this kind of multicast as a multi-RPC.

[6] However, *view synchronous* semantic of client multicasts is guaranteed in case of the failure of one of the members of the group, i.e. every process in the new view will have delivered the message multicast by the client.

3.3.4 Horus

Horus [van Renesse 92b, van Renesse 93] is inspired by the Isis toolkit and tries to overcome some of the problems of Isis. Also an effort has been made to restructure Isis in smaller, clearly defined and modular units. This modularization allows a programmer to include only the parts which are used by the applications. One of these modular units is MUTS described earlier; other modules implement view synchronous multicast, total order, etc. . The source of inspiration for this modular architecture has been upcoming micro kernel architectures where the kernel only provides an absolute minimum, and higher level services are implemented as server processes. Horus has thus been successfully ported to Mach [Accetta 86] as well as to other non-micro kernel architectures with few problems.

The heart of the Horus platform is the view synchronous communication module. Unfortunately, the protocol implemented in this module does not overcome the permanent blocking in case of only-minority partitions. In Horus, the problem is solved by letting an application make progress even in minority partitions, providing a bit to an application, indicating if the application is currently part of the primary partition or not. In the situation of only-minority partition, this bit is not set at any view of a group, and the primary partition is lost.

By defining concurrent views, Horus guarantees that the *view synchronous communication* paradigm is satisfied on a per view basis. The difficulty of having concurrent views, besides the possible loss of the primary partition, is the merging of concurrent views, when they are reconnected. Partition merging is a difficult task, especially when non reversible actions have been taken before the merge occurs. This is further discussed in Section 3.3.6 where the Transis platform is described.

3.3.5 Relacs

The *view synchronous* semantics proposed in the Relacs [Babaoğlu 95a] programming platform is different from the ones already presented. Similar to the Horus system it allows concurrent views, but Relacs puts a restriction on how new, concurrent views can be defined. In Relacs, the *view synchronous* semantics can be decomposed into the following components:

- quasi-strong partial membership service;
- view synchronous multicast message semantics based on the membership service above.

The *quasi-strong partial membership service* can be summarized by a membership service delivering views to sets of processes, where two concurrent views, resulting from the partitioning of

a common view may overlap only if one of the new views is a subset of the other. Symmetrically, there exists a restriction for the merging case that can be summarized by saying that two concurrent views can merge to a single one, if they do not intersect. For a complete description refer to [Babaoğlu 95c, Babaoğlu 95b]

Based on these definitions of concurrent views, Relacs defines the *view synchronous multicast* as follows: for each message m multicast by some process, if there exists a process $p \in V_i$ which delivers m, then for all new views V_j resulting from V_i due to a view change, all processes $q \in V_i \cap V_j$ deliver m.

Relacs is also confronted with the problem of partition and partition merging. Similarly to Horus, the primary partition property is not guaranteed and the authors of Relacs simply delegate the problem of merging the state or messages of two concurrent views to the application. [Babaoğlu 96] gives a good overview of the different scenarios an application programmer has to face when developing applications which should support state creation, transfer and merging.

In contrast to all other platforms and communication subsystems (besides Phoenix), Relacs identifies and considers link failures as a major problem and addresses resulting transitivity problems arising in large scale distributed systems. Phoenix and Relacs are, to our knowledge, the first fault-tolerant programming platforms which identify the transitivity problem.

Unfortunately, the minimal set of communication primitives provided by the Relacs platform, makes the design of Relacs applications a difficult task[7]. Relacs also considers the numerical large scale [Babaoğlu 94], already introduced in Section 2.6, and, in the minimal interface, provides adequate primitives to implement *core members*, *clients* and *sinks*.

3.3.6 Transis & Lansis

Transis [Amir 92b] is a programming platform providing fault-tolerant total order multicast to process groups. Similar to Horus and Relacs, Transis allows concurrent views, and thus has to live and to deal with loss of the primary partition and partition merging.

One particularity of the Transis platform (as well as Horus and Totem) is its ability to deal with partitions [Dolev 94]. As concurrent views are allowed, it is possible that a system becomes partitioned, leading to several concurrent views. After partition repair, the partitions will merge again defining a single view. This allows the system to make progress even if no primary partition is defined. The way partitions are merged makes it necessary to keep all *sensible* messages which

[7] Phoenix and Relacs are the fruit of a collaboration in the ESPRIT BRA project BROADCAST, but Phoenix adds on top of the common, minimal interface an object-oriented, threaded programming interface simplifying application programming. Further, due to different definitions of the *view synchronous* paradigm, different protocols are implemented.

have to be sent to other partitions once the partitions get connected again. This method has two major drawbacks:

- if partition holds for a long time, buffers for *sensible* messages can become quite big (this can be attenuated by involving secondary storage, but the problem remains);
- if it is necessary to have consistency on non-reversible actions, messages in minority partitions cannot be delivered.

The last point can be compared with the blocking of the message delivery until a primary partition can be defined.

The authors of Transis describe a few concrete applications, which can really exploit concurrent views and partition merging. Partition merging in a consistent way is often difficult, sometimes impossible. [Dolev 95] gives some examples which exploit these concurrent views and partition merging. These sample applications can be classified in two categories:

- either they require a majority in order that the application does not block, or
- the partition merging restricts itself to merging the memberships of the reconnected partitions.

The CoRel replication service [Keidar 95], or an *atomic commit protocol* called E3PH [Keidar 94] are examples of applications which require a majority to proceed, whereas [Amir 94] describe a logging service which requires only acknowledgement of membership changes (no requirements on the delivery of the messages). Another message logging service, called Persistent Replication Service Layer (PRSL), does not impose any order on the messages, and thus partition merging is trivial.

A last interesting point to mention is how a distributed system built with Transis starts up. On startup every process builds its singleton view. There are as many concurrent views as there are processes starting up. These concurrent, singleton views are merged until they form a single view containing all the processes.

Lansis

Transis uses a communication infrastructure called Lansis [Amir 92a], which provides an efficient reliable multicast primitive based on the broadcast service provided by local area networks. This broadcast service makes Lansis and thus Transis unusable in large scale. There are recent extensions of Lansis providing total order multicast over multiple broadcast networks, but if the number of processes per network is small or equal to one, this approach is rather inefficient.

The broadcast service used by Lansis is not reliable. Lost messages are recognized by piggy-backing on new messages acknowledgements of messages which have already been received. When another process receives such a message it can determine whether it has seen the same messages as the sender of the message received. Missed messages can be requested at processes which have acknowledged the message. In this way Lansis avoids the explicit sending of acknowledgements, but requires that every process from time to time multicasts a message, at least a multicast containing only acknowledgements.

3.4 Conclusions

This chapter has given a short overview of work related to Phoenix. Many recent platforms (Totem, Transis, Horus, Relacs) have identified the partition problem in large scale, but their solution can no longer guarantee the primary partition property[8]. The introduction of concurrent views in these platforms implies not only the loss of the primary partition property, but also the introduction of a new problem, the merging of partitions. As this merging cannot be generalized, a separate, often complex solution has to be developed for any particular application. Phoenix, in contrast, guarantees the primary partition property without loosing liveness. Further, particular mechanisms in the application programming interface allows the implementation of concurrent views, although the programmer has to deal with them explicitly.

None of the reliable communication subsystems addresses the problem of transitivity of communication in case of link failures, link failures which do not necessarily lead to partitions. In the fault-tolerant programming platforms, only Relacs, besides Phoenix, identifies the transitivity problem, but only Phoenix proposes a complete solution.

Another aspect neglected by many fault-tolerant programming platforms, is the multiple forms of scalability. Especially in numerical large scale, it is necessary to provide adequate support for dealing with huge number of participating processes. Phoenix not only proposes adequate architecture support for applications with a huge number of processes, but also provides the corresponding object-oriented programming interface.

Many recent platforms (Transis, Totem, Relacs but not Horus), provide only a simple, minimal programming interface. This often allows the implementation of fault-tolerant applications only at the process or even at one process per site level, making applications monolithic and hardly scalable. In Phoenix the grain of replication is the object, a very light-weight concept: a threading system allows several objects to share the same process and address space, leading to extended scalability.

[8]To summarize: a system allowing minority views to make progress, cannot guarantee to solve consensus.

Chapter 4

Overview of the Phoenix Architecture

4.1 Introduction

Phoenix is a programming platform allowing the provision of fault-tolerance by replication. Phoenix is divided into a set of layers which are used by the top layer application either in a direct or an indirect way. Information is passed to lower layers using procedure calls, whereas information from lower layers is delivered to higher layers using callbacks; these callbacks are installed from higher layers into lower layers upon startup of the layer. As all applications use the same underlying architecture, the common underlying parts have been separated from the application in order that multiple applications can benefit from one communication block, called the Phoenix daemon, implementing the core layers. Figure 4.1 shows the conceptual architecture with the application-specific layer in white and the core layers implemented in the daemon in gray.

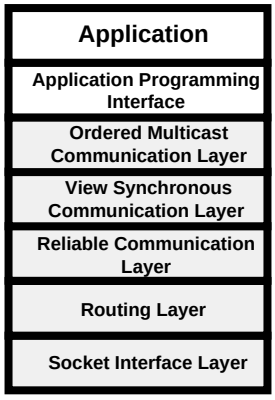

Figure 4.1: Conceptual Architecture

There is one daemon per site implementing the core layers of Phoenix and several applications can use it. Figure 4.2 shows a typical configuration of Phoenix application in a real system.

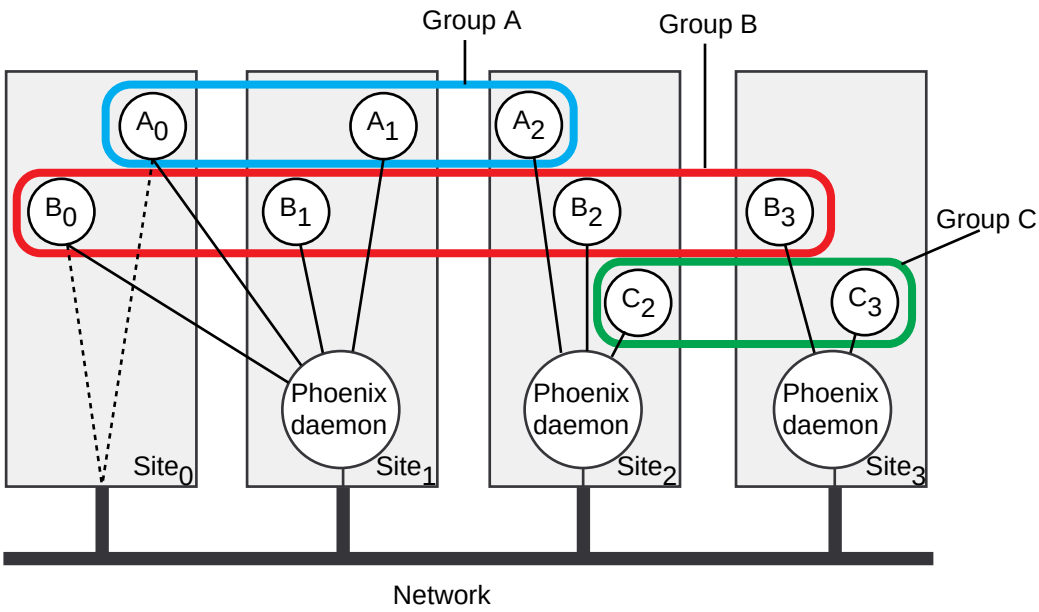

Figure 4.2: Phoenix System Configuration

Applications interact with the daemon by procedure calls, converted to message exchanges between the daemon and the applications. It is interesting that the communication between the daemon and an application uses the same reliable communication layer as the communication between the daemons. Figure 4.3 shows a real Phoenix system configuration with the various layers.

With this separation it is even possible to have an application on some site X using the daemon of site Y. This can be interesting for scalability, as not every application has to have its own daemon on its site. For fault-tolerance in a system composed of multiple local area networks, for example, it might be sufficient to have a small number of daemons (e.g. 3 to be fault-tolerant to daemon crash) per local area network, whereas there is no restriction on how many applications use these daemons. Another advantage of this architecture is that the daemon can be run on a *stable* machine[1], whereas applications can be run on any other machine. Such a solution also has drawbacks, for instance the failure of the daemon or of the machine on which the daemon is running implies the failure of all applications using this daemon. Another drawback is the indirect communication between applications through their daemons. This problem is discussed in Chapter 10, where the application programming interface and its implementation are discussed.

[1] A stable machine can be for example a dedicated machine where no user process executes, or a machine with a power backup (UPS) to resist power failures.

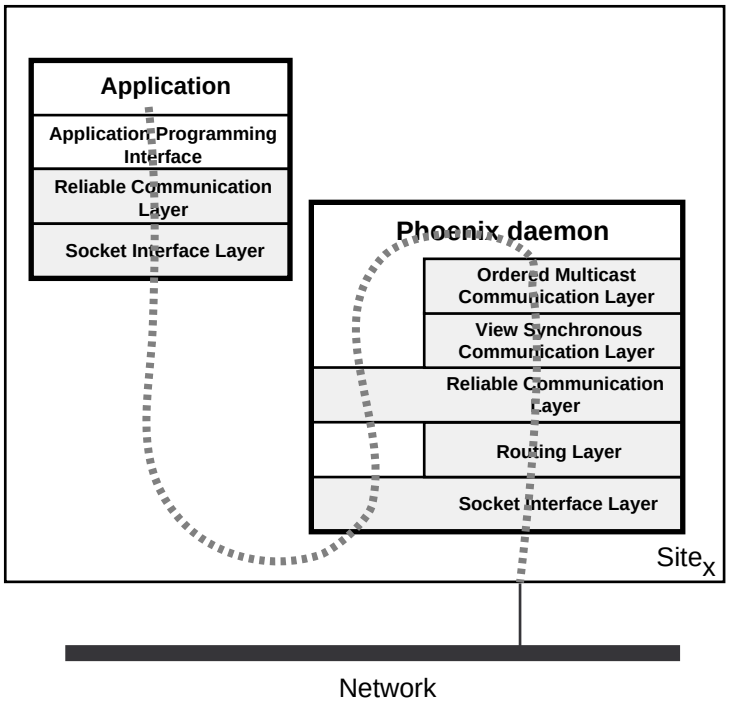

Figure 4.3: Detailed System Configuration

Concerning the description of the architecture, it can be assumed, that the application and the daemon are merged into one sole unit as there is conceptually no difference (see Figure 4.1).

In the following sections each of the layers in Figure 4.1 are described by focusing on how it interacts with the over- and underlying layers, whereas the implementation of the different layers is described separately in the next chapters. The analysis of the architecture is done in a bottom-up fashion in order to be able to give a good view of which layer uses which information from the layer below. However a short introduction describing how the different layers interact will first be given.

4.2 Layered Architecture and the Event-Driven Programming Model

The main idea in a layered architecture is to design each layer to be as independent as possible from other layers. Since a layered architecture is a hierarchy, it is important that a layer is only dependent on the underlying layers, not on those overlying it. This means that a layer can know the layers and their entries which are below itself, but knows nothing about the layers above. This allows the use of a layer in different contexts, as the layer makes no assumptions about the overlying layers. However, this is in contradiction with the flow of information through the

layers, as information can flow from the top to the bottom layer and vice-versa. Typically, in a communication architecture, an outgoing message goes through all the layers in a top-down manner, whereas an incoming message enters through the bottom layer and has to be pushed through the layers in a bottom-up way. This can be implemented using upcalls.

In the presented solution the layers do not need to know directly which procedures should be called in the overlying layers. An overlying layer tells the underlying layer on startup which procedures to call, and the underlying layer calls them when necessary, once they are installed. In order to keep track of those procedures from overlying layers, the current layer keeps a pointer to these procedures in *procedure variables*. The action of calling overlying procedures through procedure variables is known as *calling back scheme*. Thus, the installed procedures are called *callback procedures*, or simply *callbacks*.

Callback procedures are usually coupled with an event-driven programming model, where the main task waits for the first event, acts accordingly and then waits for the next event. So, the basic idea in an event-driven programming model is that the overlying layer does not explicitly wait for something to happen, e.g. receive a message, rather the underlying layer will inform it that something has happened[2]. The occurrence of an event within some layer L, e.g. the reception of a message, triggers the call to a callback procedure which propagates the event to the layer $L+1$. In order that the layer L can call a callback procedure, it has to have control over the flow of execution. This means that, once the callbacks of layer $L+1$ are installed, the layer $L+1$ passes the control to the layer L by calling a special entry in the layer L, called the *loop*. At this point the layer $L+1$ regains control only when the layer L calls a callback of layer $L+1$. After processing the event, the callback of layer $L+1$ usually terminates, i.e. the control is given back to the *loop* of layer L. Figure 4.4 shows a typical architecture with 3 layers and Figure 4.5 a typical calling scheme.

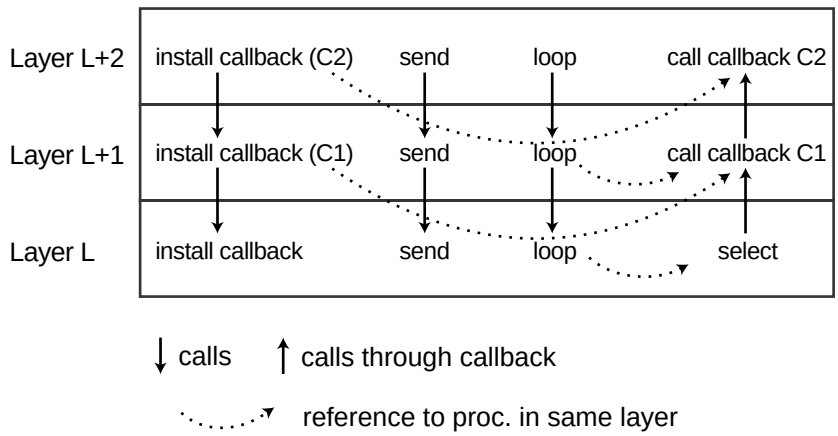

Figure 4.4: Sample Architecture

[2]This is also called the Hollywood principle: *don't call us, leave your number and we'll call you!*

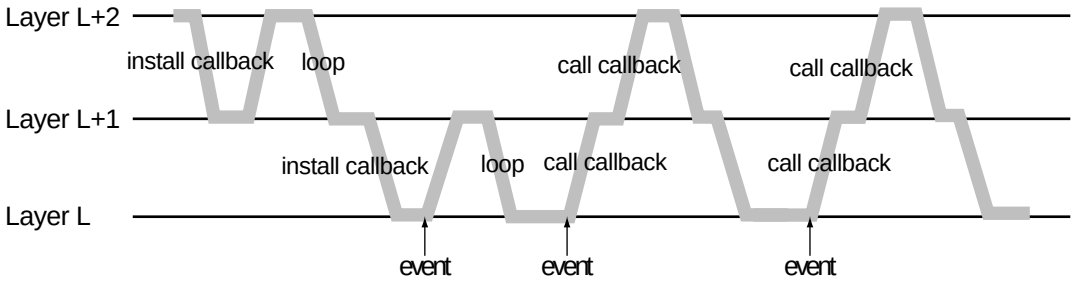

Figure 4.5: Sample Execution

Note that this scheme can be easily generalized to an arbitrary number of layers, and that the *loop* of each layer usually consists only of the call of the *loop* of the underlying layer.

This kind of layered architecture is particularly interesting when implementing protocols based on state machines. Each time an event happens, the *loop* in the lowest layer L calls the callback of the overlying layer $L + 1$ which corresponds to the event. The overlying layer does the transition from one state to another, depending on the context and the contents of the message, where the transition from one state to another can include the call of yet another callback in the next overlying layer $L + 2$ and so on. This also allows the higher layers to be informed about the occurrence of an event, if necessary. Finally, all callbacks terminate and the *loop* of layer L finally regains the control for the next event.

4.3 Phoenix Socket Interface Layer

The lowest Phoenix layer is the socket interface layer (Figures 4.1 and 4.3). Phoenix uses UDP[3] [Comer 88] as the communication infrastructure for all its communication. UDP is a typical low level communication protocol without many guarantees, but it is efficient, and its simple interface is ideal to implement more powerful protocols on top of it. The main characteristics of UDP are the following:

- point-to-point abstraction: a message has a source, and a destination. There is a logical channel between the source and the destination which is responsible for the transport of messages.
- unreliability: UDP only provides unreliable communication, i.e. messages can be lost or duplicated during transmission. Corrupt messages are discarded by UDP using checksums, and transmission delays for correct messages are not bounded. UDP does not implement any flow and congestion control, which is an issue that greatly influences the reliability of communication.

[3]UDP stands for (U)ser (D)atagram (P)rotocol.

- message boundaries: if a message is delivered at a destination, it is delivered with exactly the same boundaries as it was sent by the sender or not at all.
- message size is bounded and system-dependent, typically limited by the length of the message buffer in the system.
- the send primitive is non-blocking and the receive primitive can also be made non-blocking.

The non-blocking feature of the send and receive primitive is important as this enables Phoenix to do other things during the transmission of a message. Typically Phoenix waits either for the arrival of a message, which will be delivered using callbacks to upper layers, or the expiry of a timer, previously set up by an overlying layer, which will also be delivered as a *timeout event* to the upper layers[4].

The task of the Phoenix socket interface layer is to convert the system call for receiving a message or the timeout, into the call to the corresponding callback procedure installed from a higher layer. In case of message reception, the received message is passed as a parameter to the callback. In the case of the expiry of a timer, the corresponding callback is called without any parameter. Chapter 5 gives a complete description of the implementation of the socket interface layer. Its interface is as follows:

- send (dst, msg): this primitive can be called from the overlying routing layer to send message *msg* to the destination *dst*, using point-to-point communication. The *dst* parameter is composed of a site and a port on this site. The message can be lost, duplicated but not corrupted, and there is no upper bound on the transmission delay. The call to this primitive does not block the caller.
- loop (): this procedure allows the dispatching of the two events *message reception* and *timeout expiry* at the socket interface layer. In particular this procedure allows the socket interface layer to receive messages and deliver them through the callbacks installed from the overlying layer, using the procedures described below. The other information delivered by this layer is the expiry of formerly installed timeouts using the *set_timeout_callback* procedure below. This information is delivered to the overlying layer through another callback (see Chapter 5 for its implementation and use).
- set_message_callback (proc): this procedure allows an overlying layer to install a callback procedure *proc* through which received messages are delivered.

[4] A non-blocking receive call is implemented using the system call *select* under the UNIX operating system. The *select* system call returns information either about the availability of data on a descriptor or the expiry of a formerly specified timeout passed as a parameter to the call.

- set_timeout_callback (proc, time): this procedure allows the overlying layer to install a procedure *proc* and a timeout value *time*. When the timeout expires, the procedure *proc* is called.

Note that in the actual architecture the overlying layer is the routing layer (see Figure 4.1). As other layers above the routing layer should also have access to the primitives of the socket interface layer, all the primitives of the socket interface layer are present in the routing layer, as well as in any other overlying layers, i.e. each layer redefines the primitives of all its underlying layers.

4.4 Routing Layer

The aim of the routing layer is to improve the reliability of the socket interface layer, which provides point-to-point communication (using UDP) without ensuring transitivity of the communication.

Figure 4.6 shows a typical example of non-transitivity of UDP in a system with three processes p_1, p_2 and p_3.

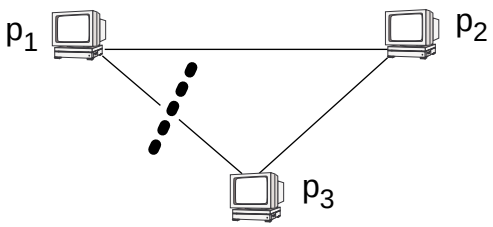

Figure 4.6: Non-Transitivity

A link failure between p_1 and p_3 might lead to the following situation: p_1 and p_3 cannot communicate, but both processes can communicate with p_2. Transitivity of the communication ensures that if p_1 can communicate with p_2, and p_2 with p_3, then p_1 is able to communicate with p_3. UDP does not, and cannot, provide this property for any possible situation (see Chapter 6).

As link failures or overloaded links, leading to the situation illustrated in Figure 4.6, are not uncommon in large scale systems, Phoenix addresses the problem. A distributed application consists of a set of processes, that can implement their own routing strategy[5]. With transitive communications a message sent has a much higher probability of being received by the destination process.

[5]The IP protocol, on which UDP and the socket interface layer are based, is already doing some dynamic routing to mask link failures, but experiences on the Internet have shown that this routing is not sufficient to guarantee transitivity.

Besides rerouting messages, if necessary, the routing layer also delivers information to higher layers about the reachability or the unreachability of processes. The higher layers are interested in this reachability information, especially when it changes. As we will see later, the information about the reachability or the unreachability of a process is important for the reliable communication layer (see Chapter 7) and the view synchronous communication layer (see Chapter 8). Therefore, the dynamic routing layer provides the possibility to install a callback, which is called each time a change in the reachability information occurs. A process p_j unreachable from p_i is suspected by p_i to have failed.

A higher layer using the routing layer, calls its send primitive; messages are transparently rerouted, if necessary. The send primitive of the routing layer reads the routing table, and sends the message either to a routing process or to the destination process using the *send* of the socket interface layer. There is no retransmission in case of loss of the message; retransmission is handled by the overlying reliable communication layer (see Figure 4.1). This offers the following advantage: upon the second and subsequent transmissions of a message, routing information may have changed.

The routing layer also acts as a router for incoming messages from the socket interface layer. To do so, the routing layer applies the same lookup and send technique described above. Messages received from the socket interface layer for the local address are handed through to the upper layer again using the callback mechanism.

The interface, provided by the routing layer, can be summarized as follows:

- routed_send (dst, msg): this is a non-blocking, unreliable send primitive sending a message *msg* to a destination *dst*; there is no retransmission. The destination is searched in the routing table and the message is routed accordingly.

- set_message_callback (proc): allows the installation of a callback procedure *proc* in the routing layer, which is called each time a message has to be delivered to the upper layer.

- set_reachability_callback (proc): allows the installation of a callback procedure *proc* in the routing layer, which is called each time a change occurs in the set of reachable and unreachable processes. The procedure *proc* has two parameters, when called: the *CommSet* of reachable processes and the *SuspSet* of unreachable processes.

Remember that the routing layer also provides access to the primitives to the underlying, socket interface layer (see Section 4.3).

4.5 Reliable Communication Layer

The purpose of this layer is to implement reliable channels[6]. Reliability can be defined by the following property:

"If the sender and the destination do not fail,
then the message will be eventually delivered by the destination process."

Unfortunately, this definition does not consider link failures. In Chapter 7 this definition is adapted to suit the model better where link failures are considered.

The send primitive of the routing layer is not reliable. The task of the reliable communication layer is to ensure, or at least to do its best to ensure reliability on top of the underlying, unreliable primitives[7].

A higher communication layer can install an acknowledgement callback in the reliable communication layer which will be called each time the sender knows that a message has been successfully delivered at a destination. If a message has been acknowledged it is discarded. Similarly, there is a possibility to install another callback, called the undelivered message callback, which informs an overlying layer, after some time, that a particular message still cannot be considered as delivered at the destination[8]. This callback does not stop the reliable communication layer from retransmitting the message. Thus, the callback can be called more than once. It is up to the sender to decide when to cancel the send of a message using the *cancel_send* primitive. If the message is cancelled, the message has been sent with *best-effort* communication semantics. If the message is not cancelled, the send is considered to be *reliable*.

The reliable communication layer provides unordered and *first-in-first-out* communication. The *first-in-first-out* order is implemented on top of the unordered message delivery and an overlying layer can indicate by a flag, whether it wants *first-in-first-out* order on some message or not. *First-in-first-out* order is guaranteed for each destination independently.

On the receiver side, the reliable communication layer ensures *at-most-once* delivery for each message[9].

The reliable communication layer also implements message fragmenting. As the message size in the socket interface layer is limited, it is necessary to cut long messages into smaller fragments

[6] It is probably better to say that this layer implements *eventually* reliable channels [Basu 96].

[7] We see in Chapter 7 that lost messages are retransmitted until they are acknowledged by the destination.

[8] We cannot know, whether the message has been delivered, as all acknowledgements could be lost on their way back.

[9] UDP cannot only lose or corrupt messages, it can also duplicate them e.g. at a router or a gateway. Further retransmissions of messages are another source of duplicates.

in order to send them. At the receiver these fragments are reassembled, and delivered, when all fragments are received. This allows a caller to send a message of any size.

Summarizing this section, the reliable communication layer provides the following interface:

- reliable_send (dst, msg): sends a message *msg* to a process *dst* and retransmits it until an acknowledgement is received. An optional FIFO flag ensures the *first-in-first-out* delivery of messages (see Chapter 7).
- group_reliable_send (dst, msg): this is a slightly modified *reliable_send* using another algorithm for guaranteeing *at-most-once* delivery (see Chapter 7).
- cancel_send (msg): cancels the send of a message *msg*.
- cancel_sends_to_dest (dst): this call cancels the retransmission of all messages to the specified destination *dst*.
- set_message_callback (proc): this procedure allows the installation of a callback procedure *proc* which is called each time the reliable communication layer wants to deliver a message. Each message is delivered *at-most-once*.
- set_acknowledgement_callback (proc): this procedure allows the installation of a callback procedure *proc* which is called each time a message, sent by the *reliable_send* primitive, is acknowledged by the destination.
- set_undelivered_callback (proc): this procedure allows the installation of a callback procedure *proc* which is called when a message is, after several retransmissions, still not acknowledged.
- get_msg_id (msg, *var* id): this procedure retrieves the identifier *id* of a message *msg* which is handled by the reliable communication layer.

4.6 View Synchronous Communication Layer

The view synchronous communication layer implements the view synchronous communication paradigm introduced in Chapter 2. At each process which is member of a group, it maintains the associated structures, i.e. the membership of the current view and the number of the view. Further, it provides a *reliable multicast* primitive to multicast a message inside a group. Multicast messages and views are delivered at group members using callbacks, where the messages are ordered with respect to views, i.e. every process, member of a view $V_i(g)$ and the next view $V_{i+1}(g)$, has delivered the same set of messages in the view $V_i(g)$. Changes in group membership

can be due to processes explicitly joining or leaving the group, but also due to failure or partition of processes. If the membership of a group g changes, the view synchronous communication layer of the members of g executes a protocol that ensures the view synchrony property. This protocol is called the view change protocol. During the view change protocol from $V_i(g)$ to $V_{i+1}(g)$, new messages, coming from an overlying layer, are multicast in the new view $V_{i+1}(g)$. An overlying layer knows that on receiving a new view $V_{i+1}(g)$, all messages of $V_i(g)$ have been delivered.

The view change protocol is launched when communication problems occur with one or more processes of a group. These problems are signalled by the delivery of a new reachability set from the routing layer. Upon launch of the view change protocol concerning group g, the view synchronous communication layer calls a particular callback, the *intermediate view callback* which has been installed from an overlying layer (the ordered multicast communication layer), having as parameters the two sets of reachable and suspected processes of group g[10]. Each time these sets change during the view change protocol, the callback is called, until a new view is installed. This allows the overlying layers to know that some view change is taking place, and to know at any time the set of reachable processes of the group. This might allow upper layers to take adequate actions whenever necessary.

The view synchronous communication layer uses the *reliable_send* of the reliable communication layer to implement the multicast of application messages received from upper layers. The same *reliable_send* is also used to implement the view change protocol. Messages and acknowledgements are received through the corresponding callbacks installed in the reliable communication layer.

A process can join a group by calling the join primitive. The join primitive launches the view change protocol that ends up defining a new view including the joining process. A process leaves a group by either explicitly calling the leave primitive, or when it quits or crashes. In either case this will launch the view change protocol, which installs a new view excluding the leaving or crashed process.

Here is the interface to the view synchronous communication layer:

- join (g): this primitive is called by a process to join a group g.
- leave (g): this primitive is called by a process to leave a group g.
- vsc-mcast (g, msg): this primitive is called to multicast a message to a group. The sender must be a member of the group, and the sender also receives the message.
- set_message_callback (proc): allows the installation of a procedure *proc*, which is called each

[10]This is in contrast to the reachability callback of the routing layer, where not only processes of a group, but reachability information about all processes in the system are returned.

time a message is received.

- set_viewchange_callback (proc): allows the installation of a procedure *proc*, which is called each time a new view is defined.
- set_intermediateview_callback (proc): allows the installation of a procedure *proc*, which is called for the first time, when a view change is launched. As long as there is no new view this procedure is called each time the set of reachable processes of the group changes.

The primitives from the reliable communication layer and from the view synchronous communication layer (besides the intermediate view delivery callback) define the minimal set of communication primitives (see Section 4.8) for building fault-tolerant, group oriented applications. This is discussed in Chapter 10, where this minimal group communication infrastructure is summarized and a more suitable, adequate and complete programming interface is presented.

The details of the view synchronous communication layer are discussed in Chapter 8.

4.7 Ordered Multicast Communication Layer

The view synchronous communication layer orders only application messages with respect to views. The ordered multicast communication layer defines primitives to order application messages. These primitives are:

- *first-in-first-out* multicast
- weak totally ordered multicast
- strong totally ordered multicast
- uniform multicast
- global order multicast

Using a *first-in-first-out* multicast, the sender has the guarantee that two consecutive multicasts from the same sender are delivered by each group member in the order they were sent.

If order is needed among all messages in a group, the *weak totally ordered* and the *strong totally ordered multicast* are provided. The difference between these two multicasts is explained in Chapter 9.

The *uniform* multicast ensures the following property: if a member of view $V_i(g)$ delivers a message m while in view $V_i(g)$, then every process in view $V_i(g)$ which is a member of the next view $V_{i+1}(g)$ also delivers m.

Global order multicast orders a message m with respect to all other messages m', independently of the primitive used to multicast m'. This is similar to the *GBCAST* primitive of the earlier implementation of Isis [Birman 87].

A precise definition of these communication primitives is given in Chapter 9. To summarize, the ordered multicast communication layer provides the following interface:

- FIFO multicast: fifo-mcast (group, msg)
- weak totally ordered multicast: wto-mcast (group, msg)
- uniform multicast: uni-mcast (group, msg)
- strong totally ordered multicast: sto-mcast (group, msg)
- global order multicast: glo-mcast (group, msg)

4.8 Application Programming Interface (API)

The application programming interface layer (API layer in Figures 4.1 and 4.7) provides the applications with the primitives of all underlying layers. These primitives define a minimal set of communication primitives which allows the building of fault-tolerant, distributed applications based on replication. This minimal interface is the basis of the communication between the daemon and the local application (see Figure 4.3). In order to complete this minimal interface, Phoenix includes an object-oriented version of the interface, which hides all the details of dealing with different kinds of messages and control of execution flow. This is discussed in Chapter 10, where the class descriptions for the different Phoenix process types (*members*, *clients* and *sinks*) are given.

4.9 Conclusions

Figure 4.7 summarizes the most important primitives and the flow of information through the different layers of the Phoenix system.

This chapter has given a complete overview of the Phoenix architecture. The aim was to give a first description of the specific tasks of each layer and the information which is passed from one layer to the other in either direction. To each layer in the architecture there is a corresponding chapter in the following pages giving a detailed description of the structures, algorithms and protocols used in the different layers.

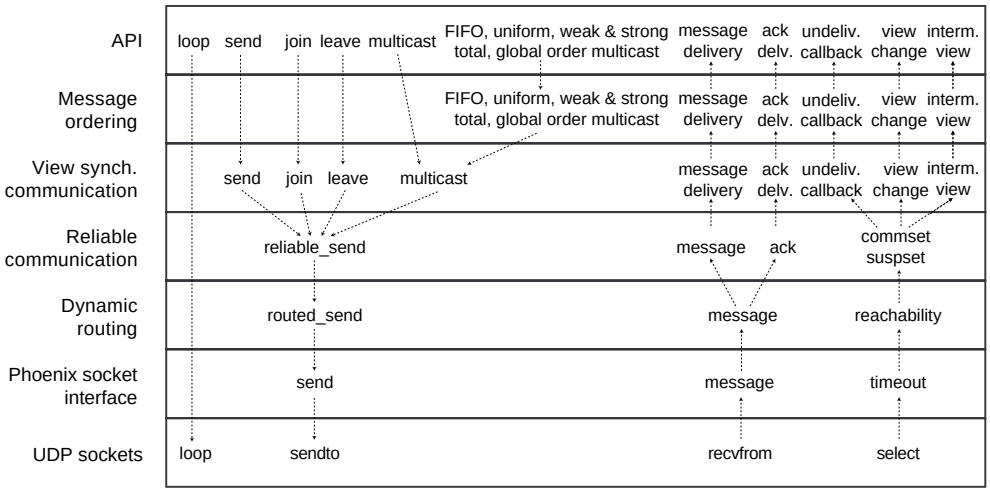

Figure 4.7: Information Flow in the Architecture

A clear modularity and the strict use of callbacks for passing information from lower layers to upper layers, allows one to replace or to pull out layers from the architecture without rewriting code of other layers. One example is the dynamic routing layer, which is useless in a local area network, but can be replaced by a dummy interface handing through the *send* directly to the corresponding send call in the socket interface layer. As another example, consider the port of the Phoenix system to another communication infrastructure, which requires only the replacement and rewriting of the socket interface layer.

Chapter 5

Phoenix Socket Interface Layer

5.1 Introduction

As already described in Chapter 4, the task of the Phoenix socket interface layer, or in short the socket interface layer, is two-fold: it is responsible (1) for incoming and outgoing communication, and (2) for generating upcalls upon message reception and timeouts. This chapter shows the necessary system requirements as well as a description of the implementation of this socket interface layer, based on the programming model described in Section 4.2.

In Phoenix, the basic events that can occur are the reception of a message and the expiry of a timer. For this reason the socket interface layer provides the upper layer with the possibility of installing a message delivery callback and one or more timeout callbacks. The message delivery callback is usually not changed during the whole execution of a program whereas the timeout callback can change. In particular the instant when the callback is called is not depending on the arrival of a message, but is specified upon installation. As there are several layers which will use this timeout callback feature of the socket interface layer, the socket interface layer keeps track of the different callbacks and their associated deadlines, putting them in a list ordered by increasing timeouts. Once all callbacks are installed, it is up to the *loop* to keep track of the different timeouts and adjust them in the following way. Timeouts are specified as a relative value taking the current time as a reference, which means, the *loop* has to subtract the time spent in message delivery callbacks or other timeout callbacks from the timeouts of all remaining timeout callbacks.

Such a scheme naturally provides only the property that a timeout callback is called no earlier than specified, but the actual execution of the timeout callback can be later than specified. But, if the callbacks are programmed in a way that they do not spend much execution time, a timely, but not real-time, execution of the callbacks is guaranteed.

5.2 Implementation

Phoenix is based on the UDP communication infrastructure. Under the UNIX operating system, the BSD socket interface (cf. [Leffler 89]) is used by the socket interface layer. A socket is the structure making the bridge between the user application and the operating system and the network.

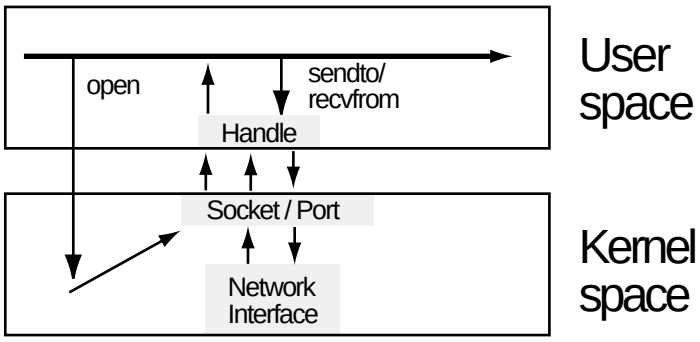

Figure 5.1: BSD Socket Interface

Once a socket has been *opened* it is possible to use it for sending messages. The socket system call *sendto* takes as parameters a set of contiguous bytes, called a message, and a destination to send to. This *sendto* system call does not block[1]. On the other hand, the *recvfrom* system also takes a socket as a parameter. This *recvfrom* call blocks unless there is a message available. As the *loop* also has to handle timeouts, the *select* system call has to be used. The *select* system call terminates if either (1) a message is available on a specified socket or (2) a timeout has expired. The *loop* can exploit the information returned by the *select* call in the following way:

- message available on the socket (sockready, line 11 in Figure 5.2): the *loop* calls the *recvfrom* system call, which does not block as there is a message ready to be received, and delivers the message using the corresponding message delivery callback. Once the callback has terminated, the time spent in the callback is subtracted from the timeouts installed.
- a timeout occurred (timeout, line 15 in Figure 5.2): the *loop* takes the first entry in the list (the list is ordered by increasing timeouts) and calls back the associated procedure. After termination of the callback, as in the message receive case, the remaining timeouts are adjusted.

Here is the implementation of the *loop* and the *send* primitive provided to upper layers. The variable $\mathcal{L}$ is global to this layer and contains the list of events installed from upper layers.

[1]The network interface actually tries to avoid collisions, but if there is some free bandwidth, the message will be sent.

```
1   var ℒ : pointer to event; /* global event list */
2
3   procedure SOCK_loop();
4       var     event : event = record { time, timeout_callback } end;
5                   UPPER_message_callback: pointer to procedure;
6                   buffer : array [0..MAX] of bytes;
7   begin
8       entrytime := gettime();
9       event := getfirstevent(ℒ);
10      case select(socket, timeout) of
11          sockready:
12              recvfrom(socket, buffer);
13              UPPER_message_callback(buffer);
14              /* callback installed by SOCK_set_message_callback */
15          timeout:
16              event.timeout_callback();
17              /* callback installed by SOCK_set_timeout_callback */
18      endcase
19      adjustlist(ℒ, entrytime, gettime());
20  endproc
21
22  procedure SOCK_send(dest, m);
23      sendto(socket, m, dest);
24  endproc
25
26  procedure SOCK_set_timeout_callback(proc, time);
27      insert_in_list(ℒ, compose(proc, time));
28      /* compose combine a procedure and a timeout to a structure
29         to insert in the the list ℒ */
30  endproc
31
32  procedure SOCK_set_message_callback(proc);
33      UPPERmessagecallback := proc;
34  endproc
```

Figure 5.2: Primitives Provided by the Socket Interface Layer

Note that *UPPER_message_callback* and *timeout_callback* are pointers to procedures previously installed by the upper layer in the socket interface layer. Note further that a callback (message or timeout), during its execution, can install new timeout callbacks and send or retransmit messages.

As only the *loop* procedure dispatches events, and there is only one process control flow, mutual exclusion through all layers of the daemon and their protocols is guaranteed. Typically, an installed timeout callback, whose timeout has expired during the processing of another callback, is only dispatched, once the *loop* procedure gets the control back.

5.3 Conclusions

The socket interface layer is the lowest layer in the Phoenix architecture. The *loop* procedure of this layer builds the heart of the Phoenix system and from there all upcoming events are dispatched to higher layers through callbacks. This could seem too restrictive, but for the programming of the Phoenix daemon and Phoenix applications, this is sufficient. The daemon can be seen as a server: upon reception of a message, the associated callbacks execute the corresponding steps and terminate. Explicit control can be obtained by specifying timeout callbacks, which can be used to do periodical tasks (e.g. retransmissions in the reliable communication layer). This basic layer combined with the callback concept, allows all overlying layers to be programmed in an event-driven way, i.e. they (1) start with some initialization task, (2) install their callbacks in the next lower layer, and finally (3) call the *loop* in the next lower layer. At this point, every action is managed by callbacks. This concept is pulled throughout all layers of the daemon, which means there is only one execution flow[2].

[2]We will see in Chapter 10 that this can be to constraining for programming applications, based on additional events such as keyboard inputs, and also for more complex synchronization needs.

Chapter 6

Dynamic Routing Layer

6.1 Introduction

As described in Chapter 4, the dynamic routing layer increases the reliability of interprocess communication in a system which is subject to link failures or link congestions. Link failures might separate two processes one from the other, until the link gets repaired. In the context of a distributed system, where there are usually more than two processes, it is possible for a link failure to split the system into completely distinct subsystems, but it is also possible that these subsystems are not disconnected. Figure 6.1 illustrates the cases where the system is completely cut into two distinct parts, and Figure 6.2 the case where the link failure does not really cut the system into two distinct parts.

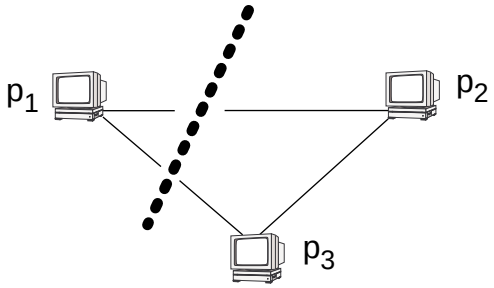

Figure 6.1: Distinct Subsystems

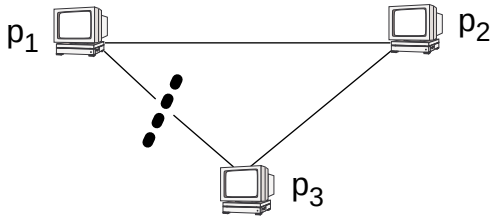

Figure 6.2: Not Distinct Subsystems

In Figure 6.2, process p_1 is still able to communicate with process p_2, but not with process p_3, whereas in Figure 6.1 process p_1 cannot communicate with either of them. Section 6.2 explains why this situation can occur with IP routing. Further, process p_2 can, in both figures, communicate with p_3. If process p_2 were able to forward messages from p_1 to p_3 and vice versa, the link failure between p_1 and p_3 in Figure 6.2 would be masked. To accomplish this rerouting

it is necessary to have a routing table at each process of the distributed application and update it accordingly to link failures, process failures and their repair. How these routing tables are calculated and exploited is the main theme of this chapter, but first, a brief overview of the communication subsystem UDP/IP [Comer 88] and its relevance to the dynamic routing layer is given.

6.2 Communication Subsystem

The Phoenix communication subsystem uses the socket interface layer (Chapter 5) which is itself based on UDP [Comer 88]. The best-known large scale network using UDP as one of its basic communication protocols is the Internet [Leiner 93, Krol 93], a network composed of millions of sites, routers, and gateways. The interesting thing in Internet is, that it is built up of a large set of local area networks, but the interconnection of the networks is not hierarchical, although the addressing scheme of networks, routers and machines (the IP addresses) gives the illusion of a hierarchical organization. In other words, there is no central administration node in the Internet which all communication goes through: the Internet can be seen as a connected network where nodes represent local area networks. Due to the large number of parts involved in such a network, the failure of one component can occur with high probability and the failure of one or more components can cut a network in one or several partitions. First, let us consider Figure 6.3 presenting a typical distributed application with three processes p_1, p_2, p_3 mapped on the Internet.

Such a network can be seen as a graph, where the sites, routers and gateways are vertices, and the communication links are edges. In Figure 6.3 processes represent the sites they are running on[1] and are denoted p_1, p_2, p_3. Routers and gateways are denoted r and communication links are denoted l. The set of links which connects two vertices (i.e. processes, routers and gateways) is called a path. For example, the path between p_1 and p_3 is composed of the links $l_0, l_1, l_3, l_4, l_5, l_{14}$. Figure 6.3 also shows that there are different paths from one process to another, and even between the same two local area networks there is often more than one path. For example, p_2 can be reached from p_1 by the following sets of links: $l_0, l_1, l_3, l_4, l_6, l_7, l_8, l_{15}$ or $l_0, l_1, l_2, l_{12}, l_{14}, l_{15}$ or $l_0, l_1, l_2, l_{11}, l_{10}, l_9, l_8, l_{15}$, etc.. A local area network can be seen as the communication link from a site to a gateway. Therefore, it can be supposed that the process represents the site it is running on, and this site is connected to the gateway through a link.

As routers and gateways can be attached to more than two links, they have routing tables, telling them, for every message received, on which link it should be forwarded. A message for

[1] We shall see later that our routing is only possible if the Phoenix daemon process is present on some site as only the daemon part of Phoenix contains the routing layer (see Figure 4.3 of Chapter 4 page 35) and all Phoenix applications use this daemon for communication with other sites.

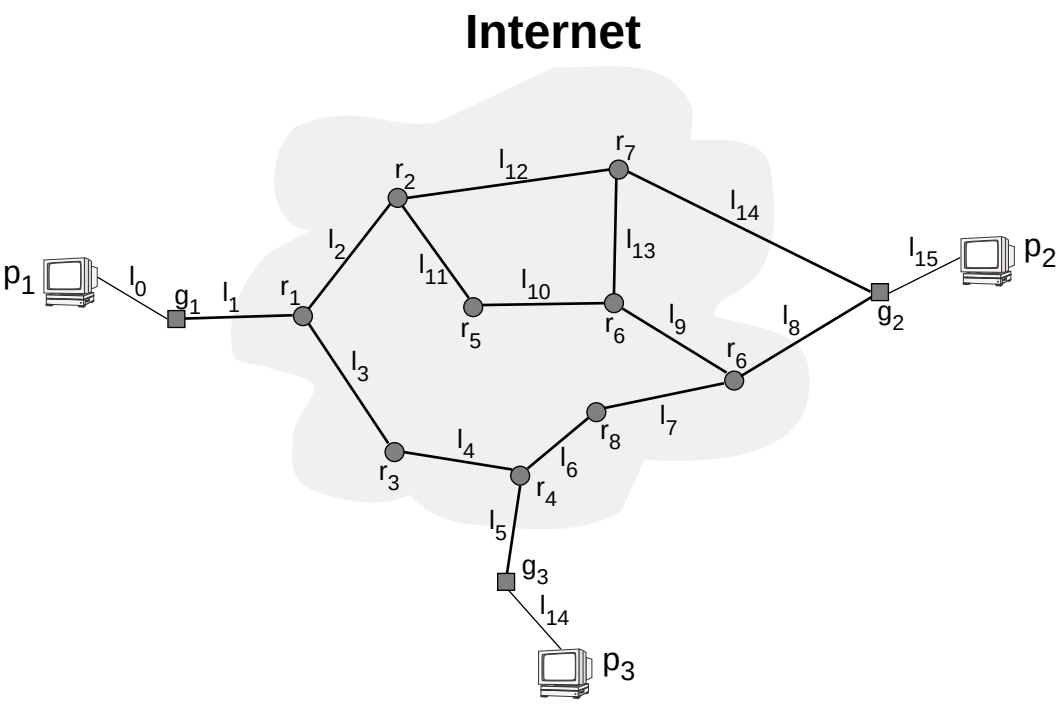

Figure 6.3: Wide Area Network

process p_3, i.e. the site on which p_3 is running, received by the gateway r_1 has to be forwarded on link l_3, whereas a message for p_2 has to be forwarded on link l_2. The task of the IP routing protocol is to keep the corresponding routing tables at all routers and gateways. These tables are more or less static, i.e. they are programmed once and not modified thereafter anymore. However, they could contain redundant routes, e.g. there are two or more links on paths with a lot of traffic[2]. Routing on the IP layer is exploiting this possible redundancy in case of link failure or link congestion, but it is limited to a small set of possible routes. In case of a serious link failure, as Figure 6.4 shows, IP is no longer able to route a message to some destinations.

The link failures illustrated in Figure 6.4 (links l_4 and l_9 have failed), have the consequence that p_1 can no longer communicate with p_3 and vice-versa. However, it is possible to pass through process p_2 in order to reach the process p_3 from p_1. To provide such a functionality, each router and gateway in the Internet would have to keep track of every other gateway and router, and store routing information about them. In a large scale network, such as the Internet, with millions of sites interconnected by tens of thousands of routers and gateways, it would be impossible to store routing information for all routers and gateways at each destination, as this would use huge amounts of memory at each router, gateway and site. This means, for example, that the router r_1 is actually unable to forward a message on another link than l_3, as this is the only route to process p_3.

[2] A typical example is the links across the Atlantic where several links in parallel connect Europe and the USA.

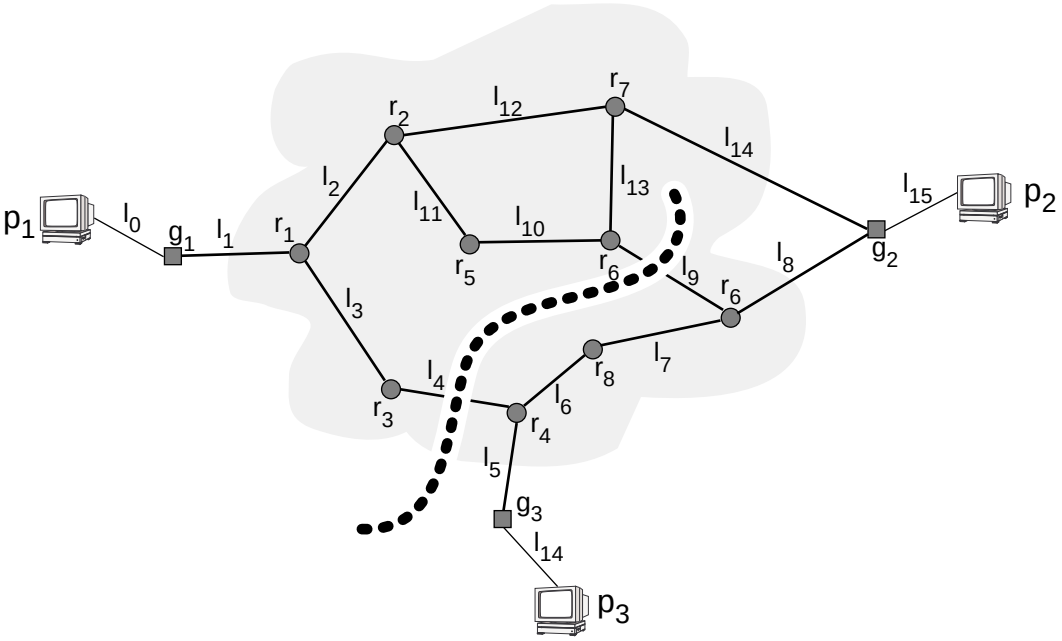

Figure 6.4: Wide Area Network with some Link Failures

This is the point where the dynamic routing of Phoenix plays an important role. One distributed Phoenix application runs on only a small subsystem of the whole Internet, typically 5 to 100 processes in distinct local area networks. In such a subsystem, considering the involved processes as *high-level routers*, it becomes possible to store all available routing information. Considering Figure 6.4 again, it is possible that the process p_1 stores a piece of information that p_3 is not directly accessible, but is accessible through process p_2. So for sending messages from p_1 to p_3, p_1 sends them to p_2, which will forward them to p_3. In order that the processes are able to do this forwarding, they periodically have to exchange reachability information with all other processes. On the basis of the local and the received reachability information, a routing table can be generated which masks cases of failure like the one described above.

As already introduced in Chapter 2, the only way to know whether a process p_j is directly reachable from some process p_i, is to send a message to p_j and to wait for an acknowledgement. In this way each process builds up its local reachability information. This is the information which is exchanged with other processes and allows one to make "unreachable" processes reachable thanks to routing.

The rest of this chapter is dedicated to the protocol used to exchange and calculate the routing information, and a discussion about advantages and disadvantages of the routing layer.

6.3 Protocol Overview

6.3.1 Introduction

The routing protocol is inspired by [Dolev 93], which gives a self-stabilizing algorithm in a dynamic, distributed system, where dynamic means that communication links and processors may fail and recover. For the Phoenix routing protocol, the self-stabilization structures are the routing tables, telling whether there is a route from one process to another, and through which process. The algorithm given by Dolev is based on a shared memory architecture with read/write registers. The Phoenix implementation has to suit an asynchronous, distributed message-passing system where messages can be lost. The differences will be discussed in detail in Section 6.6, but first, the modified protocol is described.

The key idea of the routing protocol is to obtain a copy of each other process' routing table in the system. For this purpose each process sends its routing table to all other processes periodically. In this way each process should periodically receive a routing table from each other process. If a process p_i receives no table from a process p_j for some time, it will suspect either that the process or the link has failed. Tables are sent using the send primitive of the socket interface layer, and received through a callback procedure, installed in the socket interface layer. Each time a process receives a routing table it compares it with its own routing table and changes its own routing table, if there is a *better* route to the process from which the table was received. What *better* means is described in Section 6.5.

The protocol proceeds in asynchronous rounds. In each round, every process p_i sends its routing table to all other processes which are concerned by the routing. Then process p_i waits for a specific timeout τ. During the period waiting for the timeout, process p_i receives routing tables from other processes and inspects their contents. After the timeout expires, new routing information is calculated, higher layers are eventually informed about changes, and a new round is launched. A process p_j from which no routing information is received for more than λ rounds is suspected[3]. The rounds are asynchronous and local to each process, i.e. there is no synchronization between processes. The only assumption is that the period between successive rounds should allow the reception of routing tables from the other processes (i.e. greater than the average *round-trip-time* to the slowest destination). This is not a necessary condition, but significantly increases the accuracy of the information contained in the routing table. The impact of the asynchrony of the rounds is further discussed in Section 6.6.

[3] As messages can get lost, it is a good idea to choose $\lambda \neq 1$; the bigger λ the greater the probability is that the destination has failed or is unreachable, but the longer the time that will go by until an overlying layer will be informed. We shall see in the next chapter about reliable communication, that λ can be chosen equal to half the number of retransmissions before the reliable communication layer informs its own overlying layer; this gives time to the routing layer to recompute its routing information.

6.3.2 Data Structures

Each process manages a routing table. Figure 6.5 shows the routing table in a system with three processes:

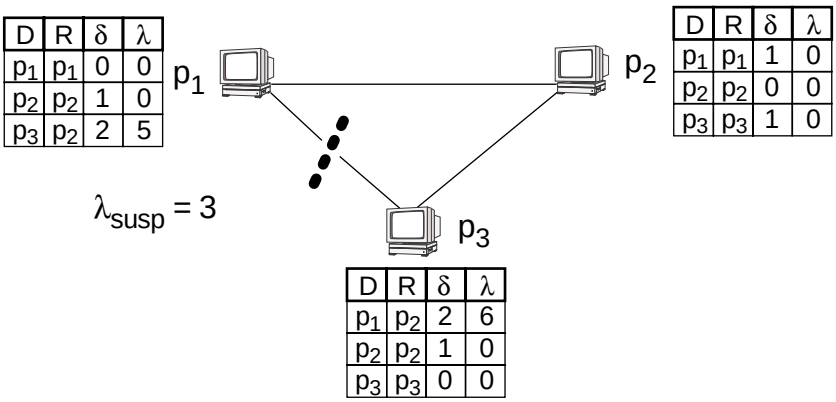

Figure 6.5: Routing Tables

The entries in the tables can be summarized in the following way. For each destination process p_j the routing table at process p_i contains an entry with the following fields:

- **D: destination process** p_j: key of the entry
- **R: routing process** p_r: this is the process to which the message for p_j has to be sent; in the normal situation p_j and p_r are equal.
- δ**: distance** d: this field contains the value ∞ if p_j is unreachable. Otherwise it contains a value expressing the minimum distance between p_i and p_j over all possible routes (see Section 6.5).
- λ**: timeout counter:** this value is increased each time a round expires without having received a table from p_j.

6.4 Implementation of the Routing Protocol

A summarized description of the protocol is given here; the complete protocol can be found in Appendix A.1. The routing protocol is composed of the three blocks:

- *initialization part*: initializes the data structures;
- *table reception part*: consults received messages and changes the local routing table accordingly;

- *timeout part*: responsible for generating suspicions and for sending the routing tables of a new round.

6.4.1 Initialization

The initialization part is executed when a process starts up. Initially no process is suspected and accessible directly, i.e. the *routing process* is set equal to the *destination process*. The distance is set to an initial distance, which could vary from destination to destination (see Section 6.5). Then, the first round is launched; the process sends its routing table to all other processes.

6.4.2 Table Reception

The table reception part is executed during the delivery of a routing table received from another process. First, the *timeout counter* field is reset to zero. Secondly, each entry for a process p_x in the routing table received from p_j is compared to the same entry p_x of the local routing table. There are two cases to consider:

1.) If process p_x is inaccessible from p_j (distance ∞), and p_i uses p_j to route messages to p_x, p_x becomes inaccessible from p_i too.

2.) If process p_x is accessible through p_j with a shorter distance than the one in the current table, p_j becomes the new router for a message sent to p_x (the additional distance between p_i and p_j has to be added before comparing the current distance with the one obtained by using p_j as a router to p_x).

6.4.3 Timeout

The timeout part is executed periodically by a timeout callback installed in the socket interface layer. Each time the callback is called at p_i, it executes the following steps for each entry p_x of the routing table:

1.) Increments the *timeout counter*.

2.) If the *timeout counter* exceeds an initially specified value λ' (e.g. $\lambda' = \lambda/2$ and messages are not already routed to another process, i.e. *dest process* equals *routing process* for this destination, the distance is set to a high value $\delta_{p_i}[p_x] = n$ where n is the number of participating processes; in the following rounds this will allow the choice of another routing through another process (one with distance $< n$ for the considered destination, if it exists) before generating a suspicion of the destination after λ rounds.

3.) If the *timeout counter* exceeds an initially specified value λ and messages are not routed to another process, i.e. *dest process* equals *routing process* for this destination, the destination is suspected and its distance set to ∞. With the suspicion of the destination p_x, all destinations using p_x as a router also become suspect.

4.) If there is a change in the reachability of processes since the last round, the new reachability set *CommSet* and the new unreachability set *SuspSet* are delivered to the upper layer.

5.) The next round is launched by sending to all processes (including those suspected) the routing table.

Note that in steps 2 and 3, a modified distance is not a reason to generate new reachability information. The distance has to change to or from ∞, which indicates that a process has become (un)reachable.

6.5 Shortest Route

In order to be able to calculate the shortest route from a process p_i to p_j, a *weight* has to be attached to every link. The shortest path problem is well studied in the literature. One of the simple methods [Ford 62] is considered here. Consider for each edge (actually a link) a weight. In Figure 6.6 the trivial case is illustrated, where every edge has the same weight.

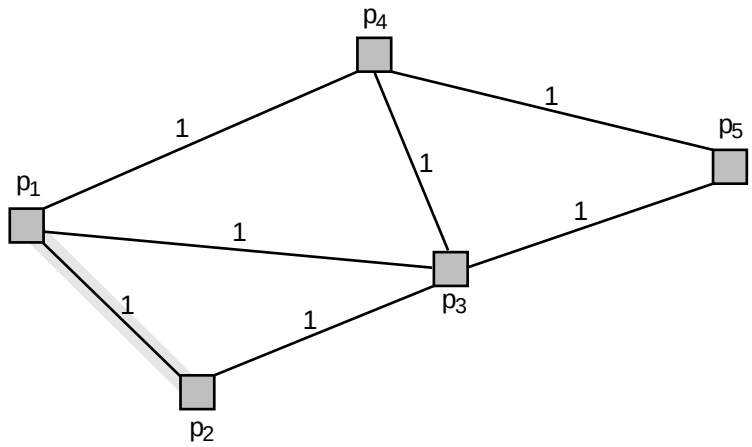

Figure 6.6: Shortest Path with Uniform Weights

The case of equal weights is the most common case, and is used for example in the Internet routing protocol. There are many reasons not to use equal weights on the edges. One reason is that the *link* is actually composed of several links in the underlying communication infrastructure. The elements composing the underlying communication infrastructure can vary in reliability, throughput and performance.

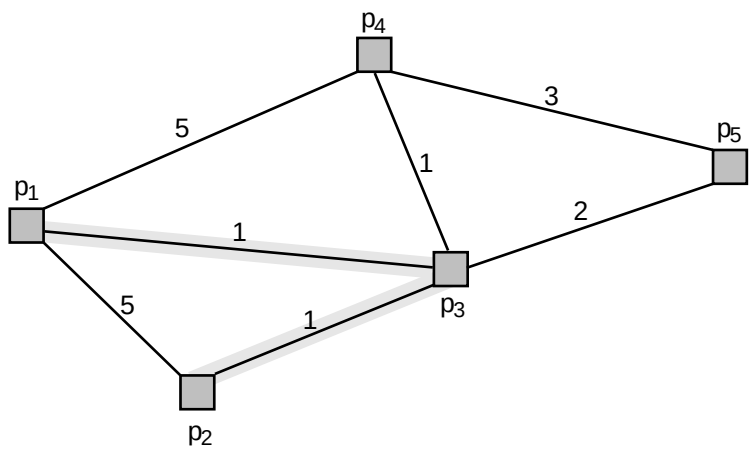

Figure 6.7: Shortest Path with Non-Uniform Weights

Figure 6.7 shows for example a case where the link between p_1 and p_2 is very slow, whereas p_1 and p_2 have a high speed link to p_3. Such cases are common on the Internet. Using link speeds as the weights on the links can improve performance, still having the advantage that, in case of the failure of the high speed link, the routing protocol will switch automatically to the slower link until the faster link is available again. Another interesting way to model link weights is to define them depending on some variable parameter depending on the actual state of a link, for example the actual link load, the actual throughput, the message loss rate, etc.

The modification to the protocol is only minor and concerns only the part where the protocol has to decide to change the router or not. The protocol in Appendix A.1 includes this feature.

6.6 Self-Stabilization

One interesting thing about the routing protocol is its self-stabilizing property. Self-stabilization can be defined as follows:

"Self-stabilization is the fact that a distributed system can tolerate any number and any type of faults in its history, but after the last fault occurs, the system starts to converge to a legitimate behavior [Dolev 93]".

Self-stabilizing, respectively legitimate behavior, means in the Phoenix context that routing tables become stable after some time, if there are no further changes in the state of processes and links. The self-stabilizing character of this protocol is similar to the one described in [Dolev 93]. One of the differences between the algorithm described in [Dolev 93] and the one here is that the communication model in [Dolev 93] is based on synchronized processes, clocks

and rounds, or in other words transmission delays are bounded and fixed. This fact allows Dolev to give an upper boundary for the time after which the system becomes stable when no further failure occurs. In the Phoenix system, based on asynchronous communication, transmission delays are not bounded, but a timeout mechanism is used to suspect processes. A process which has not replied after the timeout, is suspected. This could be a false suspicion, but even if a false suspicion holds for a long enough time the routing tables become stable.

Due to asynchrony of communication, a process p_i may not receive any routing table from p_j in round r, and two tables in round $r + 1$. In this case process p_i considers only the table from p_j with the highest round number. This corresponds to the most recent table, and former tables of the same process are discarded.

6.7 Discussion

6.7.1 Increased Reliability of Delivered Information

The benefits of using the routing layer are two-fold:

- it increases the probability that a message is delivered at its destination, and
- the reachability information delivered to upper layers is more accurate than the one delivered without routing.

The reliable communication layer (see Chapter 7) benefits from the increased reliability in message transmission, whereas the view synchronous communication layer (see Chapter 8) benefits from the more accurate reachability information. The protocol used to implement the heart of the view synchronous communication layer requires a *failure detector*, and the routing layer is actually implementing a *failure detector*, thanks to the two sets *CommSet* and *SuspSet*. Another important point is, that the routing layer leads to a reduced number of configuration changes since link failures do not necessarily lead to partitions. This makes the system more stable.

6.7.2 Performance Aspect

Pipelining Effect

One of the features of the routing layer is its complete transparency to the overlying layers in the absence of failures. The cost of rerouting a message only has to be paid if there is a link failure which can be masked by rerouting the messages. Rerouting is costly, as each message has to be handled by additional processes, but if reliability is an important issue rerouting is an

effective mechanism to attain reliability. Rerouting also implies increased latency of messages, and increases round-trip-times for a message-acknowledgement pair. However, the pipelining effect created by sending several consecutive messages considerably attenuates these increased values.

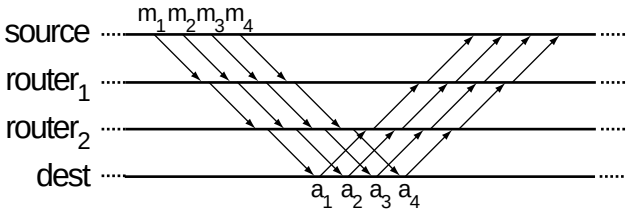

Figure 6.8: Pipelining Effect

Consider Figure 6.8, where a process *source* sends four messages to process *dest*. Actually the process *source* does not have to wait for the acknowledgement a_1 from process *dest* and can send the other three messages as soon as the call of the first send has terminated. Thus, the transfer of the four messages does not take four times longer. The time for the reception of the first acknowledgement a_1 is increased due to routing, but the delays of the other acknowledgements $a_{2,3,4}$ are as small as if routing were not present.

Common Path Feature

In a distributed system it is often the case that a process wants to broadcast a message to every process in the system. Consider Figure 6.9 where some link failures have occurred.

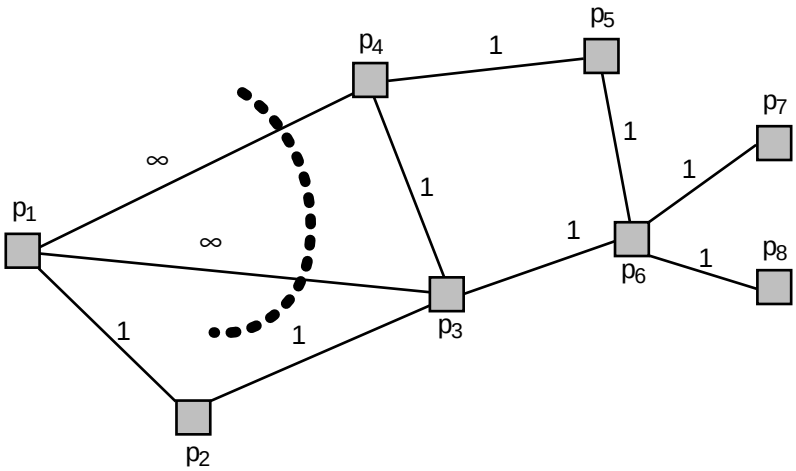

Figure 6.9: Common Paths

Suppose that process p_1 wants to broadcast a message. Suppose that the routing layer offers a primitive which allows a higher layer to specify a set of destinations for a message. When this primitive is called, one way to execute the send, is to send a copy of the message to each

destination. In the case of Figure 6.9 all messages from p_1 to any other process are routed through p_2 and p_3. The other way to do the send is to analyze the set of destinations and identify destinations which use a common initial link: there will be only one send of the message for each common link. Additional information in the header indicates the set of destinations for some particular message and link. At the next *router* this information is extracted and interpreted. Consider the two messages sent to the processes p_7 and p_8. They take the same path up to process p_6. So, only one message is sent to p_6, and will be split up at p_6, where one copy is for the process itself, and there are two further copies, one for p_7 and the other for p_8.

This common path algorithm avoids sending several copies of the same message over the same link, and thus decreases the communication traffic on a particular link.

Another feature of the common path routing shows up, when a slow communication end-point wants to broadcast a message. Consider Figure 6.10.

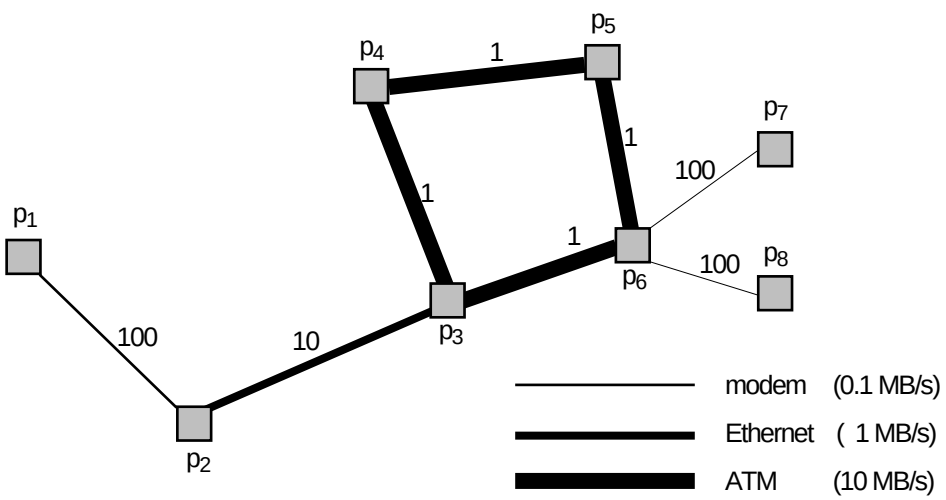

Figure 6.10: Common Path Feature

A process p_1 is attached to a high speed network through a slow link, e.g. a modem to an Ethernet or an Ethernet to an ATM network. The distributed system is composed of the processes $p_1, ..., p_8$. If p_1 sends a message to all processes, it has to send seven messages. If the common path feature is active, there is only one message sent on the slow link between p_1 and p_2, and from p_2 to p_3. At p_3 the message is copied, and sent accordingly to the destinations in the header and the information in the local routing table. Thus, the common path routing allows process p_1 to send seven times more messages than without using this scheme, which is considerable, especially when an initial link is slow.

Assuming that from each slow network there is only one link to a fast network, e.g. the link p_1 to p_2 in Figure 6.10 is the only link from p_1 which has a weight other than ∞ (compare with Figure 6.9), and the weights of the edges are assigned dependent on their throughput, then the

presented algorithm automatically discovers the feature described above.

Note that the reliability of communication is increased using common path routing. As there are fewer messages to send, there are smaller risks to be confronted concerning congestion, message corruption or loss. This is especially important for slow links.

6.7.3 IP Multicast

The weakness of the routing in the IP layer has already been mentioned in Section 6.2. More interesting is the comparison with the IP multicast [Deering 88, Deering 89, Deering 90] which provides similar facilities to those of the routing layer described, but on a more static and non-fault-tolerant base.

The IP multicast [Deering 89, Deering 90] was designed to transmit data from one process to a set of other processes, not necessarily in the same local area network[4]. The particularity of the IP multicast is that the set of destination processes is not initially known, but they share a particular IP address, called the multicast address. A destination can receive messages multicast to a particular IP multicast address, by subscribing to this *multicast address*. This subscription is distributed to all local area networks, containing processes which have subscribed to this address. This subscription itself can be seen as a multicast to a special multicast address, to which all processes intending to multicast messages have subscribed. The IP multicast is implemented on top of IP and uses a daemon in each participating local area network. The failure of this daemon implies, that multicasts are no longer forwarded to other destinations nor are multicasts coming from another network distributed in the local area network. In the Phoenix system, the routing is part of the Phoenix daemon and is up as long as the daemon is running.

Another weakness of the IP multicast is that the actual set of communicating processes is not known, but there is only some approximation as subscriptions to a multicast address are multicast themselves to all considered processes. Again, the presented solution goes further by having a complete, but still dynamic view of the system. Further, the Phoenix routing does not uselessly increase traffic by multicasting messages to branches where no processes have subscribed to the considered multicast address. The routing layer, completed with the common path feature, can be seen as a specialized implementation of IP multicast on top of IP. The difference is that the routing layer knows a priori the set of processes involved in the system, and uses only these processes to route messages. IP multicast benefits from upcoming support in operating system kernels, routers and gateways, but only a few systems and networks, e.g. the MBone [Eriksson 94], provide the IP multicast outside of local area networks.

One advantage of the IP multicast, not considered here, is the ability to send one single IP

[4]Only the routing aspect of the IP multicast is considered here.

multicast received by several processes in the same local area network. The routing layer does not consider this situation, as it is aimed at large scale applications, where the number of processes per local area network is small (often 1), but there are a lot of local area networks.

6.8 Conclusions

This chapter has shown how to improve the reliability of communication thanks to routing. An algorithm has been proposed which is not only able to mask link failures, but also increases the throughput of communication. The criterion for choosing the best communication path has been identified. The interesting thing about the criterion is its genericity, which makes it simple to replace a *shortest* path method by another suiting the needs of other applications. Using the common path algorithm when sending messages, traffic can be significantly reduced on the network in many situations when rerouting is necessary due to a link failure. This algorithm also has been shown to be useful for increasing the throughput of multicasts when slow links are involved. However, these arguments only apply if large scale use is considered.

Chapter 7

Reliable Communication Layer

7.1 Introduction

Reliable communication between processes is important for building distributed, fault-tolerant applications. Usual definitions and implementations of reliable communication layers consider only point-to-point communication, whereas the following definitions consider in addition reliability in a group communication context. In this context usually more than two processes communicate with each other. Chapter 6 has showed that the Phoenix routing layer increases the reliability of communication in such a group communication context. However, the Phoenix routing layer does not give guarantees on the delivery of the messages. The reliable communication layer, presented here, provides *reliable communication* on top of this routing layer.

The use of *reliable* semantics requires a precise definition. The aim of this chapter is to define different types of reliability (see Section 7.2) for different contexts, in order to give specific implementations afterwards (see Section 7.3). Each implementation provides different reliability semantics needed by the different overlying communication layers. For instance the view synchronous communication inside a group described in Chapter 8 needs another form of reliability than communication between a client and a group.

7.2 Definitions

7.2.1 Reliability in the Absence of Link Failures

A reliable communication can be defined as follows:

"If the sender and the destination do not fail,
then the message will eventually be delivered at the destination."

Consider for the moment that the sender and the destination do not fail. It is possible that a message does not arrive the first time it is sent. Message retransmission is necessary to ensure that the message eventually arrives at the destination.

7.2.2 Reliability with Link Failures

A message can also be lost because of link failures. By considering the failure of the transport media, the definition of reliable communication above is not correct anymore. An extended version could be:

"If the sender and the destination do not fail,
and the link between the two is eventually operational,
the message will eventually be delivered at the destination."

Chapter 6 has introduced the reachability sets *CommSet* and *SuspSet* (see Section 6.4.3), which are delivered by the dynamic routing layer through the reachability callback. These sets contain information about reachable and unreachable processes, and based on this information, the above definition can be even refined:

"If the sender and the destination do not fail,
and the destination is eventually in the CommSet of the sender,
then a message will eventually be delivered at the destination."

This definition gives a much more precise definition of reliability considering more than two communicating processes and link failures.

7.2.3 Message Cancelling

Consider a process p sending messages to q, and q having crashed. The problem with the reliability definition is that it assumes infinite memory. As it is not possible to distinguish a failed process from one that is only slow [Fischer 85], p has to buffer all unacknowledged messages for q. As q has crashed, sooner or later the memory of p becomes full, if the sender p continues generating new messages for process q. To avoid this problem, the reliable communication provides a primitive to cancel a message:

"If p sends a message m to q, and later cancels m,
then q might not receive message m."

This leads to violation of the reliability property in case process q has not failed, but was only partitioned or slow. However, we will see in Chapter 8 that in group communication systems message cancelling makes sense and reliability can be reestablished by defining a new context[1]. The idea is to resynchronize the sender and destination, as soon as they can communicate again. The resynchronization consists of informing the destination about the cancelled messages without transmitting them (see Chapter 8 for details).

To summarize, the reliable communication layer provides *at-most-once* delivery semantics, rather than reliable communication semantics. The two different implementations are discussed in the following section.

7.3 Implementation of At-Most-Once Delivery

Reliable communication over unreliable channels is achieved by sending a message and waiting for the acknowledgement from the destination. If this acknowledgement is not received, the message is retransmitted. When and how messages are sent and retransmitted is discussed in Section 7.4 together with flow and congestion control. This section presents the main aspects on how to deal with multiple receptions of the same message due to retransmission.

The reliability definition states that a message sent *once*, must be delivered eventually *once*. We have seen in Section 7.2.3, that this assumes infinite memory. In order to limit this memory consumption the *once* guarantee has to be loosen to an *at-most-once* guarantee.

The reliable communication layer proposes two different implementations of the *at-most-once* guarantee. The first implementation is used to send messages to processes inside a group and is called *group-at-most-once* implementation. The second implementation is used for the communication outside a group, e.g. a process and a group member, and is called simply *at-most-once* implementation.

The difference between the *at-most-once* and the *group-at-most-once* consists in the fact, that in the *at-most-once* case, time stamps will be used in order to avoid infinite memory consumption, whereas in the *group-at-most-once* case, explicit resynchronization will allow limiting the memory consumption.

Common to the two semantics is the need to detect multiple received messages using unique message identifiers. Before giving the algorithms for the two delivery implementations, a simple method for generating these unique message identifiers is presented.

[1] This context is actually a new view, which is discussed in Chapter 8.

7.3.1 Message Identifiers

In Phoenix, each message to be sent is tagged with a unique identifier. This identifier is composed of the following fields:

- IP number of the sending host
- process identifier
- sequence number

Process Identifier

To each process on a site is associated a process identifier local to the site[2]. This number is unique for each process and a process that recovers after a crash gets a new process identifier[3].

Sequence Numbers

The message identifier is completed with a sequence number local to each process, beginning at zero and incremented each time a new message is sent by the reliable communication layer. These numbers are bounded in an interval $[0..\mathcal{I}-1]$, but under reasonable assumptions corresponding techniques allow them to seem *unbounded* (for an implementation see [Tomlinson 75]).

7.3.2 Implementation of Group-At-Most-Once Delivery

Inside a group every *correct*[4] process should have delivered all messages multicast to the group in some view[5]. To each view will be associated a unique view number (see Section 8.3) which is incremented each time the view changes. Messages from old views will be detected using this view number and discarded. As a process has to deliver all messages for some view, the set of sequence numbers for one view has to be contiguous and may not contain any missing sequence number in contrast to the *at-most-once* implementation in Section 7.3.3, where the set of sequence numbers must not be contiguous and there is no notion of view numbers.

[2]In particular the daemon and the different Phoenix applications using the daemon (see Figure 7.3) have different process identifiers as each one includes a reliable communication layer.

[3]Process identifiers are actually bounded to some interval, but a stably stored counter, which is incremented each time the identifier resets to zero (e.g. after the crash of the site), guarantees the uniqueness and monotone incrementation of this counter.

[4]A definition of *correct* process will be given Chapter 8.

[5]We will see in Chapter 8, that the multicast to a group will be implemented by sending a copy of the message to each member of the group.

In the *group-at-most-once* implementation, a process keeps the last received message sequence number lc_{src} from every process in the group[6]. However, there is also a need for a list of sequence numbers, whose messages are delivered out of sequence. This is necessary as messages are not necessarily received in the order in which they are sent. For each source src of messages, there is such a list L_{src}, containing the set of messages delivered out of sequence. Further the function $seq(m)$ allows one to get the sequence number of a message m. The protocol which is executed when a message m from source src is received, can be summarized by the following algorithm:

```
1   var L : array of sequence_nbr_sets
2
3   procedure message_delivery_callback (m)
4   begin
5       if seq(m) > lc_src + 1 ∧ seq(m) ∉ L_src then
6           deliver(m);
7           L_src := L_src ∪ {seq(m)};
8       elsif seq(m) = lc_src + 1 then
9           deliver(m);
10          lc_src := lc_src + 1;
11          while lc_src ∈ L_src do
12              L_src := L_src − {lc_src};
13              lc_src := lc_src + 1;
14          end while
15      else
16          discard(m);
17      end if
18      send acknowledgement
19  end
```

Figure 7.1: Group-At-Most-Once Delivery Algorithm

Line 5 tests whether the received message is out of sequence. Before delivering m (line 6) and adding the sequencer number of m to L_{src} (line 7), the algorithm verifies that the message is not already in the list L_{src}. Lines 8 to 15 are executed, if the sequence number of the received message corresponds to the next, expected number $lc_{src} + 1$. After delivering the message (line 9), the sequence number counter lc_{src} is incremented, and lines 11 to 14 check for messages delivered out of sequence. Line 16 discards messages whose sequence number is smaller than lc_{src}, or already included in the set L_{src}.

7.3.3 Implementation of At-Most-Once Delivery

This subsection describes the key ideas of the *at-most-once* delivery implementation. In contrast to the implementation of the same *at-most-once* guarantee inside a group, there are no views and

[6]Note that a process which crashes and recovers will have a new incarnation number and is thus a *new* process with *new* message identifiers.

sequence numbers alone are not sufficient. The reason is simple: *at-most-once* delivery implies that a message does not necessarily have to be received and delivered. To illustrate this, imagine that every second message is lost[7]. Applying the algorithm of Figure 7.1, the list L_{src} (list of sequence numbers of messages delivered out of sequence) would increase indefinitely, because lines 8 to 15 of Figure 7.1 would never have been executed. Further, there are no view numbers (see Section 7.3.2) outside the group context which means that the view number mechanism cannot be used to discard old messages.

[Liskov 91] describes a solution to overcome this problem by introducing time stamps on messages. The described implementation is based on loosely synchronized clocks [Liskov 93]. The Phoenix implementation, described below, is based on the ideas in [Liskov 91], but instead of synchronized clocks, it is based on remote clock approximation. Section 7.3.4 describes how remote clocks are approximated.

For the description of the algorithm it is sufficient to assume that the remote clock approximation algorithm described later ensures that the difference $\delta_p(q)$ between a local and remote clock of two processes p and q can be calculated[8], at any moment in time. If a process p adds the value $\delta_p(q)$ ($\delta_p(q)$ can be negative) to the time stamp of a message m from p to q makes q think that p's clock is synchronized with that of q. This allows q to compute an approximation of the time $t_{trans}(m)$ during which the message m was in transit between p and q. It can be further assumed that the maximum error of this approximation is never greater than β[9]. We will see later that this β in the actual implementation is equal to the average round-trip-time of a message exchange.

Idea

The idea consists of distinguishing between *recent* and *old* messages, and only *recent* messages are considered for replica recognition[10]. The distinction between *old* and *recent* is based on the time stamps of the messages and the approximation of the transit time t_{trans} of the messages. A message m is considered as *recent* if $t_{trans}(m) < \tau$. Similarly, a message m is considered *old* if $t_{trans}(m) \geq \tau$.

The algorithm can be outlined as follows:

- only *recent* messages are subject to duplicate recognition, whereas *old* messages are discarded immediately;

[7] This scenario is possible as a Phoenix *client* is allowed to cancel the send of a message to a *member* at any time, and at a later point to issue another message to the same *member*.

[8] This approximation would allow synchronization of the clocks between two processes at any time, but an explicit synchronization is not necessary for the protocol presented here.

[9] [Liskov 91] states that this skew β can be held as small as 100 ms, even in wide area networks.

[10] Comparing with Section 7.3.2, there is a *view* of *recent* messages; this view changes with time.

- sequence numbers of *recent* messages[11] are checked against a list of sequence numbers of *recently* received messages in order to detect replicas and discard them;
- sequence numbers of messages which have become *old*, are removed from the list, as received replicas with *old* time stamps are discarded anyway.

This algorithm is safe regarding *at-most-once* delivery, but the probability that a message is rejected, although it was received for the first time is not zero. However, by choosing τ equal to some reasonable upper bound for the maximum time in transit of a message, this probability can be reduced to a very small percentage; the greater τ, the smaller the percentage. This is not the only parameter influencing the choice of τ.

Rejecting a message m whose $t_{trans}(m) \geq \tau$ allows the elimination of all sequence numbers in the list of sequence numbers of messages which have been sent before m. This can also be interpreted in the following way: a sequence number of a message received more than τ time units before, can be discarded. Anyway, the infinite growth of the sequence number list is avoided. Thus, the choice of τ is a trade-off between the probability of rejecting *new* messages, i.e. messages never received before, and memory consumption for the list of sequence numbers.

Algorithm

Figure 7.2 gives the algorithm for *at-most-once* delivery guarantee. For this purpose, each message m sent by p to q is time stamped with the value $t^s + \delta_p(q)$, where t^s is the local clock value to p when the message is sent for the first time. This value makes q think that the clock of p is synchronized with the clock of q and makes the computation of $t_{trans}(m)$ simple. A message is time stamped when it is sent for the first time (further retransmissions do not modify this value). Time t^r in Figure 7.2 is the clock value at the destination each time a message is received. This reception time t^r is compared with the time stamp $t^s + \delta_p(q)$ contained in message m. If m is not an *old* message, the sequence number of m is matched to the list M_{src} of sequence numbers of *recently* received messages from process src in order to detect doubly received messages.

In Figure 7.2 line 5 computes $t_{trans}(m)$, the time the message has been in transit. Line 6 checks this transmission time against τ in order to determine whether it is an *old* or a *recent* message. In the case of a *recent* message, lines 7 to 14 are executed where line 8 checks if the message has already been received and delivered once. This is done by looking for the sequence number of the message in the list of received sequence numbers M_{src}. If not present, the message is delivered and its sequence number, as well as the time at which the sequence number has to be deleted

[11] The time stamp could have been used as a sequence number, but as sequence numbers are already generated for the *group-at-most-once* delivery, the algorithm is based on sequence numbers. Further, a time stamp does not indicate, through an incarnation number, the failure of the sending process.

```
1   var M : array of sequence_nbr_sets
2
3   procedure message_delivery_callback (m)
4   begin
5       t^s := timestamp(m); t^r := gettime(); t_trans = t^r − t^s;
6       if t_trans < τ then
7           /* check if the message ID is already in the received list */
8           if seq(m) ∉ M_src then
9               element.num := seq(m);
10              element.time := t^s + τ + 2 β;
11              M_src := M_src ∪ {element};
12              deliver(m);
13          end if
14          send_positive_acknowledgement();
15      else
16          discard(m);
17          send_negative_acknowledgement();
18      end if
19
20      forall element ∈ M_src do
21          if t^r > element.time then
22              M_src := M_src − {element};
23          end if
24      end for
25  end
```

Figure 7.2: At-Most-Once Delivery Algorithm

from the M_{src} are inserted the list. Note that 2 β is added (remember that β is the maximum error of the remote clock approximation) to ensure that the error of the approximation of $\delta_p(q)$ does not influence the correctness of the algorithm. Line 14 sends a positive acknowledgement to the sender in order for the sender to stop retransmission. In lines 16 and 17, messages which have taken longer than τ are discarded and a negative acknowledgement is sent back, telling the sender to stop retransmission[12]. The acknowledgement is negative, because the message was not delivered[13]. Finally, lines 20 to 24, executed at each reception of a message, delete all sequence numbers in the list M_{src} whose messages would anyway be discarded due to the condition in line 6.

Correctness

The algorithm for *at-most-once* delivery guarantees for any positive values τ and β, that a message is delivered at most once. The proof is obvious since the algorithm discards messages

[12]Note that the negative acknowledgement is not necessary, but it avoids useless retransmissions of messages which are discarded.

[13]It is possible that a discarded message has already been delivered at an earlier point in time, but due to lost acknowledgements, the message was still retransmitted by the sender.

before discarding their sequence numbers in the list if any.

The Parameter τ

As seen above, τ is the maximum time a message may be in transit in order to be considered for replica recognition. The choice of τ depends on the trade-off between probability of discarding of a message received for the first time, and the memory consumption of the sequence number list. If τ is chosen small, *at-most-once* delivery is still guaranteed, but the probability of discarding messages, received for the first time, is increased. If τ is chosen great, the probability of first message discarding becomes very small, but the sequence number list can become very large. In the actual implementation, the value of τ has been chosen for each pair of processes p and q independently (thus an extended notation might be $\tau_p(q)$) and proportionally to β, which as we will see in the next section, is set equal to the mean average round-trip-time $rtt(p, q)$ between p and q (thus an extended notation might be $\beta_p(q)$)[14]. In this way, sequence numbers from slower destinations are kept longer as their transit time is also greater. To summarize, the shorter the round-trip-time is, the smaller is the probability that a message is received late, and the earlier the sequence number of a message can be discarded. This also is important, as in local area networks it is easy to generate a lot of traffic per second and thus memory should be freed more quickly.

7.3.4 Remote Clock Approximation

The remote clock approximation for reliable communication is based on ideas developed for clock synchronization. However, we will see that for the reliable communication layer, clock synchronization is not required, and that it is sufficient to have an approximation of the remote clocks. For completeness a short overview of clock synchronization is given and afterwards the protocol for remote clock approximation is developed.

Clock Synchronization

Clock synchronization has been extensively studied in the literature. Among the large number of protocols, some are particularly interesting, as they provide clock synchronization in large scale networks, e.g. in the Internet. The network time protocol NTP [Mills 91, Mills 92] is one among several protocols providing loosely synchronized clocks. Loosely synchronized in the context of NTP means that clocks can differ from host to host, but an upper bound β can be given on the maximum difference between the clocks. NTP is based on master time servers

[14] In the actual implementation 100 has been taken as the factor between β and τ. The following example should illustrate the choice: if the average round-trip-time is equal to 0.5 seconds (this value is reasonable for many large scale destinations, whereas [Liskov 91] gives an even lower value of 0.1 seconds), τ equals to 50 seconds. These 50 seconds correspond to the maximum value observed for first message reception during tests with different hosts on the Internet. The tests consisted of sending a message using the service *ping* and waiting for the answer.

basing the local clocks on hardware means as atomic clocks [Allan 74], radio clocks [Blair 74], GPS [Ananda 88], etc.. Other time servers are clients of these master servers and organized in a hierarchical scheme in order to be scalable.

Remote Clock Approximation

The remote clock approximation protocol is based on the general ideas of NTP, but the presented ad-hoc implementation is much simplier and less restrictive. For the *at-most-once* delivery implementation, described in Section 7.3.3, it is not necessary that all clocks are synchronized throughout all processes, but pair-wise approximated between communicating processes[15]. Thus for each destination q a process p computes an approximation of q's clock and thus an approximation $\delta_p(q)$ of the difference between these clocks. This pair-wise approximation makes the remote clock approximation protocol independent of a master time server, eliminating a component of NTP which has only very small support for fault-tolerance. Moreover, this makes the reliable communication layer independent of the availability of specialized hardware and the operation of NTP. The value $\delta_p(q)$ will be added to every time stamp of p generated on a message for q before it is sent, and makes q think that p's clock is synchronized with that of q.

The Protocol

In order to calculate the value $\delta_p(q)$, p and q have to exchange at least one message pair. This approximation will be improved with each subsequent message exchange. To be able to have a consistent approximation, the protocol assumes further, similar to NTP, that clocks are monotonously increasing and that the difference in frequency is small[16].

The notation $t^s_{p,i}$ represents the local time of p at which process p has sent (s for send) message m_i (see Figure 7.3). Process p tags a message m_i with its local clock $t^s_{p,i} = 2$ and sends it to q. When q receives the message on $t^r_{q,i} = 27$ (r for receive) it sends a message m_j back[17] containing the time stamp $t^s_{p,i} = 2$ of message m_i and its current local clock $t^r_{q,i} = t^s_{q,j} = 27$ when message m_j is sent. When p receives the message m_j, it stores the current local clock in $t^r_{p,j} = 7$ and can make the following deductions:

- p knows the round-trip-time $rtt(p, q)$ between p and q ($= 5$ in Figure 7.3);
- p knows the time $t^s_{q,j}$ on process q when m_j was sent ($= 27$ in Figure 7.3).

This allows p to compute the approximation of the difference $\delta_p(q)$ between the clock of p and

[15]NTP was developed for distributed applications exploiting synchronized clocks. [Liskov 93] gives a good overview of such types of applications, e.g. timed authentication tickets in Kerberos [Steiner 88], cache consistency [Gray 89].

[16]Typical skew is no more than 1 second/day.

[17]Message m_j could be considered as the acknowledgement for message m_i, but any message back to p could be used if its send is immediately following the reception of m_i.

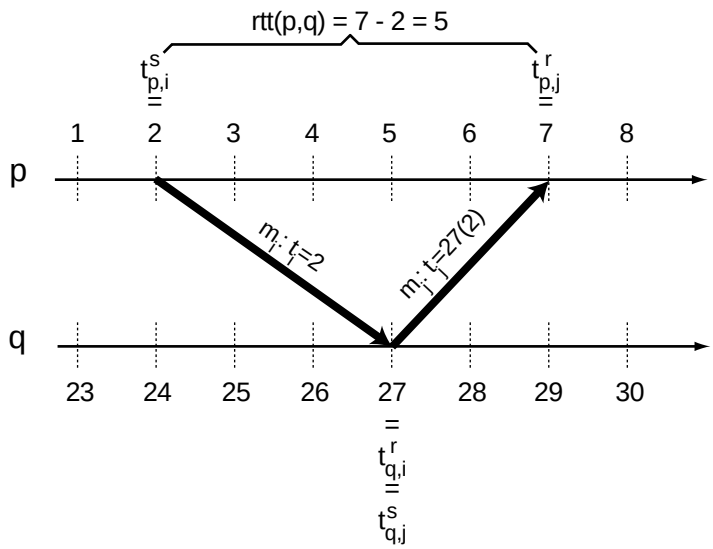

Figure 7.3: Approximation of p's Clock at q

q at p[18].

$$\delta_p(q) = t^s_{q,j} - (t^s_{p,i} + \frac{t^r_{p,j} - t^s_{p,i}}{2}) = t^s_{q,j} - (t^s_{p,i} + \frac{rtt(p,q)}{2})$$

assuming that m_i and m_j have the same transit time[19]. With the values of Figure 7.3:

$$\delta_p(q) = 27 - (2 + \frac{7-2}{2}) = 22.5$$

This approximation $\delta_p(q)$ is based on round-trip-times of messages and as these round trip times vary, especially in large scale networks, the protocol has to take into account this variance in the round-trip-time. Thus, the values $rtt(p,q)$ and $\delta_p(q)$ are not fixed but weighted averages on the set of the last x successful message exchanges and piggy-backed on successive messages.

7.4 Flow and Congestion Control

One important issue in reliable communication is flow and congestion control. Flow and congestion control influence how messages are sent and retransmitted. The particularity of flow control is that only the load at the destination is controlled. The aim of flow control is not to send messages to some destination, if there are already enough messages in transit. What

[18]Note as each process does its approximation independently $\delta_p(q)$ is not necessarily equal to $\delta_q(p)$.

[19]In Figure 7.2 2 β has been added to the transit time making the algorithm correct even if $t_{trans}(m_i) \gg t_{trans}(m_j)$ or vice-versa.

enough means and how it is quantified and controlled is discussed in Section 7.4.1. Congestion control also takes the load of the network into account. Varying network loads lead to varying delays on messages. The aim of congestion control is to avoid retransmitting a message, before an acknowledgement could have arrived, and to avoid transmitting additional messages on an already loaded network. Section 7.4.2 explains in detail how congestion is *measured* and (re)transmissions controlled.

This is not an exhaustive analysis of these two aspects in controlling the sending of messages, but it shows two important points in order to implement efficient reliable communication.

7.4.1 Flow Control

Flow control is applied to the sender and consists of limiting the traffic to some destination. The implemented protocol, called *simple window protocol*, verifies that there is never more traffic ongoing to some destination than there is space in the window. Upon reception of acknowledgements, which means that the destination has delivered the acknowledged messages, the window is filled up again, if there are messages waiting to be sent[20]. This protocol can be compared with the sliding window protocol of TCP/IP [Comer 88], but the protocol presented here differs from TCP/IP in that it does not guarantee *first-in-first-out* order on messages as in TCP/IP. The sliding window protocol of TCP/IP retransmits all messages starting from the one that was not acknowledged, independently of whether there are later messages in the window, which have already been acknowledged. The reliable communication layer only retransmits lost messages and delivers all messages received immediately (except if *first-in-first-out* order has been specified).

One interesting effect of this simple window protocol is that it helps to keep sequence number lists small at the destination (see Section 7.3.2). As the sender is not transmitting more messages than there is space in the window, the destination only keeps track of a limited number of sequence numbers.

7.4.2 Congestion Control

Congestion occurs if there is too much traffic on some communication link. This congestion is not only due to the sender and the destination, but also due to other processes having in common some of the communication links between the considered sender and destination. At the sender, congestion can be noticed by considering the percentage of messages lost and the increase of round-trip-times for a send-acknowledgement pair. If a network or destination is loaded, messages are lost more often and the transfer time for a message increases. Congestion

[20] The space available in a window can be quantified e.g. the maximum number of messages or the maximum number of bytes in transit.

control tries to exploit the available bandwidth at a maximum, and to be as resilient as possible when congestion occurs, by not overloading an already loaded network.

There are two parameters which determine the traffic generated by a sender:

- frequency of retransmission of messages
- window size of the simple window protocol

Retransmission Strategy

A first rule for retransmissions is not to retransmit a message before its acknowledgement could have arrived. For this purpose the retransmission algorithm takes into account round-trip-times for a send/acknowledgement pair, averaged over the last couple of messages.

Retransmission is necessary when a message or its acknowledgement have gotten lost. If a retransmitted message also gets lost, the network or destination could be overloaded. Thus, in order not to add additional traffic to an already overloaded destination, the period between retransmissions of the same message is increased after each retransmission. In the actual implementation, an exponential back-off has been choosen, i.e. for the first retransmission of a message, the sender waits only once the round-trip-time, whereas for the second, it waits two times the round-trip-time, then four, eight and so on.

Window Size Handling

Concerning the window size in the simple window protocol, an averaged and weighted percentage of message loss is used to increase or to reduce the window size. As long as no message is lost, the window is growing by some amount, e.g. one message more each time an acknowledgement is received. If one message gets lost, the window size is halved. This loss can be due to the current load of the network, and/or a load at the destination independent of the load generated by the sender. If more than half of the messages get lost, the window size is reset to 1.

These two simple algorithms allow an efficient control of congestion, where in the normal case, exploitation of maximum bandwidth is tried with large window sizes and rare retransmissions. In case of congestion, and thus increased message loss, window size is small and in case of serious congestion window size is one message and retransmissions are delayed with exponential back-off. Therefore, the traffic generated due to retransmissions to a failed process becomes very small.

Finally, averaged values on round-trip-times (already used in Section 7.3.4 for remote clock approximation) of messages and message loss allow the congestion control to react in a timely manner to changing loads on the network and at the destination.

[Comer 88], [Hickey 95], [Jacobson 88] describe more sophisticated flow and congestion control mechanisms.

7.5 Message Fragmentation

The message size of UDP is limited to the size of the send buffer. This limit is inherited by the socket interface and the routing layer. Further, on the Ethernet, the size of packets is even more limited to a size called the MTU (maximum transmission unit) which is system dependent, but usually smaller than 8 kB. As the UDP buffer can be greater than the MTU, UDP fragments messages in smaller parts in order to suit the MTU. The limit of the UDP message size (typically 64 kB) is insufficient for some applications, and there is a need for greater message sizes, e.g. for a file or state transfer. For this reason the reliable communication layer allows the sending of messages of any length. The possible fragmentation of a message is hidden and an overlying layer does not see any difference between the sending of fragmented and non-fragmented messages.

Outgoing messages are fragmented transparently by the reliable communication layer, if necessary. These fragments are sent to the destination process using the same primitive as the initial send, so each message fragment is managed for retransmission as is any other message. The destination recognizes incoming, fragmented messages. Using fragment numbers, contained in the header of the received messages, the receiver is able to reassemble the fragments back into complete messages. Once all fragments have been received, and only then, the complete message is delivered as a whole.

Each fragment is acknowledged independently by the destination, whereas the sender collects the acknowledgements until it has received acknowledgements for all fragments. Then, one acknowledgement is delivered to the upper layer, at the sender.

One problem occurs due to the *at-most-once* delivery guarantee. As the sequence of messages received can have holes, it is possible that not all fragments have been received by the destination. As the remaining fragments are not retransmitted, the fragments, already present in the destination buffer, will never be delivered. Similar to UDP fragmentation, non-delivered message fragments are deleted after some timeout.

For fragmented messages, it is necessary to introduce a higher level acknowledgement for the delivery of the whole message. Although all fragments could have been acknowledged correctly by the destination, it is possible that fragments would have been discarded before the message was complete at the receiver. Thus, upon the delivery of a message that was fragmented, a high level acknowledgement is sent to the source. At the source, this is the only acknowledgement that will be delivered to overlying layers.

For the implementation of the *at-most-once* guarantee, the discarding of message fragments is based on a timeout, which has been chosen equal to the value τ of Section 7.3.3. In the normal case this value gives all fragments enough time to arrive at the destination, before the first fragments get discarded (see Section 7.3). For the implementation of the *group-at-most-once* guarantee, message fragments are discarded through the cancel primitives when a view change occurs.

7.6 Reliable Communication Layer Programming Paradigm

In contrast to other reliable communication layers, the reliable communication layer of the Phoenix system provides new semantics in case of communication problems. In other communication layers, the programmer receives an error code when the send of the message did not succeed. Moreover, the programmer has to specify a timeout after which the send should be discarded. Failure of the destination is handled either by an error code in blocking send calls, or in interrupt handlers, signalling problems in communication. The Phoenix reliable communication layer all failure and non-failure cases are handled by the same mechanism, namely callbacks.

First, the two non-blocking send primitives are specified:

- group_reliable_send (dst, msg): this primitive is used to send message inside a group with *group-at-most-once* delivery guarantee;
- reliable_send (dst, msg): a message sent with this primitive is delivered with *at-most-once* guarantee.

The two send semantics are distinguished only in the way they ensure *at-most-once* delivery.

For message cancelling there are two primitives:

- cancel_send (msg): cancels the send and retransmission of a message *msg*.
- cancel_sends_to_dest (dst): this call cancels the send and retransmission of all messages to the specified destination *dst*.

These cancel primitives can be used on any message sent with the reliable send primitive, whereas for a group reliable send, the call of one of these primitives implies not communicating anymore with this destination[21]. Chapter 8 shows how this function is exploited.

[21] In the Phoenix system there exists a resynchronization primitive, which allows a process of group, which has been excluded from a group and to whom messages have been cancelled, to join a group again without changing its process identifier.

In order to receive messages and feed-back on the success or the failure of sends, an overlying layer has to provide, upon initialization of the reliable communication layer, different callbacks (these procedures will be called by the reliable communication layer):

- message_delivery_callback (proc): this callback, installed in the reliable communication layer, is called when a message is received; the received message is passed as a parameter to the callback.

- message_acknowledgement_callback (proc): this callback is called when an acknowledgement for a message is received.

- undelivered_message_callback (proc): this callback is called when a message is not acknowledged by some destination after several retries. The not successfully transmitted message is passed as a parameter to the callback. Message retransmission is not stopped.

When an overlying layer or application sends a message to a destination, using the two provided *reliable_send* primitives, either the *message acknowledgement callback* or the *undelivered message callback* will be called telling that there is a communication problem.

There are two interesting things to mention: as the reliable communication layer fully takes care of flow and congestion control, the programmer does not have to worry about timeouts: the system will automatically adapt an appropriate timeout, guaranteeing an adequate delivery quality depending on the current system load and the responsiveness of the destination.

Note that if no acknowledgement is received, this does not mean that the message has not been delivered at the destination, but that either the message or the acknowledgement got lost.

The *undelivered message callback* is used for both send semantics (inside a group and outside). To stop the retransmission of a message which is not acknowledged, the *cancel_send* primitive is used[22]. For reliable sends outside the group context, the *cancel_send* primitive is implicitly called during the execution of the *undelivered message callback*. For group reliable sends, messages have to be cancelled explicitly by an overlying layer or the application (see Chapter 8 for further details).

[22]This also makes sense for messages, sent outside a group, for example to cancel a request to some slow site, because a faster site has already responded.

7.7 Discussion

7.7.1 Why a New Communication Layer

The research on reliable communication is not new, and there are many different implementations. Chapter 3 already gave an overview of reliable communication layers. The most similar protocol to the reliable communication layer is TCP/IP [Comer 88]. Although, TCP/IP provides reliable point-to-point communication, its interface is not well suited for the needs of Phoenix.

Here is a comparison between TCP/IP and the reliable communication layer (RCL):

protocol	TCP/IP	RCL
connection	stream-oriented	message oriented (keeps message bounds)
# of simultaneous connections	limited by available descriptors	unlimited
connection establishment send primitive	explicit negotiation blocking	no connection required non-blocking
FIFO	only	optional
flow control	yes	yes
infinite retransmission	no	yes
congestion control	yes	yes
failure	signal handling	event

Figure 7.4: Comparison of TCP/IP and RCL

There are other reliable communication protocols, e.g. RDP [Velten 84], AFDP [Cooperstock 95], and RMP [Whetten 94], which are discussed in Chapter 3, but all these protocols are specialized for some particular context or task, often unsuitable for the concerns of reliability in large scale and of fault-tolerance required for the Phoenix system. Thus, Phoenix has its own reliable communication layer, providing a simple interface, including non-blocking primitive calls, an event driven programming model based on callbacks, and the integration of failure management in the programming paradigm.

7.8 Summary

This chapter gives the specification of a reliable communication layer, well suited for implementing fault-tolerant protocols on top of it. Different sending strategy and delivery guarantees provide a flexible interface for all overlying communication layers. The implementation part of this chapter shows the problems, trade-offs and drawbacks of the different send and deliv-

ery strategies, and a section about flow and congestion control ensures a reasonable use of the destination and network resources.

In Chapter 8, we will see how the view synchronous communication layer makes use of the *group_reliable_send* primitive and its *at-most-once* delivery guarantee. On the other side, the *at-most-once* delivery guarantee, as well as the *undelivered message callback* of this layer are exploited in Chapter 10, where the application programming interface is described.

Chapter 8

View Synchronous Communication Layer

8.1 Introduction

The *view synchronous communication* layer provides a reliable multicast primitive to members of a group. The current members of a group build together the current view of the group, and as long as this view does not change, there is no difficulty in providing the reliability of the multicast. However, if processes fail, a message multicast by the crashed process might not be received by all the surviving processes: the simple reliability definition of Chapter 7 is not sufficient anymore. This chapter introduces a paradigm which allows a definition of *reliability* in the context of process groups and in case of process failures. This paradigm is called the *view synchronous communication* paradigm, first introduced in [Birman 87]. The paradigm allows the smooth integration of leaves and joins of processes.

The *view synchronous communication* layer provides a *view synchronous* multicast primitive satisfying the *view synchronous communication* paradigm. Furthermore, it manages the views and the messages for each group. For each group, the layer can be either in normal mode or in view change mode. In normal mode the *view synchronous* multicast is executed using the primitive described in Section 8.3; in view change mode the layer does not multicast messages anymore, but tries to guarantee the *view synchronous communication* paradigm, by forwarding messages and defining a new view accordingly; this is discussed in Section 8.4.5.

8.2 Definitions

8.2.1 The VSC Paradigm

The *view synchronous communication* paradigm was first introduced in [Birman 87] and is summarized by the following informal definition:

"Given two consecutive views V_i and V_{i+1},
all processes in view $V_i \bigcap V_{i+1}$ have delivered
the same set of messages during view V_i."

For a formal definition let us consider the view change from view V_i to V_{i+1}. Consider two processes p_x and p_y members of view V_i. Further, during view V_i process p_x delivers the set of messages $M_{i,x}$ and p_y the set of messages $M_{i,y}$.

View synchronous communication can be defined by the two agreement properties:

(A1) Agreement on the next view: if p_x and p_y both deliver the next view V_{i+1}, then they agree on this view:

$$V_{i+1,x} = V_{i+1,y} = V_{i+1}$$

(A2) Agreement on the set of messages: If p_x and p_y both deliver the next view V_{i+1}, then they agree on the set of messages delivered in view V_i:

$$M_{i,x} = M_{i,y} = M_i$$

In order to avoid the trivial solution where the new view is always the empty set, the following non-triviality condition is needed:

(NT) Non-triviality: crashed, suspected or leaving processes are eventually removed from a view, and new and not suspected processes that want to join are eventually included in a view.

These two agreement conditions, together with the assumption that processes of the initial view V_0 agree on V_0, lead the processes to agree on a sequence of views, and on the set of messages delivered in each view.

This definition of *view synchronous communication* represents the primary partition implementation of the *view synchronous communication* paradigm as only one unique sequence of views is considered, whereas other possible definitions are discussed in Section 8.7.1. Section 8.4.5 presents a protocol implementing *view synchronous communication* providing the primary partition property.

8.2.2 Events

The definition of the *view synchronous* communication paradigm, makes reference to message delivery. In order to be able to define the delivery of a message, events are introduced.

Consider two events e_i^p and e_j^p occurring on the same process p. If event e_i^p occurs before e_j^p, we note:

$$e_i^p \rightarrow e_j^p$$

Having this *happened-before* relation, the formula BEFORE considering two events e_i^p and e_j^p can be defined.

Definition

The formula

$$\text{BEFORE}(e_i^p, e_j^p)$$

holds if the two events have occurred, and $e_i^p \rightarrow e_j^p$.

8.2.3 Message Multicast, Reception and Delivery

The multicast, the reception and the delivery of a message can be seen as events. For a layer L, each message is sent by calling the corresponding primitive of layer $L-1$, is received from the underlying layer $L-1$ and delivered to the overlying layer $L+1$ during some view V_i. The multicast, the reception and the delivery of a message m during view V_i are associated with their corresponding events:

$$mcast_i^p(m) \quad rcv_i^p(m) \quad dlv_i^p(m)$$

Definitions

Formula MCAST holds as soon as message m is multicast by p in view V_i:

$$\text{MCAST}_i^p(m)$$

holds when the corresponding event $mcast_i^p(m)$ has occurred. Formula RCV holds as soon as the message reception event has occurred. Thus, if a process p receives message m during view V_i, the formula:

$$\text{RCV}_i^p(m)$$

holds. Similarly, formula DLV holds as soon as the delivery event has occurred. Thus, if a process p delivers message m during view V_i, the formula:

$$\text{DLV}_i^p(m)$$

holds. The installation of a view V_i can be seen as the delivery of a special message m_{V_i} in view V_{i-1}. This allows one to define the ordering of messages by views in which they are delivered:

$$\forall m : \text{DLV}_i^p(m) \Rightarrow \text{BEFORE}(dlv_{i-1}^p(m_{V_i}), dlv_i^p(m)) \ \wedge \ \text{BEFORE}(dlv_i^p(m), dlv_i^p(m_{V_{i+1}}))$$

In other words, every message m delivered in some view V_i is delivered after the delivery of view V_i and before the delivery of view V_{i+1}.

8.2.4 Correct and Incorrect Processes

The definitions of what is a correct and what is an incorrect process are based on the definition of the message delivery, and the fact that the view installation is also considered as the delivery of a special message.

Definition

A process p member of a view V_i is correct during view V_i, if

$$\text{DLV}_{i-1}^p(m_{V_i}) \ \wedge \ \text{DLV}_i^p(m_{V_{i+1}})$$

holds.

The set $\mathcal{C}(V_i)$ of correct processes during a view V_i is defined as follows:

$$\mathcal{C}(V_i) \stackrel{def.}{=} \{p \mid \text{DLV}_{i-1}^p(m_{V_i}) \ \wedge \ \text{DLV}_i^p(m_{V_{i+1}})\}$$

Note that in order to be a correct process of view V_i, it is not sufficient to be a member of view V_{i+1}, but also necessary to have delivered view V_{i+1}. Further a correct process in view V_i must also have delivered view V_i, which excludes processes that joined the group during view V_i, i.e. are only members of view V_{i+1}. Finally, as incorrect processes are not members of view V_{i+1}, they never deliver view V_{i+1}.

The *view synchronous communication* layer cannot guarantee that a process, which is a member of a view V_{i+1}, really delivers that view, because a process can fail before delivering a new view.

Illustration

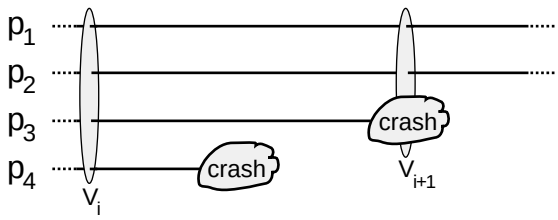

Figure 8.1: Correctness of Processes

Considering Figure 8.1, process p_4 is not a correct process in view V_i since it fails during view V_i. Process p_3 is not considered as correct either, although it is a member of the view V_{i+1} as it fails before delivering view V_{i+1}. For the processes p_1 and p_2, however, it is impossible to decide whether process p_3 failed before delivering the new view V_{i+1} or immediately after.

8.2.5 View Synchronous Multicast

Having defined $\mathcal{C}(V_i)$, the definition of the *view synchronous* multicast is as follows:

Definition

For a message m multicast in view V_i using the *view synchronous* multicast primitive, the following property is guaranteed:

$$\exists p \in \mathcal{C}(V_i) : \textsc{dlv}_i^p(m) \Rightarrow \forall q \in \mathcal{C}(V_i) : \textsc{dlv}_i^q(m)$$

or in other words, if one correct process in view V_i has delivered m while in view V_i, then all correct processes in view V_i have delivered message m in V_i before delivering V_{i+1}. There are no guarantees about the messages delivered by incorrect processes.

In order to avoid the trivial case where no message is ever delivered in any view V_i, the following liveness property is satisfied:

$$\exists p \in V_i : \textsc{mcast}_i^p(m) \Rightarrow \textsc{dlv}_i^p(m)$$

If p is a correct process, then every correct process will deliver the message in V_i.

8.2.6 Message Stability

Processes members of a view V_i of a group g communicate by sending multicasts to all the processes in view V_i. We will see below that it is important for a sender $s \in V_i$ of a message m to know whether all other members of V_i have received message m. A message received by all processes of some view V_i is called a *stable* message.

Definition

A message m is stable in view V_i, if

$$\forall q \in V_i : \text{RCV}_i^q(m)$$

is satisfied, where formula $\text{RCV}_i^q(m)$ holds, if process q has received message m in view V_i by the *view synchronous communication* layer. Note that in order to be stable, it is sufficient that message m has been received by all processes; this does not necessarily mean that the message has been delivered to the overlying layer[1].

The *view synchronous property* is trivially satisfied for stable messages. Therefore, to ensure *view synchrony*, the *view synchronous communication* layer only has to keep track of non-stable messages.

8.3 Implementation of the View Synchronous Multicast Primitive in Normal Mode

8.3.1 Overview

The multicast of a message to the current view of a group is implemented by sending a copy of the message to each of the processes of the current view. This is implemented by using the *group_reliable_send* primitive, provided by the reliable communication layer. The multicast primitive adds the following information to the message:

- *current view number*: this allows a receiver to distinguish between messages for past, current and future views[2];

[1] This will be exploited in Chapter 9, where two messages have to be received in order to deliver one.

[2] A receiver can receive a multicast for a view, which is not yet installed at the receiver.

- *group identifier*: this allows a receiver to distinguish between multicasts for different groups;
- *multicast sequence number* and *source id*: this information allows the recognition of duplicate messages.

These three fields attache a system-wide unique identifier to each multicast. A process receiving a multicast m has to keep a copy of the message until m is stable. The *view change* protocol (see Section 8.4.5) will stabilize unstable messages when necessary. As the *view change* protocol is only launched when necessary, the accumulated messages multicast during a view can occupy a lot of memory. Further, the more messages are unstable, the more traffic will be generated during the *view change* protocol in order to stabilize these messages. Thus messages are also stabilized during the normal mode of execution in order to limit this memory consumption. The next subsection presents a protocol for message stabilization.

The *view number*, included in a message, is exploited at the reception of the message. The receiver of a multicast compares the *view number* with the current view number and acts accordingly: messages for past views are discarded, messages for the current view are delivered, and messages for future views are buffered until the corresponding view is installed. Upon view installation a process checks if there are buffered messages for this view and delivers them.

8.3.2 Message Stabilization Protocol

The definition of the *stable* property of a message in Section 8.2.6 is valid only for a global observer of a system. For the local detection of this property, two points of view have to be considered:

- *Sender*: a sender s knows that a multicast m has been received by all members of a view, when it has received an acknowledgement[3] for this message from each member;
- *Receiver*: a receiver r cannot know when a multicast m has been received by all members of a view V_i, as the acknowledgement is only sent to the sender s of the multicast.

The aim of the stabilization protocol is to provide the knowledge of when a message m is received by all destinations, to the sender and the receivers of m.

Stabilization Protocol

The protocol has three steps:

[3]Remember that the *group_reliable_send* primitive of the reliable communication layer generates an acknowledgement at the sender, when the message has been successfully received.

1.) a process p multicasts a message m to the members of a view V_i;

2.) each member of view V_i sends back an acknowledgement to p;

3.) as soon as p has received an acknowledgement from every member in V_i, it multicasts a message $stable(m)$ to the members of V_i, and discards message m.

Note that the acknowledgements of step 2.) are not explicit calls to the *send primitive*, but automatically generated by the *reliable communication layer*. Further, it is possible to piggy-back stability information about one or several messages on future multicasts; thus no additional messages are necessarily issued.

8.4 Implementation of the View Change Protocol

8.4.1 Overview

As seen in the introduction to this chapter, the view synchronous communication layer is either in normal mode or in view change mode. During normal mode messages are multicast and stabilized by this layer. In case of a process failure or a communication failure the dynamic routing layer will suspect some processes and generate a new reachability set. Upon delivery of a new reachability set the view synchronous communication layer changes to the view change mode and launches the *view change* protocol, which stabilizes all remaining non-stable messages, and defines a new view. Upon termination of the *view change* protocol, the new view will be installed with the guarantee that all remaining non-stable messages have been stabilized at the members of this new view. With the installation of the new view the layer changes back to normal mode.

The *view change* protocol is based on a consensus protocol. Section 8.4.2 shows how a consensus problem can be defined and how the *view synchronous communication* "problem" can be seen as a variation of the consensus problem. The *view change* protocol is based on the consensus protocol of Chandra and Toueg [Chandra 95]. The particularity of this protocol is that it solves consensus in an asynchronous system using the failure detector $\diamond\mathcal{S}$ (read "eventually strong failure detector", see Section 8.4.3). The properties of this failure detector $\diamond\mathcal{S}$ are given in Section 8.4.3, followed by Section 8.4.4, where a rough description of the consensus protocol of Chandra and Toueg based on the failure detector $\diamond\mathcal{S}$ is given. Section 8.4.5 shows how the *view change* protocol has been implemented using the consensus protocol.

8.4.2 From Consensus to VSC

In order to satisfy the *view synchronous communication* paradigm, processes have to agree on the set of messages not yet stabilized and the set of processes defining the new view. As this is an agreement or better a consensus problem, it is shown how the consensus can be defined and how *view synchronous communication* can be interpreted as a consensus problem.

Consensus

The consensus problem is defined on a set of processes each having an initial value, and these processes have to decide on a common value that has to be the initial value of one of the processes.

More specifically, a protocol which solves the consensus problem has to fulfill the following properties:

Termination:	each correct process eventually takes a decision.
Agreement:	if two correct processes take a decision, they will take the same decision.
Validity:	if a process decides on d, then d was proposed by some process.

Considering the termination property, a protocol solving consensus ensures that some decision is eventually taken.

VSC as a Consensus Problem

Having the basics of the consensus problem, it is now possible to interpret the *view synchronous communication* problem as a consensus problem. In the *view synchronous communication* problem, the decision value is not a single value, chosen out of a set of initial values, but

- a set of messages to deliver in the current view,
- a set of processes defining the next view[4].

Figure 8.2 gives a typical example for the consensus value.

Reconsider the three properties that a protocol has to fulfill in order to solve consensus. Considering the *view synchronous communication* problem these properties can be expressed in the following way:

[4]Note that the next view is defined as a set of processes, but that a process, member of the next view, can fail just before delivering the messages and installing the new view. This case will lead to another view change.

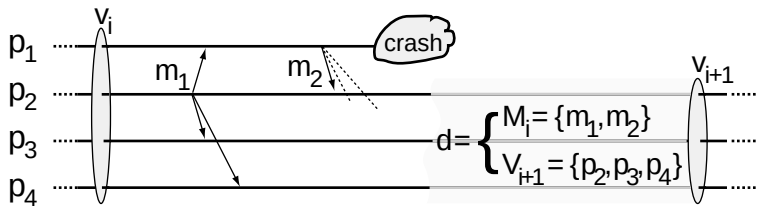

Figure 8.2: Decision Value

Termination: each correct process eventually delivers a set of messages M_i in view V_i and installs a new view V_{i+1}.

Agreement: if two processes p_x, p_y deliver each a set of messages $M_{i,x}, M_{i,y}$ in view V_i and install new views $V_{i+1,x}$ and $V_{i+1,y}$, then $M_{i,x} = M_{i,y}$ and $V_{i+1,x} = V_{i+1,y}$.

Validity: a process p in view V_i, which is not suspected by any process in V_i, is a member of the view V_{i+1}.

The action of taking a decision reflects in the *view synchronous communication* problem the delivery of the agreed set of messages and the installation of the new view.

8.4.3 Failure Detector $\diamond\mathcal{S}$

Properties

The *view change* protocol is based on the consensus protocol of Chandra and Toueg described in [Chandra 95] based on a failure detector called $\diamond\mathcal{S}$. This failure detector ensures strong completeness and eventually weak accuracy.

The strong completeness property is defined as follows:

"Eventually every process that crashes,
is permanently suspected by all correct processes",

whereas the eventually weak accuracy is defined as follows:

"There is a time after which some correct process
is never suspected by any correct process".

[Guerraoui 96] shows that there exists another eventually weak accuracy property called eventually weak , -accuracy, defined as follows:

"There is a time after which some correct process is never suspected by any process in , ".

The second accuracy property can be seen as weaker than the initial one, as only a majority is never suspected after some time. In [Guerraoui 96], the failure detector $\diamond\mathcal{S}(,\ )$ is defined by the strong completeness and the weak , -accuracy properties. The authors show that the two failure detectors $\diamond\mathcal{S}$ and $\diamond\mathcal{S}(,\ )$ are equivalent, i.e. both failure detectors are sufficient to solve consensus.

Neither the $\diamond\mathcal{S}$ nor the $\diamond\mathcal{S}(,\ )$ failure detector can be built in an asynchronous system (if it were possible, one could solve consensus, which is in contradiction with [Fischer 85]). However, the next paragraph shows that the Phoenix routing layer will implement the failure detector $\diamond\mathcal{S}(,\ )$ with high probability.

Ensuring $\diamond\mathcal{S}(,\ )$ with High Probability

Let , be any majority of the current view. While the system is unstable the properties of $\mathcal{S}$ or $\mathcal{S}(,\ )$ are not ensured. This is however not in contradiction with $\diamond\mathcal{S}(,\ )$. As soon as the system becomes stable for long enough with a majority , of connected processes, and the assumption is that this eventually happens, the completeness and weak , -accuracy properties will be satisfied.

Consider first the completeness property. Remember that the routing layer is periodically exchanging routing tables with other processes, especially with those processes members of the same groups. If one of the processes fails or gets partitioned, this implies sooner or later a change in the reachability sets of the routing layer (safety guarantee of the routing layer). Thus, the routing layer satisfies the *completeness* property, stated above, by delivering reachability sets, where the crashed or partitioned processes are eventually unreachable.

Consider then the weak , -accuracy property. If the system is stable and there is a majority , of connected processes, then it can be assumed that the reachability sets, delivered by the routing layer, are such that there is a correct process that is not suspected by any process in , .

8.4.4 Consensus Protocol of Chandra and Toueg

A description of only the most important elements in the consensus protocol of Chandra and Toueg is given here. A detailed description and proofs can be found in [Chandra 95]. The algorithm itself can be found in Appendix A.2. The consensus protocol of Chandra and Toueg is based on the failure detector $\diamond\mathcal{S}$, but [Guerraoui 96] shows its correctness and liveness with failure detector $\diamond\mathcal{S}(,\ )$, where , is a majority set.

The consensus protocol is based on the *rotating coordinator* paradigm [Reischuck 82, Chandra 90]. Computation proceeds in asynchronous rounds, where in each round another process is the co-

ordinator. The coordinator of round r is the process with rank $c = (r \ mod \ n) + 1$, where n is the number of processes participating to the protocol. Each coordinator tries to come to an agreement on a value and to decide. If agreement is not possible, the processes move to the next round. A round can be roughly outlined by the following four phases:

Phase 1.) the participants send their estimates to the coordinator[5];

Phase 2.) the coordinator gathers a majority of estimates and proposes a value out of the estimates based on the criteria described below;

Phase 3.) either (1) a participant receives the proposition of the coordinator, accepts it as its new estimate for future rounds, and sends an acknowledgement to the coordinator, or (2) the participant suspects the coordinator[6] and sends a negative acknowledgement to the coordinator. In both cases, the participant moves to the next round.

Phase 4.) the coordinator gathers a majority of responses; if there is a majority of positive acknowledgements, the coordinator decides, and sends the decision to the participants. Otherwise the coordinator moves to the next round.

Figure 8.3 gives a first overview of one round of the consensus protocol.

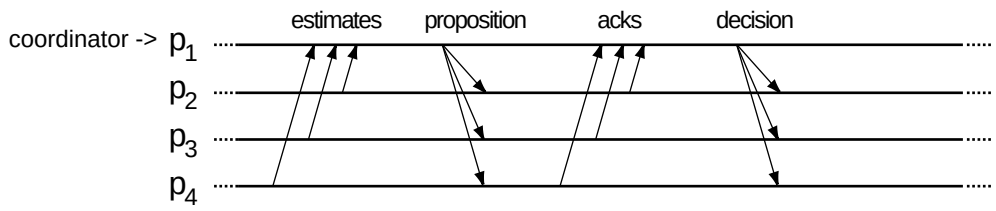

Figure 8.3: Consensus Protocol with No Failure

Choosing the Proposition Value

All messages are time stamped with the round number in which they are issued. The round number r is used to order messages at a process in respect to the current round number; a message with a smaller round number than the current in which the process is, is discarded, a message with a greater round number is buffered until the process is itself in this round. The estimates, sent by the participants, contain additional information:

- the round number t_s in which the latest proposition has been accepted and acknowledged;

[5]The coordinator is not sending its own estimate to itself, but it takes it into account in phase 2.).

[6]A process p is suspected when the routing layer delivers a reachability set not including p.

- the estimate itself.

The proposition and the decision message contain the proposition value, or the decision value.

The stamp t_s in the estimates is used in phase 2.) by the coordinator to choose one estimate out of the received estimates. The estimate is chosen by the coordinator in the following way:

a.) search among the estimates the one with the largest t_s;

b.) if there is more than one estimate satisfying condition a.) select one of them arbitrarily.

A participant, receiving a proposition, adopts it as its new estimate for future rounds and updates its t_s. Figure 8.4 shows a situation where the round number r and the stamp t_s are used.

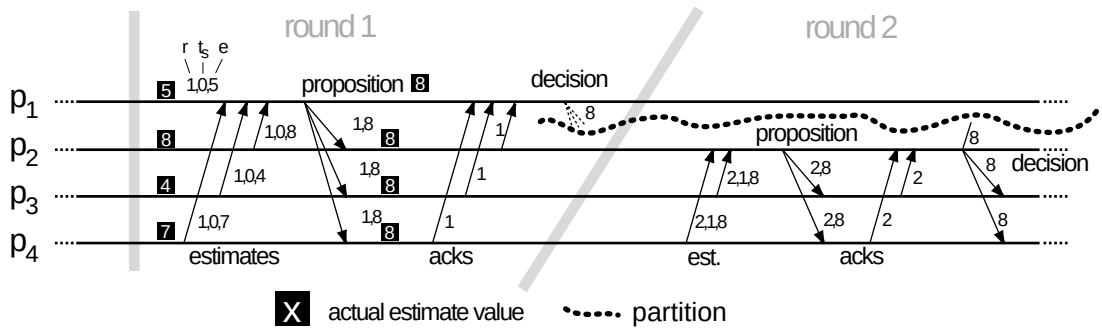

Figure 8.4: Partitioned Coordinator

The coordinator p_1 receives the estimates 8, 4 and 7; the coordinator's estimate is 5. All these estimates are sent in round number 1 and the round of the last update of this estimate t_s is 0. The coordinator proposes 8 as the decision value; each participant receives this proposition, adopts it as its new estimate and updates t_s to 1. All participants send back a positive acknowledgement which the coordinator successfully receives and decides 8. Assume that at this point, the coordinator gets partitioned, before it can send the decision. The participants move to round 2, and send their estimate to the new coordinator p_2. All estimates are equal to 8, with $t_s = 1$. Process p_2 receives a majority of estimates, and proposes again 8.

Note that, if a majority of processes accepts a proposition from a coordinator, this proposition is locked, i.e. in future rounds it does not change anymore and the protocol will eventually decide on this value. Thus, the protocol goes through three *epochs*, each of which may be composed of several asynchronous rounds. In the first *epoch*, any estimate can become the proposition/decision value. In the second *epoch*, the decision is locked, and in the third *epoch*, the processes decide on the locked value.

Decision

At any moment in the protocol, a participant can receive a decision from a coordinator. If a decision is received, the participant decides and terminates. In the protocol of Chandra and Toueg, the decision value is sent to the participants using a particular primitive, called *reliable broadcast*. This primitive ensures that eventually every participant receives the decision or fails. The implementation of this *reliable broadcast* consists in sending the decision to all participants, and every participant receiving the decision for the first time sends it to all other participants and then decides. We will see that the view change protocol does not need this *reliable broadcast*, as the system continues execution after having taken a decision (see Section 8.4.5).

8.4.5 View Change Protocol

In contrast to the consensus protocol, where all processes start the protocol and terminate with a decision, the Phoenix system is already running when the need for the execution of the protocol arises. The event that triggers the launch of the protocol is either a changed reachability set received from the routing layer, or, as shown in Section 8.5, the joining of new processes or the leaving of members. In any case, it is necessary to inform all the processes that a *view change* is necessary. For this purpose the actual *view change* protocol is prefixed by a phase where a process sends a multicast to all members of the current view, inviting them to launch the protocol. Multiple receptions of this launch request are discarded.

The *view change* protocol is based on the consensus protocol described in the previous section using the same failure detector $\diamond\mathcal{S}$ or $\diamond\mathcal{S}(,\ )$. The *view change* protocol goes through the same four phases of the consensus protocol. The principal differences consist in the following points:

Initial estimate: in phase 1.) a participant p does not send a single estimate value to the coordinator, but the set of non-stable messages at p and the current reachability set;

Generation of a proposition: in phase 2.) the coordinator does not wait for a majority of estimates but for an estimate from all processes not suspected by the failure detector; the way the proposition is generated out of the estimates received is described later;

Decision: a process receiving a decision, delivers those messages contained in the decision, which are not yet delivered to the overlying layer, and then installs the new view.

The next paragraphs explain the differences in detail. Appendix A.3 gives the algorithm.

Initial Estimate

Upon launch of the *view change* protocol, each process defines its initial estimate. This estimate is composed of the set of messages not yet stable at this process and the set of processes that

should define the new view. The initial value of this new view estimate is the current set of reachable processes when the protocol is launched (remember that the routing layer delivers this information). This set can be different for each process, as can the set of unstable messages.

Generation of a Proposition

During the execution of the *view change* protocol, a coordinator receives estimates from the participants of the current round. The generation of the proposition is done by the coordinator.

A coordinator receives estimates, containing sets of messages and sets of processes. It is up to the coordinator to generate a proposition based on the estimates received. In contrast to the consensus protocol, the coordinator in the *view change* protocol, waits not only for a majority of estimates, but for an estimate from every process not suspected. Figure 8.5 and Figure 8.6 explain the reason for this modification.

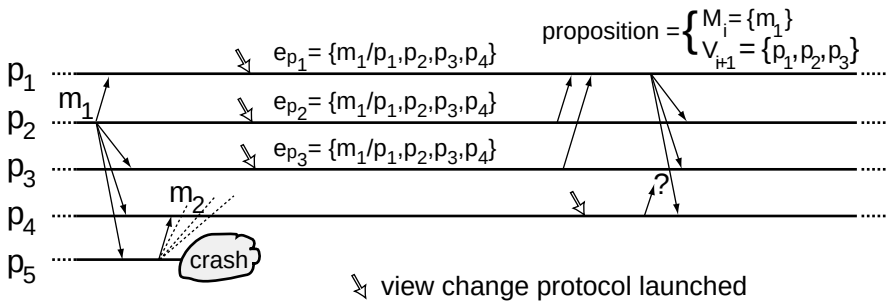

Figure 8.5: Waiting Only for a Majority

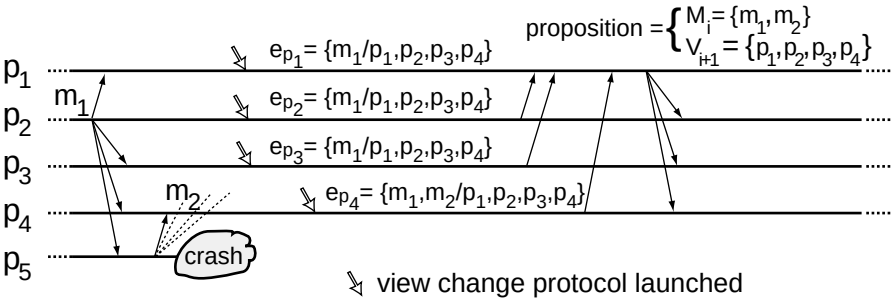

Figure 8.6: Waiting for All Processes not Suspected

The coordinator of Figure 8.5 only waits for a majority of estimates. Thus, it does not wait for the estimate of process p_4. Thus it cannot include p_4 in the new view, because p_4 could have (and actually has in the figure) delivered a message m_2 not delivered by the other processes. Thus, the proposition $(m_1/p_1, p_2, p_3, p_4)$ would violate the *view synchronous communication* property, whereas the proposition $(m_1/p_1, p_2, p_3)$ would satisfy the *view synchronous communication* property. Waiting for a majority would lead to a new view with $\lceil \frac{|V_i|}{2} \rceil$ processes, independent of whether the others are suspected or not. In order to avoid this problem the coordinator waits for an estimate from all processes not suspected (see Figure 8.6). This slight

modification does not change the liveness or safety property of the protocol. Liveness can be shown in the following way: by the completeness property of the failure detector, the estimate of a participant will arrive or the coordinator eventually suspects the participant.

Although the coordinator waits for the estimates from all processes not suspected, there is still more than one possibility to generate a proposition satisfying the *view synchronous communication* property. A valid but not suggested proposition is shown in Figure 8.5, where the coordinator could propose the sets $(m_1/p_1, p_2, p_3)$; this proposition satisfies the *view synchronous communication* property, but process p_4 has to be excluded. Thus, the choice of the proposition has to be done in a way to keep the system as stable as possible.

Consider first the set of messages to propose, as this influences the choice of the new view (as seen above). Out of the spectrum of valid sets of messages, the following two cases are interesting:

- **Intersection of all sets of unstable messages received in the estimates**
  This proposition has the advantage that every process has at least seen these messages and no further messages. Unfortunately, this choice can lead to a quick degradation of the system, as any process, having delivered more than this intersection set, cannot be included in the new view; otherwise the *view synchronous* property is not satisfied.

- **Union of all message sets received in the estimates**
  This proposition has the advantage that all processes which have sent an estimate will be considered for being in the next view.

The second solution implies that messages not delivered by some processes have to be forwarded to them. As the coordinator has received all unstable messages it can include them in the proposition which is sent to all the participants; thus the *union of all unstable messages* has been adopted in the actual implementation.

Until now, only the message part of the proposition has been considered. The next paragraph discusses the proposition for the new view.

New View Proposition

The coordinator proposes as the new view the set of processes from which it has received an estimate, i.e. all processes not suspected by the coordinator. The new view proposed by the coordinator is not necessarily equal to the reachability set of processes in the proposed view. It is possible that a process q suspects process p due to a link failure between p and q. If the new view proposed by the coordinator contains p and q, q will suspect p again and launch another *view change* protocol. This is where the routing layer becomes important. The routing guarantees eventually transitivity between processes p and q. In this typical situation the coordinator

can and will play the role of the router, as it can communicate with p and q (it received the estimates). Further *view changes* are avoided.

Decision

In a similar way to the consensus protocol, a process receiving a decision during the *view change* protocol decides. The action of deciding in the *view change* protocol consists of delivering the set of messages before installing the new view. As it is possible that a process receives a decision but it is not a member of the new view a process has to check whether it is a member of the new view.

The consensus protocol of Chandra and Toueg needs a *reliable broadcast* primitive (described in Section 8.4.4) to send the decision to all participants. This primitive ensures that every process eventually receives the decision or fails. The implementation of such a *reliable broadcast* generates a lot of traffic (a system with n processes generates $\mathcal{O}(n^2)$ messages). The actual implementation benefits from the fact that the system continues execution after having decided. By considering the decision message defining view V_{i+1} as the first *view synchronous* multicast in the new view V_{i+1}, then, by definition of the *view synchronous* property, every process member of view V_{i+2} eventually delivers this *decision* message defining view V_{i+1}, or fails. As the *decision* message is multicast to the members of V_{i+1}, it has to be sent separately to processes excluded from this view V_{i+1} using the *reliable send* primitive, to inform excluded processes of their exclusion.

Cleanup of Unstable Messages

When a process p decides on the set of messages M_i and the view V_{i+1}, all messages of view V_i are stable at the members of view V_{i+1}. Thus, process p can discard all messages multicast in view V_i. As the multicast is implemented using *group_reliable_sends*, it is possible that retransmission to failed or partitioned processes is still ongoing. They can be discarded using the *cancel_sends_to_dest* primitive of the reliable communication layer (see Section 7.6).

8.4.6 View Synchronous Multicast during View Change

Message Sending

View synchronous multicasts issued by overlying layers during the execution of the *view change* protocol are buffered in the view synchronous communication layer and delayed until the next view is installed.

Message Delivery

A view synchronous multicast m of view V_i, received after the launch of the *view change* protocol,

is discarded[7]; the receiver has already sent its estimate, not including this message, to the coordinator and is not allowed to change its mind. If the message is part of the view, it will be included in the proposition of the coordinator, and can then be safely delivered.

8.5 Implementation of Dynamic Group Composition Primitives

The next two subsections describe the implementation of the *join* and *leave* primitives. These primitives allow a group to grow and shrink dynamically, as new processes join and leave the group.

8.5.1 Join

In order for a process p_{new} to join a group, it has to issue a *join-request* to at least one member p_i of the group. The reception of the *join-request* at p_i will trigger the *view change* protocol at this process and as a consequence a *launch-request* is multicast to the group by p_i. In contrast to a usual *view change* protocol, p_i includes in its estimate an additional piece of information that there is a new process p_{new} to include in the new view. The proposition from the coordinator will include all the joining processes, and upon installation of the new view, the new processes are members of the group. Remember that the decision is the first multicast in the new view and thus the joining processes will eventually receive this message or crash.

Similarly to a regular multicast, a *join-request* received during the view change mode, is delayed until the currently executing protocol has terminated with the installation of a new view.

In order to complete the description the occurence of failures during joins has to be considered. The case of the failure of the joining process is handled as the failure of a member. If the members install a view containing the failed, joining process, they will suspect it after having installed the new view. Yet another view change protocol will be launched and another new view will be installed, excluding the failed process.

If the process p_i, which is responsible for launching the *view change* protocol, fails before it has sent its estimate to the coordinator the *join-request* is lost. This can be handled by the joining process p_{new} using a timeout.

[7] This is possible as not all processes launch the *view change* protocol at the same time.

8.5.2 Leave

A process, member of a group, has three ways to leave a group:

1.) it simply stops its execution, will be suspected and excluded from a future view, or

2.) it *multicasts* a *leave-request* to the group and stops its execution, or

3.) it *multicasts* a *leave-request* to the group and waits for the new view change, from which it is excluded.

Each of the three ways to leave a group is valid, and they all have their drawbacks and advantages. In order to understand these three leaving solutions, it is important to bear in mind that the application process is actually separated from the daemon (see Figure 4.3 of Chapter 4), i.e. the leave or failure of an application does not mean the failure or termination of the daemon.

With solution 1.) a process leaves the group in the same way, as if it had failed. For the leaving process, this is the fastest way to leave a group, but on the group side, things are much slower. The daemon responsible for the application has to detect the failure or the leave of the application, and has to launch the *view change* protocol. Solution 2.) is more resilient as the member is informing the daemon that it is leaving, and the daemon can immediately launch the *view change* protocol. Solution 3.) is important if a process has to leave a group having delivered all the messages up to and including those of the view where *leave-request* was multicast.

For all three solutions, and similarly to the handling of *join-requests*, the same additional information is included in the estimate, in order to build a proposition not including the leaving member. As the decision is again the first multicast of the new view, the coordinator sends a separate message to the excluded member.

8.6 View Synchronous Communication Layer Programming Paradigm

For an overlying layer, the *view synchronous communication* layer provides a *view synchronous* multicast primitive, as well as the primitives for joining and leaving groups. The *view synchronous communication* layers deliver multicasts and views, where multicasts are ordered with respect to views. Processes delivering a new view have the guarantee that all other processes in the intersection of the previous and the new view have delivered the same set of messages in the previous view.

We have seen in the introduction to this chapter that the VSC layer is either in normal mode or is executing the view change protocol. The execution of the protocol can be due to failures,

joins or leaves of processes. The overlying layers might be interested to learn when a *view change* protocol is launched. For this purpose an overlying layer has the possibility to install an *intermediate view* callback (described in Chapter 4) which is called each time the reachability set of processes changes. The *first* call to this callback coincides with the launch of the *view change* protocol. As it is possible that for some time no agreement can be found, all the successive reachability sets before the definition of the new view are delivered, and the overlying layer can act accordingly. Figure 8.7 shows a typical example and the different modes of the VSC layer.

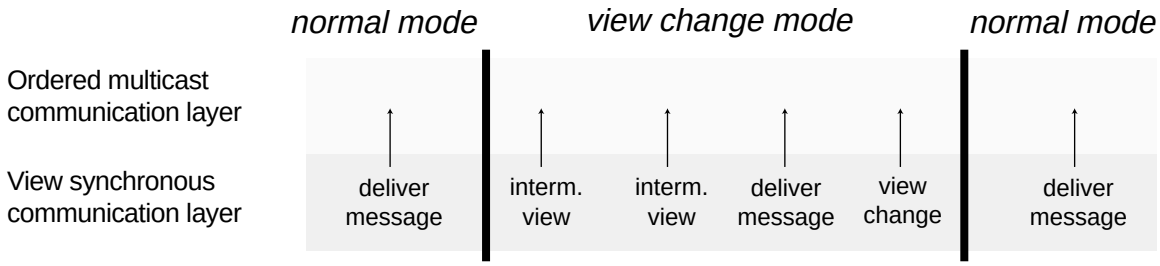

Figure 8.7: Flow of Information through VSC Layer

The contents of the *intermediate view* callback are the set of reachable processes delivered by the routing layer. Note that the *intermediate view* callbacks allow an application to make progress while the *view change* protocol executes. This is discussed further in Section 8.7.2.

8.7 Discussion

8.7.1 Unique View Sequence vs. Concurrent Views

The *view synchronous communication* paradigm leaves open many implementation issues. Consider Figure 8.8 and Figure 8.9, which both satisfy the definition of *view synchronous communication*.

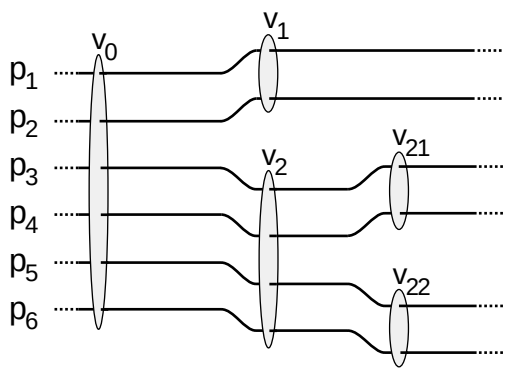

Figure 8.8: Concurrent Views

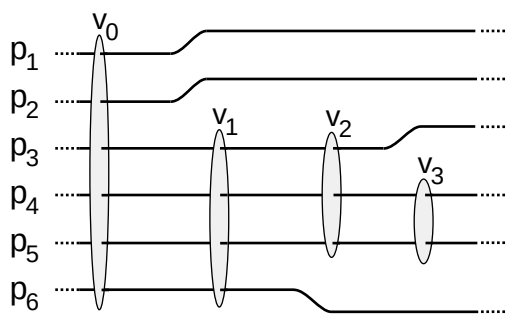

Figure 8.9: Unique View Sequence

Figure 8.8 shows concurrent views. In such a system it is possible to identify the view representing the state of the group, by considering the views containing a majority of processes out of the initial set of processes V_0 (views V_0, V_2 in Figure 8.8 are view containing a majority of processes). As soon as there is no view containing a majority, the state of the group is lost (views V_1, V_{21} and V_{22} in Figure 8.8). At a later point a view V_3 could be found again containing a majority of processes. The problem is that the minority views V_1, V_{21} and V_{22} could have divergent state information and merging would be necessary. This merge may not be simple and may even be impossible. Further, if more than a majority of the processes of view V_0 fail, the majority is lost forever.

The definition of the majority in the concurrent views approach can be compared to the static voting technique for handling replicated data described in [Gifford 79]. In this technique the way by which a majority is designated is static, i.e. there is an initial set of processes (view V_0) and the majority is always dependent on this view.

In Figure 8.9 the majority is defined with respect to the most recent view and not with respect to the initial view. In such a context the state can be maintained even if a majority of processes fails (V_3 contains only 2 members), under the condition that at each view change less than a majority of the processes in the current view fails. This leads to a unique sequence of views and the ability always to define a primary partition composed of a majority of processes. The drawback of the primary partition model is discussed in the next section.

The unique sequence of views approach can be compared to the dynamic voting technique for handling replicated data described in [Davcec 85]. In this technique the majority is designated dynamically out of the most recent *view*.

8.7.2 Primary Partition and the Blocking Problem

Section 8.7.1 showed two different possibilities to implement *view synchronous communication*. The implementation with concurrent views lacks the guaranteed existence of a primary partition. Figure 8.10 is a typical situation where the primary partition is lost, when concurrent views are allowed.

This situation is not uncommon in large scale networks. In such a situation, concurrent views are installed and the primary partition is lost. Platforms, like Horus [van Renesse 92b] or Transis [Amir 92b], have taken the approach of implementing concurrent views, because the more restrictive primary partition implementation, as in Isis, would block. Horus and Transis do not block.

In the case of the Isis view change protocol [Ricciardi 91], which is based on the stable suspicion model, the situation of Figure 8.10 leads to the permanent blocking of the whole system; as in

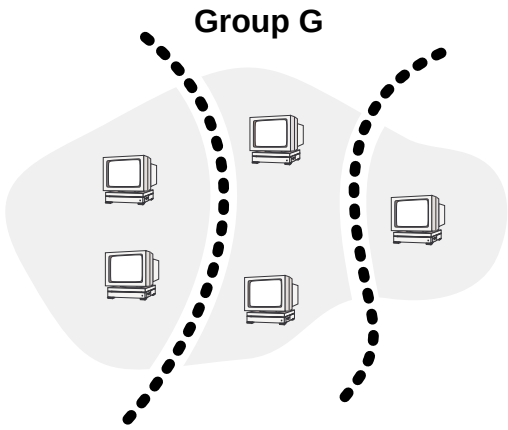

Figure 8.10: Configuration with Only-Minority Partitions

every partition there is only a minority of processes, no primary partition can be defined and because of the stable suspicion model, once a process is suspected it is suspected forever.

Phoenix is intermediate between Horus and Isis, in that it always provides a primary partition and the absence of a permanent blocking in situations like the one depicted in Figure 8.10. The *view change* protocol blocks only as long as a primary partition cannot be defined. As soon as enough link failures are repaired and a primary partition can be defined, the *view change* protocol terminates. As mentioned earlier, only the *view change* protocol blocks during the period where no primary partition can be defined, but not the application which is informed about the situation through the *intermediate view* callback. Thus, nothing prevents the application from defining a new group containing the set of processes delivered by the *intermediate view* callback. In other words it is possible to have concurrent views in Phoenix, but they have to be explicitly handled at the application level.

8.7.3 Process Membership vs. Site Membership

The Phoenix system architecture, presented in Chapter 4, distinguishes between the daemon and the application processes. On each site a daemon is running and applications are connected to the local daemon. In Phoenix the group membership is implemented at the application level, i.e. the daemon is *member* of only those groups which the applications using this daemon have actually joined. This is important for the scalability and the liveness of the system. Figure 8.11 illustrates this situation.

If the membership is managed at the site level, the partition leads to two partitions $S_1 = \{s_0, s_1, s_2\}$ and $S_2 = \{s_3, s_4, s_5, s_6\}$. As S_2 is the majority partition, the following new views for the groups can be defined: for the group g_1 the new view consists of a_3, a_4, a_5, a_6 and for the group g_2 the new view consists of b_3. The new view for group g_1 is not a problem, but for group

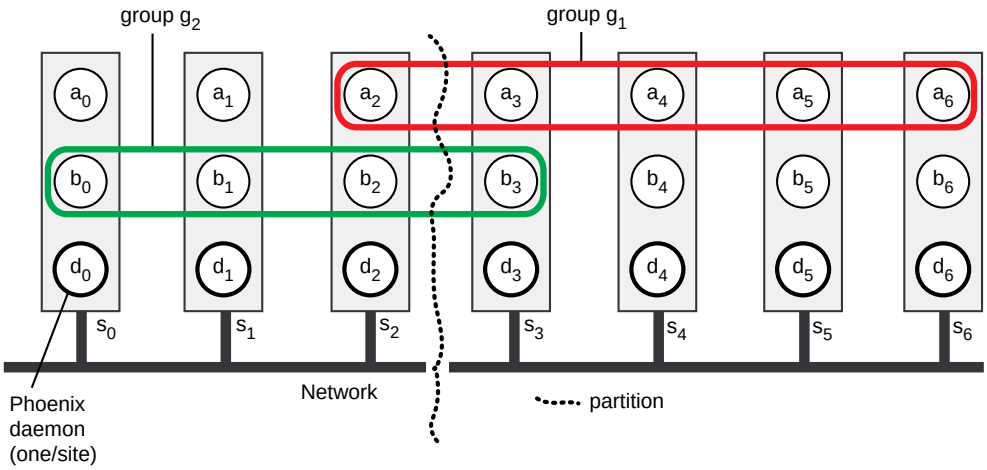

Figure 8.11: Process Membership vs. Site Membership

g_2, the new view consists of a minority of processes of the previous view. If further process b_3 crashes, consistent state information for group g_2 is lost forever even though b_0, b_1 and b_2 are alive and could even define a majority of group g_2.

With the membership managed at the application level, this situation does not occur, i.e. for group g_1 the view will be defined as stated above, however for group g_2 the defined new view consists of processes b_0, b_1, b_2.

This way of managing group membership does not only lead to a better exploitation of fault-tolerance, but also to scalability of the system. In the site membership approach all sites receive a multicast and have to filter the message depending whether a corresponding member is running on this site or not. In the process membership approach only sites which have members of the group receive the multicast. This is not only interesting in the case of regular multicasts, but especially for the execution of the view change protocol, as only a subset of these sites is concerned by the protocol.

Note that here only site failures and partitions between sites are considered, because application process failures are managed by the daemon and partition between two processes on the same site does not occur. The failure of a local process is simulated by its daemon as an explicit leave of the failing process which allows the termination of the *view change* protocol as in the case of a regular leave (see Section 8.5).

8.8 Summary

The *view synchronous communication* layer provides a new implementation of the *view synchronous communication* paradigm, which is particularly suitable for large scale systems, where

link failures are not uncommon, but consistent state among group members is necessary. The implementation guarantees that there is always a primary partition; in case of serious communication problems, the system blocks until it is again possible to define a primary partition. The *view synchronous communication* layer provides all necessary primitives for group communication, including primitives for dynamic group membership. At this layer the multicasts to the group are ordered only by views, but they are not ordered among themselves.

Ordering of application messages is done by the *ordered multicast communication layer* (see Chapter 9), on top of the view synchronous communication layer.

Chapter 9

Ordered Multicast Communication Layer

9.1 Introduction

View synchronous communication provides ordering of messages with respect to views. For many applications this kind of ordering is not sufficient to guarantee consistent state information throughout the group despite failures, joins or leaves of processes. Chapter 2 has already introduced applications that require stronger ordering guarantees than view synchronous multicasts. These stronger communication primitives are provided by the ordered multicast communication layer. The implementation of these stronger primitives are all based on the view synchronous multicast primitive (noted vsc-mcast in the following) provided by the view synchronous communication layer.

Section 9.2 gives the formal definitions of several higher level multicast primitives, and illustrates their use. Section 9.3 gives a description of the implementation of the primitives. A more detailed description of some of them can be found in [Birman 91], [Schiper 93] and [Wilhelm 95].

9.2 Definitions

The definitions are based on the definitions of events and their ordering (see Section 8.2.2), the reception and delivery of messages (see Section 8.2.3), the correctness of process (see Section 8.2.4) and the view synchronous multicast (see Section 8.2.5).

9.2.1 FIFO Multicast

Definition

For two messages m_1 and m_2 that are FIFO-multicast to view V_i, by one single process p in the order m_1 before m_2, using the FIFO multicast primitive fifo-mcast, the following condition holds:

If there is a correct process p which FIFO-multicasts message m_1 before m_2, then all correct processes q deliver message m_1 before m_2.

Formally, this can be written:

$$\exists p \in \mathcal{C}(V_i) : \text{MCAST}_i^p(m_1) \ \wedge \ \text{MCAST}_i^p(m_2) \ \bigwedge \ \text{BEFORE}(mcast_i^p(m_1), mcast_i^p(m_2)) \ \Rightarrow$$
$$\forall q \in \mathcal{C}(V_i) : \text{DLV}_i^q(m_1) \ \wedge \ \text{DLV}_i^q(m_2) \ \bigwedge \ \text{BEFORE}(dlv_i^q(m_1), dlv_i^q(m_2))$$

For the definition of the events *mcast*, *rcv* and *dlv*, the properties MCAST, RCV, DLV and BEFORE, and the definition of correct processes $\mathcal{C}(V)$ refer to Section 8.2.

Note that the regular *view synchronous* multicasts are not ordered among each other and that a process, not a member of the new view, can deliver the messages in an arbitrary order.

9.2.2 Weak Totally Ordered Multicast

Definition

For any two messages m_i and m_j, multicast in view V_i using the *weak total ordered* multicast primitive wto-mcast, the following condition holds:

If there is a correct process p which delivers m_i and m_j in the order m_i before m_j, then every correct process delivers m_i and m_j and in the same order as p.

Formally, the following condition is satisfied:

$$\exists p \in \mathcal{C}(V_i) : \text{DLV}_i^p(m_i) \ \wedge \ \text{DLV}_i^p(m_j) \ \bigwedge \ \text{BEFORE}(dlv_i^p(m_i), dlv_i^p(m_j)) \ \Rightarrow$$
$$\forall q \in \mathcal{C}(V_i) : \text{DLV}_i^q(m_i) \ \wedge \ \text{DLV}_i^q(m_j) \ \bigwedge \ \text{BEFORE}(dlv_i^q(m_i), dlv_i^q(m_j))$$

Note that an incorrect process can deliver the messages in any order as in this definition only correct processes are considered.

Illustration

Figure 9.1 illustrates a typical execution with weak totally ordered multicasts. In this example messages m_1 to m_6 are multicast using the weak total order multicast primitive wto-mcast.

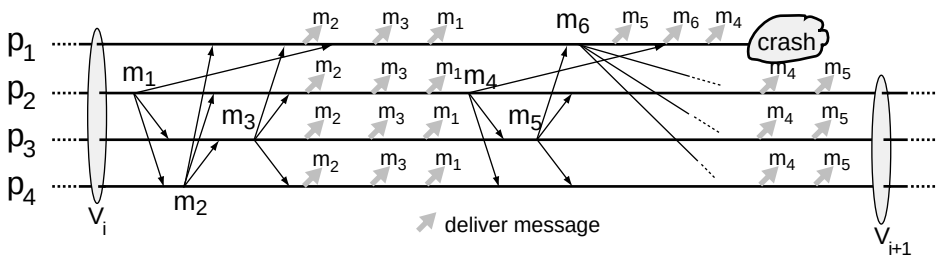

Figure 9.1: Weak Totally Ordered Multicast

Process p_1 fails and is thus allowed to deliver messages m_4 and m_5 in a different order than the other processes and m_6 is delivered only by p_1. This does not violate the weak total order definition, which states that correct processes have to deliver messages in the same order. As p_1 is incorrect, it can deliver them in any order. If this mis-ordering of m_4 and m_5 and the *non-uniform* delivery (see next subsection) of m_6 at p_1 is not desired, the strong totally ordered multicast has to be used.

9.2.3 Uniform Multicast

Definition

For a message m multicast in view V_i, using the *uniform* multicast primitive uni-mcast, the following condition is satisfied:

If any process, whether correct or not, delivers message m,
then every correct process delivers message m.

Formally:

$$\exists p \in V_i : \text{DLV}_i^p(m) \quad \Rightarrow \quad \forall q \in \mathcal{C}(V_i) : \text{DLV}_i^q(m)$$

Illustration

The uniform multicast is a typical *all-or-nothing* primitive. It ensures that if any process of view V_i (whether correct or not) has delivered message m, all correct processes will eventually deliver this message m in the same view.

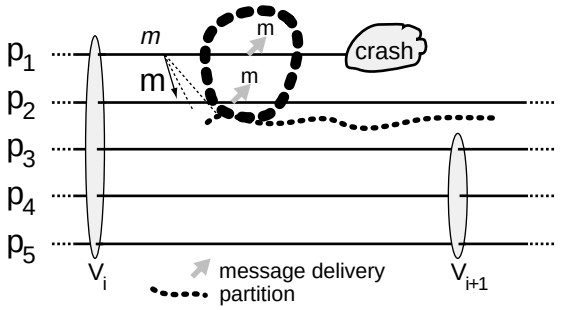

Figure 9.2: Non-Uniform Multicast

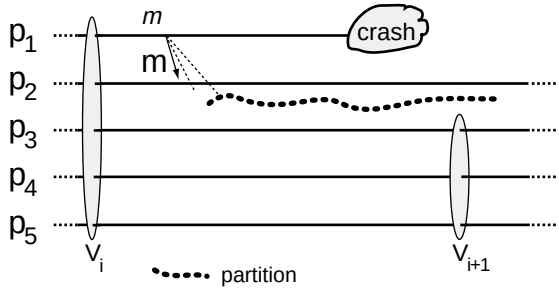

Figure 9.3: Uniform Multicast

Figures 9.2 and 9.3 compare a non-uniform multicast with a uniform multicast. In both examples, process p_1 fails while multicasting m, and only process p_2 receives the message. Process p_2 gets partitioned from the other process and is thus not included in the new view V_{i+1} (p_2 is an incorrect process). In Figure 9.2, the message is delivered by processes p_1 and p_2 (i.e. $\text{RCV}_i^{p_1}(m)$, $\text{RCV}_i^{p_2}(m)$, $\text{DLV}_i^{p_1}(m)$ and $\text{DLV}_i^{p_2}(m)$ hold), but not by p_3, p_4, and p_5 (i.e. uniformity is violated), whereas in Figure 9.3 the message is not delivered (i.e. $\text{RCV}_i^{p_1}(m)$, $\text{RCV}_i^{p_2}(m)$ still hold, but not $\text{DLV}_i^{p_1}(m)$ and $\text{DLV}_i^{p_2}(m)$) by any process (correct or not).

The uniform multicast is the adequate primitive to implement atomic commitment [Bernstein 87, Schiper 93] of transactions. Suppose that p_1 is the coordinator for an atomic commitment. If process p_1 sends the commit message C with the *uniform* multicast primitive uni-mcast, there are two cases: (1) if any process delivers C in view V_i, then every correct process also delivers C in view V_i, or (2) no process will ever deliver C in any view. In case (1), the delivery of the message C leads to *commit* the transaction, whereas in case (2), a new coordinator has to be defined which tries to *commit* the transaction in view V_{i+1}.

9.2.4 Strong Totally Ordered Multicast

Definition

For any two messages m_i and m_j multicast in view V_i using the *strong total ordered* multicast primitive sto-mcast, the following condition is satisfied:

If there is any process of view V_i (whether correct or not) which delivers m_i and m_j in the order m_i before m_j, then every correct process of view V_i delivers m_i and m_j in the order m_i before m_j.

Formally:

$$\exists p \in V_i : \text{DLV}_i^p(m_i) \ \wedge \ \text{DLV}_i^p(m_j) \ \ \bigwedge \ \ \text{BEFORE}(dlv_i^p(m_i), dlv_i^p(m_j)) \ \ \Rightarrow$$
$$\forall q \in \mathcal{C}(V_i) : \text{DLV}_i^q(m_i) \ \wedge \ \text{DLV}_i^q(m_j) \ \ \bigwedge \ \ \text{BEFORE}(dlv_i^q(m_i), dlv_i^q(m_j))$$

In contrast to the weak totally ordered multicast, the strong totally ordered multicast considers all processes and thus does not allow an incorrect process to deliver the message in another order than the delivery order of the correct processes. By comparing the formula of the strong totally ordered multicast with the one for weak totally ordered multicast, the reader can note in the first part of the formula, that the strong total order considers any process p, where the weak total order only considers any correct process p.

Illustration

The strong totally ordered multicast extends the weak one with the uniformity guarantee. Typically in Figure 9.1 messages m_4 and m_5 are delivered in the wrong order. For a weak total order multicast this is not wrong, as the definition considers only correct processes and process p_1 is not correct and can thus deliver the message in any order. Using the strong totally ordered multicast, the messages would not have been delivered in another order, even at incorrect processes. Furthermore m_6 would not have been delivered at all, as it has not been received by all processes. An execution with m_1 to m_6, multicast using the strong totally ordered multicast primitive sto-mcast, is depicted in Figure 9.4.

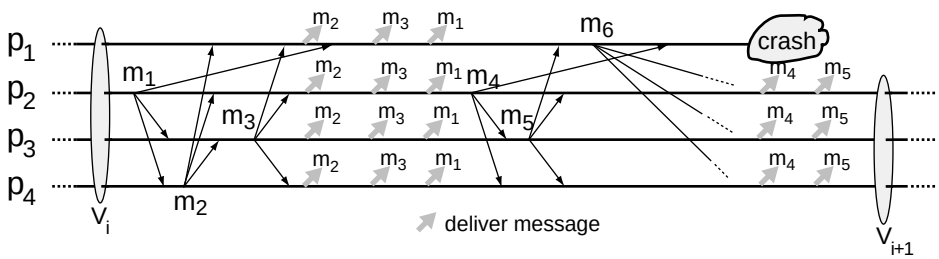

Figure 9.4: Strong Totally Ordered Multicast

Note that the definition would allow p_1 to deliver only m_5 without delivering m_4, but the implementation is even stronger not allowing this delivery order.

9.2.5 Hybrid Total Order

The two primitives that impose an order on the delivery of messages (within a view) are the weak and the strong totally ordered multicast. However, since weak total order and strong total order are basically two different primitives, the order of two messages m_i and m_j, where m_i is multicast using a wto-mcast and m_j using a sto-mcast, is not defined. Hybrid total order incorporates the

strong and the weak totally ordered multicasts and defines the ordering of messages issued with different total order primitives. There are several possibilities to incorporate these two total orders [Wilhelm 95], but only the definition that has been implemented in the Phoenix system is considered here.

Definition

For two multicasts m_s and m_w, where m_s is issued using a weak totally ordered multicast primitive and m_w is issued using a weak totally ordered multicast primitive in view V_i, the following condition is satisfied:

For any two processes p and q (whether correct or not),
where process p has delivered messages m_s and m_w in the order m_s before m_w,
process q has also delivered m_s and m_w in the order m_s before m_w.

Formally:

$$\forall p, q \in V_i : \text{DLV}_i^p(m_s) \ \wedge \ \text{DLV}_i^p(m_w) \ \wedge \ \text{DLV}_i^q(m_s) \ \wedge \ \text{DLV}_i^q(m_w) \ \bigwedge$$
$$\text{BEFORE}(dlv_i^p(m_s), dlv_i^p(m_w)) \ \Rightarrow$$
$$\text{BEFORE}(dlv_i^q(m_s), dlv_i^q(m_w))$$

The incorporation of the two total order multicasts allows a meaningful use of both primitives, the weak and the strong totally ordered multicast primitives, in applications. In the absence of this hybrid order it would only be possible to use these primitives separately.

9.2.6 Global Order Multicast

A global multicast partitions the entire set of multicasts delivered into two disjoint subsets: the multicasts delivered *before* and those *after* the global order multicast.

Definition

For two messages m and m_G, where m is multicast with any multicast primitive and m_G is multicast using the global multicast message m_G sent using the *global* multicast primitive glomcast, sent by the same process in the order m before m_G, in view V_i, the following condition is satisfied:

For every two correct processes q and r, which both deliver
the global multicast m_G, the sets of messages delivered before
the delivery of m_G are identical and include message m.

Formally:

$$\begin{gathered}\exists p \in \mathcal{C}(V_i) : \text{MCAST}_i^p(m) \wedge \text{MCAST}_i^p(m_G) \ \wedge \ \text{BEFORE}(mcast_i^p(m), mcast_i^p(m_G)) \ \bigwedge \\ \forall q, r \in \mathcal{C}(V_i) : \text{DLV}_i^q(m) \wedge \text{DLV}_i^r(m) \ \wedge \ \text{DLV}_i^q(m_G) \wedge \text{DLV}_i^r(m_G) \ \bigwedge \\ M_G^q = \{m_i \mid \text{BEFORE}(dlv_i^q(m_i), dlv_i^q(m_G))\} \wedge \\ M_G^r = \{m_i \mid \text{BEFORE}(dlv_i^r(m_i), dlv_i^r(m_G))\} \ \Rightarrow \\ M_G^q = M_G^r \ \bigwedge \ m \in M_G^q \ \wedge \ m \in M_G^r\end{gathered}$$

The global order multicast can be seen as a simulation of a view change, where the delivery of the new view is replaced by the delivery of the *global order* multicast; every process that has delivered m_G has delivered the same set of messages before delivering m_G.

9.3 Implementation

Only the main ideas for the implementation of the above primitives are given here. Further information and proofs can be found in the references given below.

All these implementations rely on the *view synchronous* property ensured by the *view synchronous communication* layer. Common to all implementations is the key idea to delay the delivery of a received message until some condition is satisfied. The next section gives a general overview of this principle.

9.3.1 Message Buffering and Delivery Delaying

Considering the Phoenix architecture (see Figure 9.1), the ordered multicast communication layer receives messages from the view synchronous communication layer and delivers them to the application layer. Figure 9.5 shows a typical example where a message m_1 is received by the ordered multicast communication layer and then delivered to the application layer[1]. Message m_2 is received, but its delivery is delayed until the message $delv(m_2)$ is received.

In the next subsections describing the implementation of different ordered multicast primitives, the delivery of received messages is often dependent on the reception of other messages, view changes and/or local state information in the layer. Received messages, whose delivery has to be delayed are buffered in the layer until their delivery condition is satisfied. In the regular case the delivery of message m is dependent on the reception of another message associated to m or to some local counter, where in the case of failure the *associated* message is the reception of a new view from the view synchronous communication layer.

[1] The application programming interface is not considered as a complete layer, since it only provides an interface to the procedures provided by the underlying layers.

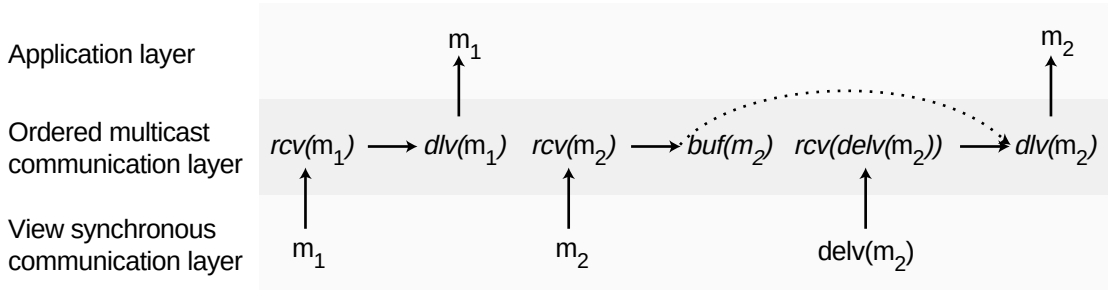

Figure 9.5: Message Reception and Delivery

In the following the term *received* means received from the view synchronous communication layer by the ordered multicast communication layer and the term *delivered* means the delivery from the ordered multicast communication layer to the application layer.

9.3.2 FIFO Multicast

The FIFO multicast primitive fifo-mcast is implemented adding a sequence number, local to each sender, to each multicast sent with this primitive. On the receiver side the ordered multicast communication layer recognizes these FIFO multicasts and orders them according to the sequencer numbers. A message, received out of the FIFO order, is buffered and delayed.

Note that for two messages m_1 and m_2 multicast using the FIFO multicast primitive fifo-mcast, it is possible that a process p_j receives the message m_2, but does not receive the preceding message m_1. Suppose that p_j fails after having received m_2, thus p_j is not a correct process. In this situation p_j will never deliver m_2, although because p_j is incorrect, the definition could allow p_j to deliver m_2, but not m_1.

9.3.3 Uniform Multicast

A process receiving a *uniform* multicast m does not deliver it immediately, but buffers it until it receives information (in form of a message $delv(m)$) that all the processes have received m. Then the message m is delivered.

The implementation is similar to the stabilization protocol presented in Section 8.3.2. Consider a message m multicast by p to a group, using the uni-mcast primitive. Process p multicasts the message m using a *view synchronous* multicast, including additional information for the ordered multicast communication layer of the destinations, not to deliver the message immediately. The destination processes which receives the message, buffer it and send back an acknowledgement to p[2]. As the current view V_i is known, process p can determine when it has received an

[2]This acknowledgement does not have to be explicitly sent, as the view synchronous multicast uses the reliable

acknowledgement from each destination. p then multicasts, again using a view synchronous multicast, a second message $delv(m)$ indicating that the buffered message m can be delivered.

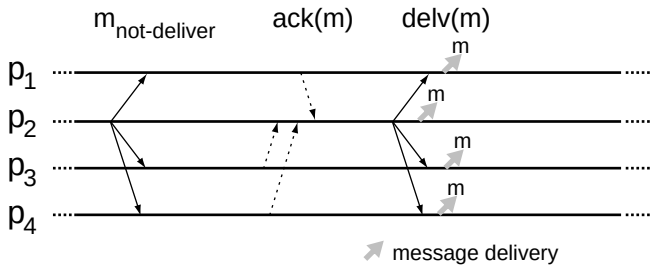

Figure 9.6: Implementation of the Uniform Multicast

This algorithm is correct as long as no failures occur. In case of a failure, the ordered communication layer sooner or later receives a new view. Before this new view is delivered, a termination protocol is executed. The termination protocol does not require any additional communication and consists of the following action: upon reception of the new view from the view synchronous communication layer, all correct processes of the new view deliver all buffered uniform multicasts before delivering the new view to the application[3]. A process that has not failed, but is excluded from the new view discards all buffered uniform multicasts. For a detailed description and the proof refer to [Schiper 93].

9.3.4 Weak Totally Ordered Multicast

The implementation of the weak totally ordered multicast corresponds to the Isis ABCAST implementation described in [Birman 91]. Suppose a process p issues a message m using the wto-mcast primitive. The message is multicast with a view synchronous multicast vsc-mcast. At the destination this multicast is recognized as a weak totally ordered multicast and buffered until the reception of a sequence number associated with the multicast. A sequencer process, for example the process with the smallest rank number, is responsible for generating a sequence number for each weak totally ordered multicast it receives. This sequence number is multicast using the view synchronous multicast primitive vsc-mcast. Thus the condition to deliver a message multicast with the weak totally ordered multicast primitive wto-mcast is the reception of the message and the reception of the sequence number from the sequencer process.

Upon reception of the sequence number for message m, a process can deliver the message m, provided that it has already delivered all weak totally ordered messages preceding m[4].

In case of a failure, an arbitrary but deterministic order on the remaining, undelivered messages

communication layer for multicasting messages; the reliable communication layer automatically generates an acknowledgement for each message, which can be exploited in the ordered multicast communication layer.

[3]This is correct, as every process in the new view has delivered or received the same set of messages and thus

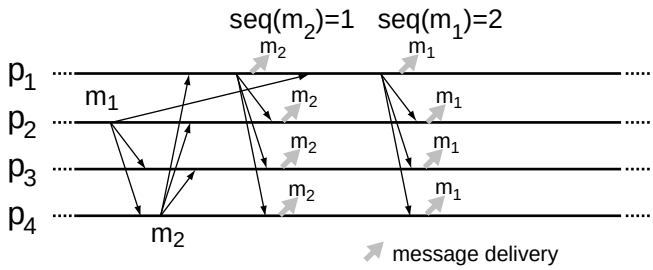

Figure 9.7: Implementation of the Weak Total Order

is constructed and the messages are delivered in this order. This a priori order is based solely on data contained in the messages without any further message exchanges. It is based first on the rank number of the sender, and then on the sequence number included in the message identifier. Because of the view synchronous property, all correct processes deliver the same set of messages and the set of messages to order is the same at each process. For the detailed description of the algorithm and its proof see [Birman 91].

9.3.5 Strong Totally Ordered Multicast

The key idea of the implementation is to add control information to strong totally ordered multicasts; this is used to define the total order. In contrast to the weak totally ordered multicast, this order is independent of the order in which the strong totally ordered multicasts are received. A coordinator *observes* the order in which the messages have to be delivered.

Each process keeps a round counter which is incremented each time it issues a strong totally ordered multicast, which is tagged with this round number. Upon reception of such a strong totally ordered multicast, which is issued using the view synchronous multicast primitive vsc-mcast, the receiver recognizes the message as a strong totally ordered message, buffers it and issues an acknowledgement to the coordinator using the reliable send primitive of the reliable communication layer. This acknowledgement contains the information whether the receiver has previously issued a strong totally ordered multicast into the same round as the one of the received message.

On the other side, the coordinator waits for the acknowledgements of all processes before multicasting the *observed* order to the processes using the view synchronous multicast primitive. A process orders the received messages in the following way: (1) two messages of different rounds are ordered by the round number in which they were issued; (2) two messages issued in the same

also all uniform multicasts.

[4] As the multicast of the view synchronous communication layer is not ordered, it is possible that the sequencing information arrives before the actual message; in the absence of failures a process waits for both messages before being able to deliver a message.

round are ordered using the rank number of the sender of the message. At the processes, the condition for delivering a strong totally ordered multicast depends on the reception of the message and the reception of the delivery information from the coordinator. Figure 9.8 illustrates the algorithm; the failure cases are discussed below.

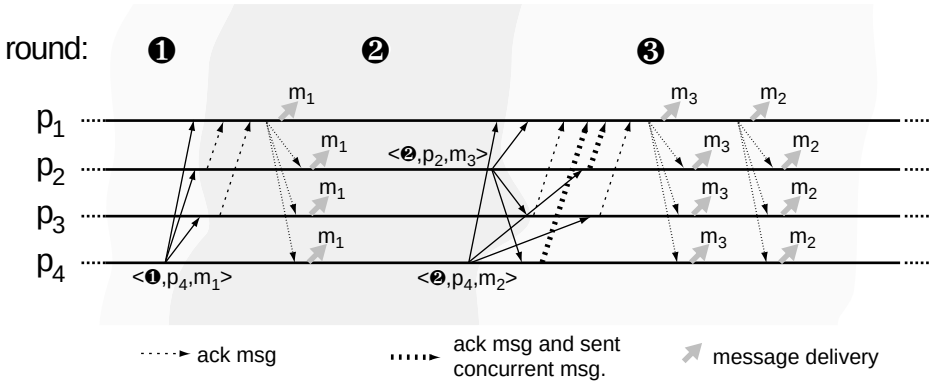

Figure 9.8: Implementation of the Strong Total Order

As the coordinator waits for the acknowledgements of all processes, before issuing the delivery messages[5], the uniform property of the message delivery is guaranteed (see Section 9.2.3 for the definition of the uniform multicast). For each round, the coordinator receives from each process at least one acknowledgement per round[6], indicating whether the process has issued a concurrent multicast in that round or not. As a process can only send at most one strong totally ordered multicast or an acknowledgement for each message issued to a given round, the coordinator receives from each process either a message or an acknowledgement. Thus the order is defined as follows: a message from round n is delivered before a message from round $n+1$, and for messages issued in the same round n, they are ordered using the rank of the sender.

In the case of a failure, the uniformity is guaranteed in the same way as in the uniform multicast, and the delivery order on messages which are not yet ordered can be determined locally by each process, as the set of messages to deliver is known and guaranteed by the underlying view synchronous communication layer. For a detailed description and the proof of the algorithm, refer to [Wilhelm 95].

9.3.6 Implementation of Hybrid Order

Incorporation of the weak and the strong totally ordered multicast is implemented in the following way. Assume that the sequencer of the weak and the coordinator of the strong totally ordered multicasts are the same process. All totally ordered multicasts (strong and weak) are issued into rounds, i.e. weak totally ordered multicasts are tagged by the round counter for strong

[5] These delivery messages are based on the observed order and not on an order assigned by the coordinator.

[6] A message is considered to be an acknowledgement for itself.

totally ordered multicasts described in Section 9.3.5, but this counter is only incremented when a strong totally ordered multicast is issued. Thus, for a given round there can be more than one weak totally ordered multicast, but only one strong totally ordered multicast per process (property of the implementation of the *strong totally ordered* multicast). Thus, the coordinator orders all the weak totally ordered multicasts of one single round in the order the coordinator receives them and before the strong totally ordered multicasts of the same round. This order is applied to the weak and strong totally ordered multicasts of all rounds.

Each process includes the number of weak totally ordered messages it has issued in a given round in the acknowledgement for the strong totally ordered message. Therefore, the coordinator knows exactly how many weak totally ordered messages have been issued in a round, and it can order the messages appropriately.

In the case of failure, the view synchronous communication layer guarantees that every correct process has delivered the same set of messages. This set of messages includes in particular the weak and strong totally ordered message as well as the delivery order messages from the coordinator. For messages to whom no delivery order message is received, all remaining total order messages are ordered in the ways described above, i.e. first by considering round numbers of all messages, then weak totally ordered multicasts before strong totally ordered multicasts and weak totally ordered multicast of the same round by using multicast sequence number and for strong totally ordered multicasts (one per round) the rank number of the sender. A detailed description with proofs is given in [Wilhelm 95].

9.3.7 Implementation of Global Order Multicast

The global order multicast of a message m_G is implemented by launching the view change protocol: the outcome is usually a new view where the composition has not changed. The delivery of the new view is replaced by the delivery of the message m_G and thus every process, having delivered m_G (in view synchronous communication this corresponds to the delivery of the new view), has delivered the same set of messages before delivering m_G. Note that similar to view synchronous communication, messages for the new view are buffered, similarly to the case of multicasts during the execution of the view change protocol described in Chapter 8.4.6, until the new view is delivered.

The proof is based on the correctness of the view change protocol.

9.4 Conclusions

This chapter has given the definition of several powerful communication primitives that provide various ordering properties.

The *weak totally ordered* multicast is similar to the Isis ABCAST [Birman 90], whereas the *strong totally ordered* multicast is not present in Isis. The uniform multicast is not present either in Isis. Other platforms, e.g. Totem [Agarwal 92], provide only *strong total order*, whereas Phoenix provides at the same time the strong and the weak primitive and lets the application choose the adequate primitive.

[Hadzilacos 93] gives some similar, but more exhaustive definitions for total order primitives. A different implementation of the *strong totally ordered* multicast of Phoenix is described in [Anceaume 93].

Chapter 10

Application Programming Interface

10.1 Introduction

Section 4.8 has already described one of the interfaces provided to implement Phoenix applications. Section 10.2 shows the limits of this minimal interface. On top of this minimal interface, Phoenix proposes a more powerful, object-oriented interface. The object-oriented Phoenix programming interface provides fault-tolerance at the object level, and simplifies the programming of complex applications, where the single process control through the callback scheme is no longer sufficient.

This extended interface is based on the *member*, *client* and *sink* entities introduced in Section 2.6. The object-oriented Phoenix interface provides a class for each of these Phoenix process types, allowing one to create instances of them. These classes and their methods are described in Section 10.3. For a detailed description of the object-oriented interface of these Phoenix process types, refer to [Felber 95].

10.2 Limits of the Minimal Programming Infrastructure

The application programming interface described in Chapter 4 provides a minimal, but complete programming infrastructure for group-oriented, fault-tolerant applications. This programming interface is based on the callback mechanism, which can be constraining when programming more complex applications. Consider a server process S_1 receiving requests from a client C. If the server S_1 has to send a request to yet another server S_2 in order to reply to the client C, the limitations of this interface become obvious. The limitation of the callback mechanism is that a callback has to terminate each time it is called. Thus, it cannot block in between, waiting for another request (see Chapter 5).

```
void message_callback (msg) {
    do some handling with message and generate reply
    reliable_send (source(msg), reply);
}
```

Figure 10.1: Typical Message Delivery Callback Code

The program fragment of Figure 10.1 is a typical message callback, implementing a server S_1. Each time a message is received, the callback is called, the corresponding reply is computed and sent back to the client C. Imagine that, in order to compute the reply, this message callback has to invoke another server S_2. As the callback always has to terminate in order to receive other requests and replies, the answer from S_2 will also be delivered through the message delivery callback at server S_1. Thus, server S_1 has to keep track of the requests received and of its own requests to other servers. Upon reception of a message m, server S_1 has to determine, if m is a new request or an answer to one of its own requests (Figure 10.2).

```
void message_callback (msg) {
    if (source (msg) == S2) { /* answer from S2 */
        look for corresponding client to send back reply
        reliable_send (client, msg);
    }
    else {
        do some handling with message and generate request for another server
        reliable_send (S2, newrequest);
        /* cannot wait here on response, because callback has to terminate here;
           save somewhere that request has been sent to S2 */
    }
}
```

Figure 10.2: Server Invoking Another Server

This solution is rather complicated, thus a simpler interface, described in the next section, is provided. The object-orientation of this interface further helps to implement fault-tolerance not only at the process level, but also at the object level.

10.2.1 Implementation of the Concurrent Threaded Callback Mechanism

Integrated in the object-oriented Phoenix interface is a threading system, allowing greater flexibility when programming applications. It is possible to have callbacks which do not have to terminate as is the case in the minimal interface. In the object-oriented interface each message received by the Phoenix system creates a new thread and the message is passed as a parameter to the created thread. The code executed by the thread is the corresponding delivery callback (e.g. the view change callback or the message delivery callback).

Usually threads are executing concurrently within a process. This is not desired in the callback scheme, where only one callback can be executing at one instant. The Phoenix system controls this concurrency by serializing message delivery. A lock, global to all objects, has to be acquired before a new message is delivered in a newly created thread. A terminating thread at the end of the callback frees this lock. This guarantees the mutual exclusion of the delivery of the message and a FIFO management for the acquisition of the lock guarantees that the messages are delivered in the same order as they are received from the daemon.

As the threaded callback mechanism can at any point in its execution block or wait for the synchronization with another thread, it has to inform the Phoenix system when this blocking occurs in order to release the lock. For this purpose the object-oriented Phoenix interface provides two procedures, *allow_concurrency* and *deny_concurrency*, global to all objects, which control this lock[1]. Upon reception of a message the Phoenix system automatically waits until it acquires the lock and releases it upon termination of the thread. If the application wants to allow the delivery of other messages, e.g. just before waiting for the answer to a request (see below), it has to explicitly call the *allow_concurrency* procedure which frees the lock. A thread can acquire the lock, in order to guarantee that no other message is delivered, by calling the *deny_concurrency* procedure; this is typically performed when the answer has been received and should be treated.

Besides this global lock for the control of concurrent message delivery each threading package provides different primitives for the synchronization of the threads. As an example consider Figure 10.3, which is an extension of the server message callback in Figure 10.2, rewritten using the threading feature.

The message delivery callback method corresponds to lines 6 to 20. At the reception of a message, the *concurrency* lock is acquired and a new thread is created. The thread executes the message delivery callback method. At line 7, the server object determines if the message is a request from a client (lines 8 to 14) or an answer from another server (lines 15 to 19).

In the first case, the server forwards the request to *otherserver* (line 9), releases the *concurrency* lock (line 10) and blocks on the call P (*answer*) (line 11). At that point, the Phoenix system is able to again acquire the lock and to deliver new messages.

In the second case the server has received the answer from *otherserver* and it safes the answer in the variable *reply* (line 17), before unblocking the other thread by calling V (*answer*) (line 18). This second thread terminates immediately, i.e. the Phoenix system releases the *concurrency* lock upon termination of the thread. At this point the thread, created for the reception of the

[1]The freeing of this *concurrency lock* does not necessarily have to be done just before a blocking operation, but when the current thread is safe to execute concurrently with another; applications are free to exploit this to execute callbacks concurrently.

```
1      ...
2      Semaphore answer (0);
3      Message reply;
4      ...
5
6  void message_callback (msg) {
7      Determine from where the message comes
8      if (source(msg) == client) {
9          reliable_send (otherserver, othermsg);
10         allow_concurrency ();
11         P (answer);
12         deny_concurrency ();
13         reliable_send (client, reply);
14     }
15     else {
16         /* received answer from 'otherserver' */
17         reply = msg;
18         V (answer);
19     }
20 }
```

Figure 10.3: Sample Code for a Not-Terminating Callback

client request, is unblocked and sends the *reply* back to the client (line 13). Before doing that it may acquire the *concurrency* lock in order to guarantee that no other message is delivered during the rest of the execution of the thread (not needed in the example in Figure 10.3). It is clear that in this way this server can only manage one request at a time, but it is possible to rewrite a message callback, which allows the management of multiple requests.

10.3 Object Oriented Programming Interface

10.3.1 Description

Besides the rather primitive and restrictive minimal interface, Phoenix provides an object-oriented programming interface. This interface provides classes for the three Phoenix process types: *members*, *clients* and *sinks*. To complete this hierarchy, a basic class *team* is defined for Phoenix processes which are not even *sinks*. For a description of the different Phoenix process types refer to Section 2.6; only the class descriptions are given here. The class hierarchy is depicted in Figure 10.4.

In this hierarchy classes have access to the methods of all classes from which they inherit, i.e. the member class inherit all methods from the client, sink and team class. On the other side

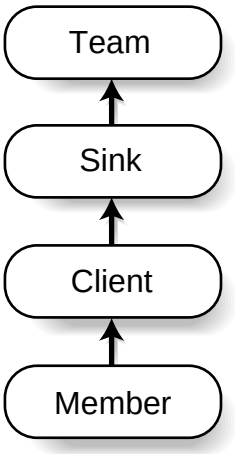

Figure 10.4: Phoenix Class Hierarchy

the client class has no access to the methods of the member class. Further, the access to the inherited classes allows an object of the member class to be client and sink of other processes in the system as well.

Teams

The class *team* is the most primitive class. The term *team* was chosen, besides the fact that a team proposes the minimal interface to be able to communicate in Phoenix, as a team object is the context where a set of threads is ensuring the execution of its methods. This set of threads is called a *team* as they are collaborating in the same context, namely the object. An object of the team class can send and receive messages, as well as receiving acknowledgements, or information about the inability to deliver some message.

```
class Team {
    // callable methods
    void SendMsg (Dest dest, Message msg);

    // methods called by Phoenix (callbacks)
    void MessageDeliveryCB (Message msg);
    void AcknowledgeDeliveryCB (Message msg);
    void UndeliveredDeliveryCB (Message msg);
}
```

Figure 10.5: Class Team

Team Interface Description

SendMsg (dest, msg): this method allows a team to send the message *msg* to any process *dest* in the system.

MessageDeliverCB (msg): this method is called each time a message is received for this

team.

AcknowledgeDeliveryCB (msg): this method is called each time a message sent, using the *SendMsg* method, is successfully delivered at its destination.

UndeliveredDeliveryCB (msg): if a message sent using the *SendMsg* method is not acknowledged after some time, this method is called.

The team class also gives access to the underlying threading system. The threading system is independent of the interface description and is thus not included here. The implementation issues are described and discussed in Section 10.2.1.

The callback methods in the team class (and for all further classes below) are typical candidates to be overwritten by the inheriting class. The methods of the inheriting class will actually deal with the information delivered by the callbacks.

Sinks

The class *sink* inherits from the team class the methods for simple communication. The sink class provides the most primitive Phoenix process type, by proposing a method to become a sink of a group. The following class description shows the most important methods:

```
class Sink : public Team {
    // callable methods
    void SinkSubscribe (char *groupname);
    void SinkUnsubscribe (char *groupname);

    // methods called by Phoenix (callbacks)
    // no called methods in this class
}
```

Figure 10.6: Class Sink

Sink Interface Description

SinkSubscribe (groupname): this method allows a sink to become sink of a group *groupname*.

SinkUnsubscribe (groupname): this method allows a sink to leave the group *groupname* from which it is a sink.

Clients

The class *client* inherits all methods from the sink class and adds its own methods. An object can be both *sink* and *client* of a group g, or *sink* of a group g and *client* of a group g'. To

become client and leave a group, the client class provides corresponding methods described in Figure 10.7:

```
class Client : public Sink {
    // callable methods
    void ClientSubscribe (char *groupname);
    void ClientUnsubscribe (char *groupname);

    // methods called by Phoenix (callbacks)
    void ViewChangeDeliveryCB (View view);
}
```

Figure 10.7: Class Client

Client Interface Description

ClientSubscribe (groupname): this method allows an object to become a client of the group *groupname*.

ClientUnsubscribe (groupname): this method allows an object to leave the group *groupname* of which it is a client.

ViewChangeDeliveryCB (view): as soon as the client is registered at the group, this method is invoked each time the view of the group changes. The parameter *view* contains the new membership of the group.

Members

The *member* class inherits all methods from both client and sink classes. A *member* object of group g can be *client* and/or *sink* of other groups. However, a *member* object can be member of only one group. Figure 10.8 shows the additional methods to enable an object to join, to leave and to multicast in a group.

```
class Member : public Client {
    // callable methods
    void Join (char *groupname);
    void Leave ();
    void Multicast (Message msg, Order ord);

    // methods called by Phoenix (callbacks)
    void IntermediateViewDeliveryCB (View view);
    void PutState (Message msg);
    Msg * GetState ();
}
```

Figure 10.8: Class Member

Member Interface Description

Join (groupname): allows an object to join the group *groupname*.

Leave (): allows an object to leave the group of which it is currently a member.

Multicast (msg, ord): this method allows an object to multicast a message *msg* with an optional parameter of type order, specifying FIFO, uniform, weak total, strong total order or global order.

IntermediateViewDeliveryCB (view): at the beginning and during the execution of view change protocol, this method is invoked each time a different intermediate view is delivered by the Phoenix system.

PutState (Message msg): a new process joining a group receives through this callback the state of the group in the form of a message *msg*. This callback is called after the delivery of a new view, but before the delivery of any message for this view.

*Msg * GetState ()*: this callback is called at a current member, when state transfer to a new member is necessary. In this case the callback is called after the delivery of a new view but before the delivery of any messages for this view. The callback has to return a message containing all the state information in aggregate form which will be sent to the new member and delivered through the *PutState* callback.

Views are delivered through the view change delivery callback inherited from the client class. Messages are delivered through the message delivery callback inherited from the team class.

Building a fault-tolerant system, including members, clients and sinks, is as simple as defining new classes inheriting from the basic classes, and overriding their methods with the ones corresponding to the specification of the system. Appendix C gives some examples of how to program with this object-oriented programming interface.

10.4 Summary

The Phoenix system provides a powerful, object-oriented programming interface. The object-orientation allows the implementation of fault-tolerance at the object level and easy definition of new classes for enhancing and specializing the application interface. Besides the basic classes for *teams*, *sinks*, *clients* and *members*, more sophisticated, higher level classes have been implemented providing adequate support for client/service interaction.

An underlying threading system allows more than one fault-tolerant object per process, and independent objects may execute concurrently within the same process context. The lightweight character of threads allows many objects, members of different groups, in one single process.

Chapter 11

Conclusions and Future Work

Large scale systems are becoming important today. Although upcoming large scale networks, such as the Internet, use basically the same protocols as simple local area networks, existing communication libraries and platforms are often unsuitable for a use in large scale. Whereas site and process failures are the most common type of failure in local area networks, link failures are new types of failure in large scale systems. The availability and coherency of a service running on such a system depend greatly on how the underlying infrastructure deals with these new types of failure.

This thesis has described the Phoenix toolkit, which allows the building of fault-tolerant, distributed applications in large scale networks. Phoenix provides fault-tolerance not only at site and process level, but on a large scale by considering and handling link failures. This fault-tolerance is not limited to the detection of these link failures, but tries actively to mask them as much as possible.

As it is not always possible to mask link failures, a large system can easily partition into smaller subsystems. Consistency in a partitioned environment can be achieved by defining a primary partition. In the absence of such a primary partition there are two ways to deal with the situation: either the platform blocks or the primary partition is lost. Using existing fault-tolerant programming platforms, Isis has taken the primary partition approach with the disadvantage that blocking in the absence of the primary partition is permanent. Horus, or Transis, aware of the problem with primary partitions, have taken the concurrent views approach, with the risk of loosing the primary partition. On a large scale, where partition is not uncommon, the probability of loosing the primary partition is considerable.

Phoenix is in between Isis and Horus, by always providing a primary partition, but blocking is not permanent. An unstable suspicion model, which allows a falsely suspected process to be considered alive again, and the transient character of the partitions, allows the continuation of

the execution with high probability after the repair of a critical partition situation.

A modified consensus protocol, together with eventually reliable channels implemented by the reliable communication layer, is the basic building block for guaranteeing *view synchrony* of communication. This protocol is based on the unstable suspicion model and allows termination as soon as a majority of processes can communicate. This provides a solid base for the main building block of the Phoenix system.

The object-oriented programming interface of the Phoenix toolkit gives adequate support for programming fault-tolerant applications at the object level. This interface provides further abstract objects for the different Phoenix process types, allowing numerically scalable applications to be built.

During the writing of this thesis a prototype has been developed, composed of the different layers described in this document. This prototype has shown the validity, the usefulness and the effectiveness of the approach.

Future Work

The Phoenix toolkit is currently in a prototype state. The algorithms and protocols are implemented, but the current system is not yet ready for wider distribution. The main point remaining is the completion of the functionality of the interface: only the main functionality has been implemented and it has to be completed by an adequate and exhaustive error management. The current lack of corresponding users and programmers documentation is currently filled out by [Doudou 96].

Performance tests on the current Phoenix platform still have to be done, although in large scale networks this is less important than consistency and reliability. The layered architecture of the Phoenix toolkit allows the independent improvement of the performance of each layer. For instance, the use of the IP multicast in the socket interface layer, allows the performance of the reliable communication layer to be improved, especially when Phoenix is used in local area networks. Using the IP multicast and a more optimistic retransmission technique (e.g. negative acknowledgements) would allow a great improvement of communication performance.

A bottleneck in the Phoenix architecture is the daemon. In Isis, *by-pass* communication allows avoidance of the indirection through a daemon, and thus improves performance, but implies problems of scalability. In Phoenix the daemon is actually an active element needed for scalability of the system, as it can represent several groups in one single unit. In case of site failures, this would allow several view change protocol messages to be combined together and the generation of less traffic. Another way to exploit this daemon would be to execute a daemon on a stable machine, whereas applications are connecting to it from unstable machines. To resume, the user should have the choice between the *by-pass* communication or the indirection through

the daemon, depending on the communication, performance and scalability requirements of the application considered.

Finally, thorough testing of all the functionalities of the interface and feedback from users of the platform would make the Phoenix toolkit an attractive, but not competing, alternative when fault-tolerance, consistent state of and suitability to large scale systems is required.

Bibliography

[Accetta 86] M. Accetta, R. Baron, D. Golub, R. Raschid, A. Tevanian & M. Young. *Mach: A new kernel foundation for UNIX development.* In Proc. of the USENIX Summer '86 Conference, pages 93–112. USENIX, June 1986.

[Agarwal 92] D. A. Agarwal, P. M. Melliar-Smith & L.E. Moser. *Totem: A Protocol for Message Ordering in a Wide Area Network.* In Proceedings of the 1st ISMM International Conference on Computer Communications and Networks, pages 1–5, June 1992.

[Allan 74] D. W. Allan, J. E. Gray & H. E. Machlan. *The National Bureau of Standards Atomic Time Scale: generation, stability and accessibility.* Time and Frequency Theory and Fundamentals, pages 205–231, 1974.

[Amir 92a] Y. Amir, D. Dolev, S. Kramer & D. Malki. *Exploiting LAN Broadcast for High Availability Communication Services.* In 6th Israeli Conference on Computer Systems and Software Engineering, 1992.

[Amir 92b] Y. Amir, D. Dolev, S. Kramer & D. Malki. *Transis: A Communication Sub-System for High Availability.* In 22nd Annual International Symposium on Fault-Tolerant Computing, pages 76–84, July 1992.

[Amir 94] Y. Amir, D. Dolev, P. M. Melliar-Smith & L.E. Moser. *Robust and Efficient Replication using Group Communication.* TR 94-20, Inst. of Comp. Sci., The Hebrew University of Jerusalem, 1994.

[Ananda 88] M. P. Ananda, H. Bernstein, W. A. Feess & T. C. Paugstat. *Global Positioning System (GPS) autonomous user system.* Journal of The Navigation, vol. 35, no. 2, pages 197–216, 1988.

[Anceaume 93] E. Anceaume. *Algorithmes de Fiabilisation de Systèmes Réparties.* PhD thesis, Université de Paris-Sud, Centre d'Orsay, January 1993.

[Babaoğlu 94] Ö. Babaoğlu & A. Schiper. *On Group Communication in Large-Scale Distributed Systems.* In Proceedings of the 7th ACM SIGOPS Workshop, September 1994.

[Babaoğlu 95a] Ö. Babaoğlu, R. Davoli, L. Giachini & M. Baker. *Relacs: A Communications Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems.* In Proceedings of the 28th International Conference on System Sciences, pages 612–621, January 1995.

[Babaoğlu 95b] Ö. Babaoğlu, R. Davoli, L. Giachini & P. Sabattini. *The Inherent Cost of Strong-Partial View Synchronous Communication.* In Distributed Algorithms, pages 72–86. Springer Verlag, 1995.

[Babaoğlu 95c] Ö. Babaoğlu, R. Davoli & A. Montresor. *Failure Detectors, Group Membership and View-Synchronous Communication in Partitionable Asynchronous Systems.* Technical report 95-18, University of Bologna, Computer Science Department, November 1995.

[Babaoğlu 96] Ö. Babaoğlu, R. Davoli & G. Dini. *Enriched View Synchrony: A Programming Paradigm for Partitionable Asynchronous Distributed Systems.* Technical report 96-3, University of Bologna, Computer Science Department, July 1996.

[Basu 96] A. Basu, B. Charron-Bost & S. Toueg. *Crash Failures vs. Crash + Link Failures.* In Proc. 15th annual ACM Symposium on Principles of Distributed Computing (brief announcement), 1996.

[Bernstein 87] A.J. Bernstein, V. Hadzilacos & N. Goodman. Concurrency control and recovery in distributed database systems. Addison-Welsey, 1987.

[Birman 87] K. Birman & T. Joseph. *Reliable Communication in the Presence of Failures.* ACM Trans. Comput. Syst., vol. 5, no. 1, pages 47–76, February 1987.

[Birman 91] K. Birman, A. Schiper & P. Stephenson. *Lightweight Causal and Atomic Group Multicast.* ACM Trans. Comput. Syst., vol. 9, no. 3, pages 272–314, August 1991.

[Birman 90] K. Birman, R. Cooper, T. A. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck & M. Wood. *The ISIS System Manual.* Dept of Computer Science, Cornell University, September 90.

[Birrell 84] A. D. Birrell & B. J. Nelson. *Implementing Remote Procedure Calls.* ACM Trans. Comp. Syst., vol. 2, no. 1, pages 39–59, 1984.

[Blair 74] B. E. Blair. *Time and Frequency Dissemination: An Overview of Principles and Techniques.* Time and Frequency Theory and Fundamentals, pages 233–313, 1974.

[Chandra 90] T. D. Chandra & S. Toueg. *Time and Message Efficient Reliable Broadcasts.* In 4th International Workshop on Distributed Algorithms proceedings (WDAG-4). Springer Verlag, November 1990.

[Chandra 95] T.D. Chandra & S. Toueg. *Unreliable Failure Detectors for Reliable Distributed Systems.* Technical report 95-1535, Department of Computer Science, Cornell University, August 1995. To appear in Journal of the ACM. A preliminary version appeared in the *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 325–340. ACM Press, August 1991.

[Chang 84] J. M. Chang & N. Maxemchuck. *Realiable Broadcast Protocols.* ACM Trans. Comput. Syst., vol. 2, no. 3, pages 251–273, August 1984.

[Clark 87] D. D. Clark, M. L. Lambert & L. Zhang. *NETBLT: A Bulk Data Transfer Protocol.* RFC 998, MIT, Cambridge, 1987.

[Comer 88] D. E. Comer. Internetworking with TCP/IP: Principles, protocols, architecture. Prentice Hall, Stevenage, 1988.

[Cooperstock 95] J. R. Cooperstock & S. Kotsopoulos. *Exploiting Group Communication for Reliable High Volume Data Distribution.* IEEE, 1995.

[Davcec 85] D. Davcec & A. Burkhard. *Consistency and Recovery Control for Replicated Files.* In Proceedings of the 10th Symposium on Operating Systems Principles, pages 87–96, 1985.

[Deering 88] S. E. Deering. *Multicast Routing in Internetworks and Extended LANs.* ACM Computer Communications Review, vol. 18, no. 4, pages 55–64, 1988.

[Deering 89] S. E. Deering. *Host extensions for IP multicasting.* RFC 1112, SRI Network Information Center, August 1989.

[Deering 90] S. E. Deering & D. R. Cheriton. *Multicast Routing in Datagram Internetworks and Extended LANs.* ACM Trans. Comp. Syst., vol. 8, no. 2, pages 85–110, May 1990.

[Dolev 93] S. Dolev, A. Israeli & S. Moran. *Self-Stabilization of Dynamic Systems Assuming Only Read/Write.* Journal of Distributed Computing, vol. 7, no. 1, 1993.

[Dolev 94] D. Dolev, D. Malki & R. Strong. *An Asynchronous Membership Protocol that Tolerates Partitions.* TR 94-6, Inst. of Comp. Sci., The Hebrew University of Jerusalem, 1994.

[Dolev 95] D. Dolev, D. Malki & R. Strong. *A Framework for Partitionable Membership Service.* TR 95-4, Inst. of Comp. Sci., The Hebrew University of Jerusalem, 1995.

[Doudou 96] A. Doudou. *Manuel d'utilisation de PHOENIX.* Technical report, LSE - DI - EPF Lausanne, 1996. To appear.

[Eriksson 94] H. Eriksson. *MBone: The Multicast Backbone.* Communications of the ACM, vol. 37, no. 8, pages 54–60, August 1994.

[Felber 95] P. Felber & R. Guerraoui. *Group Programming: An Object Oriented Approach.* In Proceedings of the International Conference on Technology of Object Oriented Languages and Systems (TOOLS), 1995.

[Fink 92] F. E. Fink & R. Robert. *Following the Fiber Distributed Data Interface.* IEEE Network, vol. 6, no. 2, pages 30–40, March 1992.

[Fischer 85] M. Fischer, N. Lynch & M. Paterson. *Impossibility of Distributed Consensus with One Faulty Process.* J. ACM, vol. 32, pages 374–382, April 1985.

[Ford 62] L. R. Ford & D. R. Fulkerson. Flows in networks. Princeton University Press, Princeton, N.J., 1962.

[Gifford 79] D.K. Gifford. *Weighted Voting for Replicated Data.* In Proceedings of the 7th Symposium on Operating Systems Principles, pages 150–159, December 1979.

[Gray 89] C. Gray & D. Cheriton. *Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency.* In Proc. of the 11th ACM Symposium on Operating Systems Principles, pages 202–210, 1989.

[Guerraoui 96] R. Guerraoui & A. Schiper. , *-Accurate Failure Detectors.* In 10th International Workshop on Distributed Algorithms proceedings (WDAG-10). Springer Verlag, October 1996.

[Hadzilacos 93] V. Hadzilacos & S. Toueg. *Fault-tolerant Broadcasts and Related Problems.* In S. Mullender, editor, Distributed Systems, chapter 5, pages 97–146. Addison-Wesley, 1993.

[Hickey 95] T. M. Hickey & R. van Renesse. *Incorporating System Resource Information into Flow Control.* TR 1489, Dept. of Computer Science, Cornell University, Ithaca, NY, Nov 95.

[Jacobson 88] V. Jacobson. *Congestion Avoidance and Control.* Communications architectures and protocols, pages 314–329, August 1988.

[Keidar 94] I. Keidar & D. Dolev. *Increasing the Resilience of Atomic Commit, at No Additional Cost.* TR 94-18, Inst. of Comp. Sci., The Hebrew University of Jerusalem, 1994.

[Keidar 95] I. Keidar. A Highly Available Paradigm for Consistent Object Replication. Master's thesis, Inst. of Comp. Sci., The Hebrew University of Jerusalem, 1995.

[Krol 93] E. Krol & E. Hoffman. *FYI on "What is the Internet?".* Request for Comments (Informational) RFC 1462, Internet Engineering Task Force, May 1993.

[Leffler 89] S. J. Leffler, M. K. McKusick, M. J. Karels & J. S. Quarterman. The design and implementation of the 4.3bsd unix operating system. Addison-Wesley, 1989.

[Leiner 93] B. Leiner & Y. Rekhter. *The MultiProtocol Internet.* Request for Comments (Informational) RFC 1560, Internet Engineering Task Force, December 1993.

[Leslie 93] I. Leslie, D. McAuley & D. L. Tennenhouse. *ATM Everywhere?* IEEE Network, March 1993.

[Liskov 91] B. Liskov, L. Shrira & J. Wroclawski. *Efficient At-most-once Messages Based on Synchronized Clocks.* ACM Trans. Comp. Syst., vol. 9, no. 2, pages 125–142, 1991.

[Liskov 93] B. Liskov. *Practical Uses of Synchronized Clocks in Distributed Systems.* Distributed Computing, no. 6, pages 211–219, 1993.

[Lugeon 94] J.-C. Lugeon & A. Sandoz. *A Light Weight Name Service for Large Scale Distributed Applications and Groupware.* Technical Report 94/63, Swiss Federal Institute of Technology, Lausanne (Switzerland), August 1994.

[Marshall 95] V. A. Marshall. *WWW — The World Wide Web.* Lecture Notes in Computer Science, vol. 1012, 1995.

[Mills 91] D. L. Mills. *Internet Time Synchronization: The Network Time Protocol.* IEEE Transactions on Communications, vol. 39, no. 10, pages 1482–1493, October 1991.

[Mills 92] D. L. Mills. *Network Time Protocol (Version 3).* RFC 1305, University of Delaware, March 1992.

[Moser 94] L. E. Moser, Y. Amir, P. M. Melliar-Smith & D. A. Agarwal. *Extended Virtual Synchrony.* In International Conference on Distributed Computing Systems, pages 56–65, June 1994.

[Reischuck 82] R. Reischuck. *A New Solution for the Byzantine General's Problem.* RJ 3673, IBM Research Laboratory, November 1982.

[Ricciardi 91] A. M. Ricciardi & K. P. Birman. *Using Process Groups to Implement Failure Detection in Asynchronous Environments.* In Proc. of the 10th ACM Symposium on Principles of Distributed Computing, pages 341–352, August 1991.

[Rozier 88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard & W. Neuhauser. *CHORUS distributed operating systems.* Computing Systems Journal, vol. 1, no. 4, pages 305–370, December 1988.

[Schiper 93] A. Schiper & A. Sandoz. *Uniform Reliable Multicast in a Virtually Synchronous Environment.* In IEEE 13th International Conference Distributed Computing Systems, pages 561–568, May 1993.

[Steiner 88] J. G. Steiner, C. Neumann & J. Schiller. *Kerberos: An Authentication Service for Open Network Systems.* Athena project report, MIT, Cambridge, 1988.

[Stephenson 91] P. Stephenson. *Fast Ordered Multicast.* PhD thesis, dept. of Computer Science, Cornell University, May 1991.

[Sun Microsystems Inc 89] Sun Microsystems Inc. *NFS: Network File System Protocol Specification.* RFC 1094, SRI Network Information Center, March 1989.

[Tomlinson 75] R. S. Tomlinson, B. Beranek & N. Cambridge. *Selecting Sequence Numbers.* In Proc. ACM SIGCOMM/SIGOPS Interprocess Communications Workshop. ACM, March 1975.

[van Renesse 92a] R. van Renesse, K. Birman, R. Cooper, B. Glade & P. Stephenson. *Reliable Multicast between Microkernels.* In Proceedings of the USENIX

workshop on Micro-Kernels and Other Kernel Architectures, pages 27–28, April 1992.

[van Renesse 92b] R. van Renesse, R. Cooper, B. Glade & P. Stephenson. *A RISC Approach to Process Groups.* In Proceedings of the 5th ACM SIGOPS Workshop, pages 21–23, September 1992.

[van Renesse 93] R. van Renesse, K. Birman, R. Cooper, B. Glade & P. Stephenson. *The Horus System.* In K. Birman & R. van Renesse, editors, Reliable Distributed Computing with the Isis Toolkit, pages 133–147. IEEE Computer Society Press, 1993.

[Velten 84] D. Velten, R. Hinden & J. Sax. *RDP: Reliable Data Protocol.* RFC 908, SRI Network Information Center, 1984.

[Whetten 94] B. Whetten, T. Montgomery & S. Kaplan. *A High Performance Totally Ordered Multicast Protocol.* In K. Birman, F. Mattern & A. Schiper, editors, Theory and Practice in Distributed Systems, Lecture Notes in Computer Science #938, pages 33–57. Springer-Verlag, 94.

[Wilhelm 95] U. G. Wilhelm & A. Schiper. *A Hierarchy of Totally Ordered Multicast.* In Proceedings of the 14th IEEE Symposium on Reliable Distributed Systems, September 1995.

Appendix A

Protocols

A.1 Dynamic Routing Protocol

This section gives the pseudo code of the dynamic routing protocol described in Chapter 6. Lines 1 to 11 are responsible for initializing the local data structures. Lines 13 to 28 are executed upon reception of a table, and lines 30 to 54 are executed when timeout occurs, i.e. a round is finished.

The pseudo code is based on the primitives introduced in Section 4.4 and Section 4.3. All primitives are prefixed by the layer which exports them: the routing primitives and the callbacks called are prefixed by the word **ROUTE** and the procedures called in the socket interface layer are prefixed by **SOCK**. The handlers installed from the reliable communication layer in the routing layer are generically prefixed by **UPPER**.

In the initialization phase (lines 1 to 11) first the local routing table is initialized (lines 3 to 5). Then, the routing layer installs its callback procedures for messages in the socket interface layer (line 6), and sends out its routing table to all processes (lines 7 to 8). The procedure *compose* flattens the structures passed as parameters in order to generate a message which can be sent to the other destinations. Finally, a timeout callback procedure is installed in the socket interface layer (line 9), which will be called after the specified timeout τ. Line 10 initializes the first *CommSet*, which allows later testing (in line 46) whether the *CommSet* has changed.

The procedure *ROUTE_message_callback* (lines 13 to 28), installed in the socket interface layer, is called at each message reception. First, the procedure verifies if the message has reached its destination by extracting some header information using the procedure *extract*. If the message has to be rerouted (lines 15 to 16) the routing layer calls its own send message, which will route and send the message accordingly. If the message has reached its destination the callback has to distinguish two cases: (1) it is a routing table or (2) a regular message. In case of a regular

message, the routing layer delivers the message through the callback installed from the overlying layer (line 27). Otherwise, the routing layer has received a routing table from another process and has to inspect the routing tables received (lines 18 to 25). The procedure *extract* in line 18 rebuilds the data structures flattened using the procedure *compose*. Line 19 resets the timeout counter as the process has just received information from the process. In lines 20 to 25, every entry in the routing received is matched against the local routing table in order to detect an unreachability of a process (lines 21 to 23) or a shorter path through the process from which the table was received (lines 24 to 25).

The procedure *ROUTE_timeout_callback* (lines 30 to 54), installed in the socket interface layer with a timeout value τ, is called after the specified period; one round of the routing protocol is over. For every process the procedure increments the timeout counter (line 32). If the destination is suspected and messages are not rerouted through another process to this destination, the process is suspected (expressed here by setting its distance to ∞ in line 34). Further, all processes which have been reachable through the newly suspected process also becomes unreachable (lines 35 to 37). Lines 38 to 42 are responsible for detecting any communication problems with some destination without suspecting it immediately; this will be done after $\lambda/2$ rounds and the protocol looks for another route before suspecting. A new route will be detected by assigning to the current route a distance longer than any other route (line 39)[1]. Like the suspecting case above, all destinations which are reachable from the problem destination, are also subject to the search for another route (lines 40 to 42). Once the new routing table has been computed the routing layer determines which processes are reachable (lines 44 to 45). If there is a change in the reachability of one or more processes the reachability callback installed from the overlying layer is called (line 46 to 47) with the new reachability sets. Finally, the next round of the routing protocol is launched (lines 50 to 53); the steps are identical to the those in lines 7 to 9.

```
 1  procedure ROUTE_initialize_routing()
 2      /* installation of the message callback in the socket interface layer */
 3      R[1..n] := ⟨1, 2, ..., n⟩;
 4      δ[1..n] := ⟨1, ..., n⟩; δ[myindex] := 0;
 5      λ[1..n] := ⟨0, ..., 0⟩; λ[myindex] := −1;
 6      SOCK_set_message_callback (ROUTE_message_callback);
 7      for i ∈ P − {myindex} do
 8          SOCK_send(i, compose(i, myindex, TABLE, R, δ));
 9      SOCK_set_timeout_callback (ROUTE_timeout_callback, τ);
10      PrevCommSet := {∀i | δ[i] < ∞};
11  endproc
12
13  procedure ROUTE_message_callback(m)
14      extract(m, to, from, type);
15      if to ≠ myindex then
16          ROUTE_routed_send(to, from, type, m);
```

[1]Note that for the case of suspicion (line 34) the distance becomes ∞, whereas for the case of looking for a new route (line 39) the distance becomes n.

```
17      elsif type = TABLE then
18          extract(m, to, from, type, r, d);
19          λ[from] := −1;
20          for i ∈ P − {myindex} do
21              if d[i] = ∞ then
22                  if R[i] = from then
23                      R[j] := j; δ[j] := ∞;
24              else if d[i] + δ[from] < δ[i] then
25                  R[i] := r[i]; δ[i] := d[i] + δ[from];
26      else
27          UPPER_message_callback(m);
28  endproc
29
30  procedure ROUTE_timeout_callback()
31      for i := 1 to n do
32          λ[i] := λ[i] + 1;
33          if λ[i] ≥ λ and R[i] = i then
34              R[i] := i; δ[i] := ∞;
35              for j := 1 to n do
36                  if R[j] = i then
37                      R[j] := j; δ[j] := ∞;
38          else if λ[i] ≥ λ/2 and R[i] = i then
39              R[i] := i; δ[i] := n;
40              for j := 1 to n do
41                  if R[j] = i then
42                      R[j] := j; δ[j] := n;
43      /* test if routing tables has changed */
44      CommSet := {∀i | δ[i] < ∞};
45      SuspSet := {∀i | δ[i] = ∞};
46      if CommSet ≠ PrevCommSet then
47          UPPER_reachability_callback(CommSet, SuspSet);
48          PrevCommSet = CommSet;
49      /* launch next round */
50      λ[myindex] := −1;
51      for i ∈ P − {myindex} do
52          SOCK_send (i, compose(i, myindex, TABLE, R, δ));
53      SOCK_set_timeout_callback(ROUTE_timeout_callback, τ);
54  endproc
```

Figure A.1: Routing Protocol

A.2 Consensus Protocol of Chandra and Toueg

Figure A.2 gives the pseudo code for the consensus algorithm described in [Chandra 95]. It allows one to compare it with the view synchronous communication protocol in Section A.3.

Each process p of the set P of participating processes starts with estimate value $estimate_p$ and round 0 (lines 1 to 5). Then each process executes either phases 1 and 3 or phases 2 and 4, depending whether the process is participant or coordinator. In phase 1 (lines 12 and 13), a participant sends its estimate to the coordinator. In phase 2 (lines 15 to 21), the coordinator waits for a majority of estimates and generates a proposition as described Section 8.4.4. In phase

3 (lines 23 to 30) the participants receive the proposition of the coordinator or suspect it (line 25). When a participant adopts a proposition, it sends an acknowledgement to the coordinator (lines 26 to 29); otherwise it sends a negative acknowledgement (line 30). The coordinator, in phase 4 (lines 32 to 36), collects the positive and negative acknowledgements (line 34) and broadcasts the decision value (line 36), after having received a majority of positive acknowledgements (line 35). Lines 39 to 43 describe the action taken upon reception of the decision.

```
 1  procedure propose(v_p)
 2      estimate_p := v_p;
 3      state_p := undecided;
 4      r_p := 0;
 5      ts_p := 0;
 6
 7      { Rotate through coordinators until decision is reached }
 8      while state_p = undecided do
 9          r_p := r_p + 1;
10          c_p := (r_p mod n) + 1;
11
12          Phase 1: { All processes p send estimate_p to the coordinator }
13              send (p, r_p, estimate_p, ts_p) to c_p;
14
15          Phase 2: { The current coordinator gathers a majority of estimates }
16              if p = c_p then
17                  wait until [for ⌈(n+1)/2⌉ processes q: received (q, r_p, estimate_q, ts_q) from q];
18                  msgs_p[r_p] := {(q, r_p, estimate_q, ts_q) | p received (q, r_p, estimate_q, ts_q) from q};
19                  t := largest ts_q such that (q, r_p, estimate_q, ts_q) ∈ msgs_p[r_p];
20                  estimate_p := select one estimate_q such that (q, r_p, estimate_q, t) ∈ msgs_p[r_p];
21                  send (p, r_p, estimate_p) to all;
22
23          Phase 3: { All processes wait for the new estimate proposed by the current
24                     coordinator or suspect coordinator }
25              wait until [received (c_p, r_p, estimate_{c_p}) from c_p or c_p ∈ ◇S_p];
26              if [received (c_p, r_p, estimate_{c_p}) from c_p] then
27                  estimate_p := estimate_{c_p};
28                  ts_p := r_p;
29                  send (p, r_p, ack) to c_p;
30              else send (p, r_p, nack) to c_p;
31
32          Phase 4: { The current coordinator waits for a majority of replies of all members }
33              if p = c_p then
34                  wait until [for ⌈(n+1)/2⌉ processes q: received (q, r_p, ack) or received (q, r_p, nack)];
35                  if [for ⌈(n+1)/2⌉ processes q: received (q, r_p, ack)] then
36                      Rbroadcast (p, r_p, estimate_p, decide);
37
38
39  { When p receives a decide message, it decides }
40  when Rdeliver (q, r_q, estimate_q, decide)
41      if state_p = undecided then
42          decide(estimate_q);
43          state_p := decided;
```

Figure A.2: Consensus Protocol

A.3 View Synchronous Communication Protocol

Figure A.3 gives the protocol executed during a view change of a group.

Before giving the pseudo code some of the major differences between the two protocols are explained. As in the consensus protocol, each process p in the current view[2] V_i starts with an estimate (line 2). The estimate consists of (1) the set $VscMsgSet_p$ including all messages not stable when the protocol is launched, and (2) $CommSet_p$ giving a current view of the processes reachable from p. The protocol has the same phases as the original protocol. In phase 1 (lines 12 and 13) the participant sends its estimate (the two sets) to the coordinator. In phase 2 the differences are important: in contrast to the original protocol, the coordinator waits not only for majority of estimates, but also for the suspicions of all processes in view V_i from which the coordinator has not yet received an estimate (line 18 to 20). Once the condition is satisfied, the coordinator distinguishes the case where it can build a new proposition (lines 22 to 27) from the case where there already exists a proposition (lines 29 to 30); for a detailed description refer to Section 8.4.5. In both cases, a proposition is sent to the participants (line 31). Phase 3 as well as phase 4 of both protocols are identical. A last difference is the way a decision is taken (lines 49 to 57). A process receiving a decision (line 50), checks if it is still part of the decided, new view (line 54). If it is excluded (line 55) it has to act accordingly, for instance join the group again. Otherwise the new view is saved and delivered (line 57).

```
 1  procedure propose(v_p)
 2      estimate_p := {CommSet_p, VscMsgSet_p};
 3      state_p := undecided;
 4      r_p := 0;
 5      ts_p := 0;
 6
 7      { Rotate through coordinators until decision is reached }
 8      while state_p = undecided do
 9          r_p := r_p + 1;
10          c_p := (r_p mod n) + 1;
11
12          Phase 1: { All processes p ∈ V_i send estimate_p to the coordinator }
13              send (p, r_p, estimate_p, ts_p) to c_p;
14
15          Phase 2: { The current coordinator gathers a majority of estimates
16                                  and a minority of failure suspicions in view V_i }
17              if p = c_p then
18                  wait until [for at least ⌈(n+1)/2⌉ processes q:
19                                  received (q, r_p, estimate_q, ts_q) from q or q ∈ ◇S_p(, )];
20                      { will never reach this point if not received at least a majority of estimates }
21                  msgs_p[r_p] := {(q, r_p, estimate_q, ts_q) | p received (q, r_p, estimate_q, ts_q) from q};
22                  if ts_p = 0 then
23                      { There has not yet been a successful round with a proposition;
24                        generate a proposition }
```

[2]The current view is the decided $CommSet_p$ of the previous execution of the protocol.

```
25                  estimate_p.CommSet := V_i − {q | q ∈ ◇S_p(, )};    { V_i is the current view }
26                  estimate_p.VscMsgSet := ⋃_q (estimate_q.VscMsgSet) |
27                                          (q, r_p, estimate_q, ts_q) ∈ msgs_p[r_p];
28              else
29                  t := largest ts_q such that (q, r_p, estimate_q, ts_q) ∈ msgs_p[r_p];
30                  estimate_p := select one estimate_q of (q, r_p, estimate_q, t) ∈ msgs_p[r_p];
31              send (p, r_p, estimate_p) to all;
32
33          Phase 3: { All processes wait for the new estimate
34                          proposed by the current coordinator }
35              wait until [received (c_p, r_p, estimate_{c_p}) from c_p or c_p ∈ ◇S_p(, )];
36              if [received (c_p, r_p, estimate_{c_p}) from c_p] then
37                  estimate_p := estimate_{c_p};
38                  ts_p := r_p;
39                  send (p, r_p, ack) to c_p;
40              else send (p, r_p, nack) to c_p;
41
42          Phase 4: { The current coordinator waits for a majority of replies of all processes of V_i }
43              if p = c_p then
44                  wait until [for ⌈(n+1)/2⌉ processes q: received (q, r_p, ack) or received (q, r_p, nack)];
45                  if [for ⌈(n+1)/2⌉ processes q: received (q, r_p, ack)] then
46                      Rbroadcast (p, r_p, estimate_p, decide);
47
48
49  { When p receives a decide message, it decides }
50  when Rdeliver (q, r_q, estimate_q, decide)
51      if state_p = undecided then
52          decide(estimate_q);
53          state_p := decided;
54          if p ∉ estimate_q.CommSet then
55              { was excluded, join the group again }
56          else
57              newview := estimate_q.CommSet;
```

Figure A.3: View Change Protocol

Appendix B

The Phoenix Name Service

B.1 Introduction

In several chapters of this thesis, a name server has been mentioned, but never presented. This section gives the description of the name server.

A name server allows the mapping of group names to a set of sites on which members of the group are present. Group names have to be system wide unique identifiers. Further, the name server allows the information to be modified when the composition of the group changes, when a new group is created, or when a group has ceased to exist.

One particular problem of a name service is the creation of a group. Suppose some group g does not yet exist, and two processes p and q want to create it at the same time. This problem is solved using a *lookup-and-register* operation in the name server, which looks up the composition of the group and if the group does not exist it is created and the requesting process becomes the first member of the group. Finally, the name server returns the looked up composition to the requesting process. The name server is updated by the group members, each time the view changes.

B.2 Implementation

B.2.1 Light Weight Name Server

The Phoenix name server is at the moment based on the *light weight name server* described in [Lugeon 94]. The major lack of this name server is the absence of fault-tolerance (the name server is not replicated).

B.2.2 A Fault-Tolerant Name Server

One way to render the name service for Phoenix fault-tolerant is to write it as a Phoenix application. However, this leads to a boot-strapping problem. An exhaustive description of the solution to this problem is out of the scope of this thesis.

The simple solution consists of designating a static group whose members and their localization are known by all applications requiring access to the name server.

B.2.3 Accessing the Name Server

The methods *Join* and *ViewChangeDeliveryCB* of a member, or the methods *SinkSubscribe* or *ClientSubscribe* interact with the name server group either to register or to obtain the desired information about a group. The *ViewChangeDeliveryCB* method of a member updates the information of the name service (new membership for a group). The name server group can be seen as an implicit sink of all existing groups.

Appendix C

Examples of Phoenix Applications

C.1 Small Phoenix Application

The following section of code shows a very small Phoenix application. The application creates an object of class *CSmallProcess* which joins a group given as a parameter on the command line. Then the object multicasts a message to the group and waits for a user input. The user input determines either to multicast another message, or to leave the group.

A class *CSmallProcess* inheriting of the class *PHX_Member* is defined (lines 11 to 29). Besides the constructor and the destructor, the following methods are overridden: *ViewChange*, *Body*, and *Receive*.

The constructor (lines 42 to 46) does nothing else than print a message, confirming that the object has been created.

The *ViewChange* method (lines 48 to 53) receives as parameter a *group object*. This *group object* has a *Dump* method allowing the group members to be listed.

The *Body* method (lines 55 to 88) is executed as soon as the object is instantiated. The first action for the object to do is to join the desired group (line 61); the group name is defined as a command line argument and passed to the object through the global variable *gname* at line 36. Then, the object allows Phoenix to deliver new messages (line 67) and enters a loop where it multicasts a message to the joined group with the total order primitive (line 73). At this point the user can enter either any character except 'q' to continue the multicasts, or the character 'q' to leave the loop. Once the loop has been left, exclusive execution is requested at line 80 and the group is left. Finally, the call to *PHOENIX::ExitMainLoop* (line 87) terminates the main loop launched at line 113.

The *Receive* method (lines 90 to 94) prints a message on the standard output each time a message

is received.

The main procedure (lines 96 to 121) receives the group name to join as a parameter and stores this name in the global variable *gname* (see above). Then the Phoenix library is initialized (line 107), the main procedure creates an object of class *CSmallProcess* (line 110) and executes the main loop (line 113) which can be terminated with a call to the procedure *PHOENIX::ExitMainLoop* (see above). Once the main loop has been left the created object is destroyed and the Phoenix library is advised to terminate.

small.cc

```
1   #include "phoenix.h"
2   #include "debug.h"
3
4    ////////////////////////////////////////
5    // Type definitions
6    ////////////////////////////////////////
7
8    // Smallest Phoenix process sending and receiving
9    // multicasts and view changes having an active body
10
11  class CSmallProcess : public PHX_Member  {
12
13  public:
14   // Constructor & Destructor
15      CSmallProcess ();
16      ~CSmallProcess () {}
17
18  protected:
19   // Event handling
20       // Called upon viewchange
21      void ViewChange (PHX_Group& grp);  // Override
22       // Thread associated with the object (active object)
23      void Body ();  // Override
24
25  private:
26       // Receive a request from a client
27      void Receive (PHX_Msg& msg, PHX_Id& srcid,
28                                ULONG msgid);
29  } ;
30
31   ////////////////////////////////////////
32   // Local Scope Variables
33   ////////////////////////////////////////
34
35   // Name of the joined group
36  char *gName;
37
38   ////////////////////////////////////////
39   // Class Methods
40   ////////////////////////////////////////
41
42  CSmallProcess::CSmallProcess () :
43                            PHX_Member (ACTIVE)
44  {
45      printf ("Process launched\n");
46  }
47
48  void CSmallProcess::ViewChange (PHX_Group& grp)
49  {
50      printf ("\n***********\nReceived ViewChange:\n");
51       // Dump the new group composition to the screen
52      grp.Dump (stdout);
53  }
54
55  void CSmallProcess::Body ()
56  {
57      PHX_Msg * mcastmsg;
58
59       // Join the specified group
60      printf ("Joining group\n");
61      this->Join (gName);
62
63       // Since performing a blocking operation (reading a
64       // character from the keyboard), detach thread
65       // from the Phoenix scheduling. That way thread
66       // does not block other operations.
67      PHOENIX::AllowConcurrency ();
68      char c;
69      do {
70          // multicast a message
71          mcastmsg = new PHX_Msg (14);
72          mcastmsg->Put ("Hello, World!", 14);
73          Mcast (*mcastmsg, 0, TOTALORDER);
74
75          printf ("\n(Q)uit\n");
76          cin>> c;
77      } while (toupper (c) != 'Q');
78       // Before calling a Phoenix function
79       // reenter the Phoenix scheduling
80      PHOENIX::DenyConcurrency ();
81
82       // Leave the previously joined group
83      printf ("Leaving group\n");
84      this->Leave ();
85
86       // Exit Phoenix main loop
87      PHOENIX::ExitMainLoop ();
88  }
89
90  void CSmallProcess::Receive (PHX_Msg& msg,
91                              PHX_Id& srcid, ULONG msgid)
92  {
93      printf ("Received a message\n");
94  }
95
96  int main (int argc, char **argv)
97  {
98      if (argc != 2) {
99          printf ("Usage: %s groupname\n", argv[0]);
100         exit (1);
101     }
102
103      // Store the group name
104     gName = argv[1];
105
106      // Initialize Phoenix
107     PHOENIX::Init ();
108
109      // Create an "small process" object
110     CSmallProcess *process = new CSmallProcess;
111
112      // Call Phoenix main loop
113     PHOENIX::MainLoop ();
114
115     delete process;
116      // Leave Phoenix cleanly
117     PHOENIX::Exit ();
118
119     printf ("End of program\n");
120     return (0);
121 }
```

C.2 Mini News System

This section shows the implementation of a fault-tolerant mini news system using the Phoenix toolkit. The system allows articles to be posted to a fault-tolerant server and allows one to have sinks getting these articles. There are the three distinct applications:

news base: the news base service application implements the fault-tolerant service. A news client (described below) can post articles which are sequentially numbered by the news base service application. Further a news client can request an article with a given number. In order to be fault-tolerant, there is more than one news base server in the same group. All new articles are multicast to the group of the news base servers.

news client: the news client application allows a user to post articles to the news base service. For this purpose it is sufficient to send the article to one member of the group, which will then multicast the article to the group. As the news client application becomes client of the news base service, it is informed about view changes in the news base service and consequently has access as always to at least one of the current members of the service. A news client can also request articles from the news base service by indicating the number of the desired article.

news sink: the news sink is an application which becomes a sink of the news base service. Each time the news base is updated the whole article base is sent to all the news sinks. In this way lost updates and new sinks are all updated in the same way.

In the following subsections the reader will find a short description of the implementation and the complete code for the *news client*, *news base* and *news sink* applications. All these applications take as a parameter the name of the group which is considered (*news client* becomes a client, *news base* a member, and *news sink* a sink of the group). In order to keep the code section as simple as possible, only a minimum of error checking is done.

C.2.1 News Client

The news client application defines a *CPoster* class (lines 12 to 37) inheriting from the Phoenix *PHX_Client* class. An object of the *CPoster* class subscribes to the group of news servers as a Phoenix client. Besides the constructor and the destructor, this class overrides the methods *ViewChange*, *Receive* and *Body*.

The *ViewChange* method (lines 50 to 60) is called each time the group membership changes (see Section 2.6). The method keeps track of the group (line 53) and determines one member (line 59) to whom requests will be sent.

The *Receive* method (lines 62 to 69) prints the messages received from the news base.

The *Body* method (lines 71 to 160) first subscribes to the specified group as a client (lines 73 to 87), and then asks the user if he wants (1) to post a new article to the base (lines 102 to 125) or (2) to get an article from the base (lines 126 to 151). In both cases, the corresponding parameters are asked of the user, and the request is sent to one member of news base (line 121 or line 146). If the user decides to quit the program, lines 152 to 159 are executed: the object leaves the group and exits the Phoenix main loop.

The main procedure (lines 162 to 186), is similar to the example in Section C.1, i.e. Phoenix is initialized, an object created and the Phoenix main loop launched. When this main loop is terminated the object is destroyed and the application terminates.

newsclient.cc

```
1   #include "phoenix.h"
2   #include "debug.h"
3
4   #include <ctype.h>
5
6   #define MAXARTICLELEN 1000
7
8    ////////////////////////////////////////
9    // Type definitions
10   ////////////////////////////////////////
11
12   // A class for representing a post and read application
13   // (from where you can post and read articles)
14
15  class CPoster : public PHX_Client  {
16
17  public:
18   // Constructor & Destructor
19      CPoster () : PHX_Client (ACTIVE) {}
20      ~CPoster () {}
21
22  protected:
23   // Event handling
24       // Called upon viewchange
25      void ViewChange (PHX_Group& grp);  // Override
26       // Called when message received
27      void Receive (PHX_Msg& msg, PHX_Id& srcid,
28                                          ULONG msgid);
29       // Thread associated with the object (active object)
30      void Body ();  // Override
31
32  private:
33       // current state of the group of article databases
34      PHX_Group fGroup;
35       // one member to send requests to
36      PHX_Id fOneMember;
37  } ;
38
39   ////////////////////////////////////////
40   // Local Scope Variables
41   ////////////////////////////////////////
42
43   // Name of the group of article databases
44  char *gName;
45
46   ////////////////////////////////////////
47   // Class Methods
48   ////////////////////////////////////////
49
50  void CPoster::ViewChange (PHX_Group& grp)
51  {
52      list<PHX_Id>& members = fGroup.Members ();
53      fGroup = grp;
54      printf ("\nReceived ViewChange:\n");
55       // Dump the new group composition to the screen
56      grp.Dump (stdout);
57       // Choose one member to send requests to
58      list<PHX_Id>::const_iterator i = members.begin ();
59      fOneMember = (*i);
60  }
61
62  void CPoster::Receive (PHX_Msg& msg,
63                 PHX_Id& srcid, ULONG msgid)
64  {
65      char reply[MAXARTICLELEN];
66
67      msg.Get (reply, MAXARTICLELEN);
68      printf ("%s\n", reply);
69  }
70
71  void CPoster::Body ()
72  {
73       // Prepare own ID for sends
74      int phxsize = this->Id ().XdrSize ();
75      char *myphxid = (char *) malloc (phxsize);
76      this->Id ().XdrWrite (myphxid);
77
78       // Subscribe the specified group as a client
79      printf ("Subscribing group\n");
80      this->ClientSubscribe (gName);
81
82       // Since performing a blocking operation (reading a
83       // character from the keyboard), detach thread
84       // from the Phoenix scheduling. That way thread
85       // does not block other operations.
86      PHOENIX::AllowConcurrency ();
87      printf ("\n(P)ost\n(R)ead\n(H)elp\n(Q)uit\n\n");
88
89      while (1) {
90          char command[50], c;
91          cin.getline (command,50);
92
93          c = command[0];
94          c = toupper (c);
95
96          if (c == 'Q')
97              break;
98
99          if (c == 'H')
100             printf ("\n(P)ost\n(R)ead\n(H)elp\n(Q)uit\n\n");
101
102         if (c == 'P') {
103             char text[MAXARTICLELEN];
104             int request = 0;  // post an article
105             int length;
106
107             printf ("Article text (end with ';') : \n");
108             cin.getline (text, MAXARTICLELEN, ';');
109
110             length = strlen (text);
111             PHX_Msg msg (2 * sizeof (int) +
112                          this->Id ().XdrSize () + length);
113             msg.Put (&request, sizeof (int));
114             msg.Put (myphxid, phxsize);
115             msg.Put (&length, sizeof (int));
116             msg.Put (text, length);
117
118              // Before calling a Phoenix function,
119              // reenter the Phoenix scheduling...
120             PHOENIX::DenyConcurrency ();
121             this->Send (fOneMember, msg);
122
123              // ...and exit it once more
124             PHOENIX::AllowConcurrency ();
125         }
126         if (c == 'R') {
127             int request = 1;  // read an article
128             int nb;
129
130             printf ("Article number: ");
131
132             char command[50], c;
133             cin.getline (command,50);
134
135             nb = atoi (command);
136
137             PHX_Msg msg (2 * sizeof (int) +
138                          this->Id ().XdrSize ());
139             msg.Put (&request, sizeof (int));
140             msg.Put (myphxid, phxsize);
141             msg.Put (&nb, sizeof (int));
142
143              // Before calling a Phoenix function,
144              // reenter the Phoenix scheduling...
```

```
145             PHOENIX::DenyConcurrency ();
146             this->Send (fOneMember, msg);
147
148             // ...and exit it once more
149             PHOENIX::AllowConcurrency ();
150         }
151     }
152     PHOENIX::DenyConcurrency ();
153
154     // Leave the previously subscribed group
155     printf ("Unsubscribing group\n");
156     this->ClientUnsubscribe (gName);
157
158     // Exit Phoenix main loop
159     PHOENIX::ExitMainLoop ();
160 }
161
162 int main (int argc, char **argv)
163 {
164     if (argc != 2) {
165         printf ("Usage: %s groupname\n", argv[0]);
166         exit (1);
167     }
168
169     // Store the group name
170     gName = argv[1];
171
172     // Initialize Phoenix
173     PHOENIX::Init ();
174
175     // Create a poster object
176     CPoster *poster = new CPoster;
177
178     // Call Phoenix main loop
179     PHOENIX::MainLoop ();
180
181     delete poster;
182     // Leave Phoenix cleanly
183     PHOENIX::Exit ();
184
185     return (0);
186 }
```

C.2.2 News Base

The news base application defines a class *CArticle* (lines 16 to 54) whose methods allow the contents of an article to be read and two article numbers to be compared (this will be used by the *CArticleDB* class, which manages articles in a list). Each instance of the *CArticle* class contains one article. The class *CArticleDB* (DB for database) is the manager of the article base, and thus inherits from the Phoenix *PHX_Member* class (lines 58 to 96). In addition to the previous examples, the *CArticleDB* class overrides the methods *PutState* and *GetState* (lines 70 and 71); these methods allow transfer of the article data base from the current member to a new member (see below). Further, *Read* and *Write* methods are introduced for reading existing and writing new articles (lines 81 and 83). The *SendReply* method (line 88) is used to send an answer back to a news client. The method *SendStateToSinks* (line 91) is responsible for sending the article data base to the sinks each time the data base changes.

The *ViewChange* method (lines 115 to 120) prints the current composition of the group on the standard output.

The *Body* method (lines 122 to 148) is only for joining the group and for waiting for a user input to leave the group.

The *SendReply* method (lines 150 to 157) is used to send back a message (either an acknowledgement for a write request, or the article contents for a read request) to a news client.

The *Write* method (lines 159 to 190) is invoked at each member as new articles are multicast to the group). It adds a new article to the article data base (lines 159 to 182), sends back a reply to the news client using the *SendReply* method (line 183). If the member is the first in the group (line 185), it invokes the *SendStateToSinks* method described below.

The *Read* method (lines 192 to 224) is similar to the *Write* method, but there is no update information to send to the sinks.

The *Receive* method (lines 226 to 254) is invoked at the reception of a message and dispatches the call to the corresponding methods. As new articles are first multicast to the group before they are added to the data base, the *Receive* method has to distinguish between requests received from a news client (lines 236 to 246) and the reception of the multicast. A write request from a news client is multicast to the group (lines 236 to 242)[1] and a read request is handled immediately (line 245). As the news client issues the request to only one member of the group, there is at most once reply. Upon reception of the multicast corresponding to a write request, the *Write* method is invoked.

[1]Remember that a member receives itself the multicast that it has issued.

The methods *GetState* (lines 258 to 284) and *PutState* (lines 286 to 309) are responsible for putting the whole article data base in one message, or for extracting the whole data base from a message and build the data base.

The *SendStateToSinks* method (lines 311 to 317) invokes the *GetState* method and sends the generated message to all sinks using the *SendToSinks* method provided by the *PHX_Member* class.

The *Dump* method (lines 319 to 326) is used only for debugging purposes.

The main procedure is similar to all the former main procedures.

newsbase.cc

```
1   #include <string.h>
2
3   #include "phoenix.h"
4   #include "debug.h"
5
6   #include <ctype.h>
7
8   #define MAXARTICLELEN 1000
9
10   ////////////////////////////////////////
11   // Type definitions
12   ////////////////////////////////////////
13
14   // A simple class for representing the news database
15
16   class CArticle {
17
18   public:
19   // Constructor & Destructor
20      CArticle ()
21         { fNb = 0; }
22      CArticle (int nb)
23         { fNb = nb; }
24      CArticle (int nb, char *text);
25
26   // Operators
27      // Compare 2 articles for equality
28      bool operator ==(const CArticle & a) const
29         { return (fNb == a.fNb); }
30
31   // Access
32      // Read contents of an article
33      char * Read () const { return (fText); }
34      // Return the article number
35      int Nb () const { return (fNb); }
36      // Return the length of an article
37      int Length () const { return (fLength); }
38
39   private:
40      // Article number
41      int fNb;
42      // Article length
43      int fLength;
44      // Article contents
45      char *fText;
46   ;
47
48   CArticle::CArticle (int nb, char *text)
49   {
50      fNb = nb;
51      fLength = strlen (text);
52      fText = (char *) malloc (fLength);
53      strcpy (fText, text);
54   }
55
56   // A class for representing a database of articles
57
58   class CArticleDB : public PHX_Member {
59
60   public:
61   // Constructor & Destructor
62      CArticleDB ();
63      ~CArticleDB () {}
64
65   protected:
66   // Event handling
67      // Called upon viewchange
68      void ViewChange (PHX_Group& grp); // Override
69      // Called upon state transfer
70      PHX_Msg * GetState (); // Override
71      void PutState (PHX_Msg& state); // Override
72
73      // Thread associated with the object (active object)
74      void Body (); // Override
75
76      // Dump the internal state of the objects (i.e. all articles)
77      void Dump (FILE *f);
78
79   private:
80      // Write an article in the database
81      void Write (PHX_Msg& msg);
82      // Read an article in the database
83      void Read (PHX_Msg& msg);
84      // Receive a request from a client
85      void Receive (PHX_Msg& msg, PHX_Id& srcid,
86                                    ULONG msgid);
87      // Send a reply to a requester
88      void SendReply (PHX_Id& srcid, char *msg);
89      // Send the internal state (i.e. all articles)
90      // to the sink members
91      void SendStateToSinks ();
92
93      // List of all articles
94      list<CArticle> fArticles;
95      int fLastNumber;
96   ;
97
98   ////////////////////////////////////////
99   // Local Scope Variables
100  ////////////////////////////////////////
101
102  char *gName;
103  // Name of the group of article databases
104
105  ////////////////////////////////////////
106  // Class Methods
107  ////////////////////////////////////////
108
109  CArticleDB::CArticleDB () : PHX_Member (ACTIVE)
110  {
111     printf ("Creating database\n");
112     fLastNumber = 0;
113  }
114
115  void CArticleDB::ViewChange (PHX_Group& grp)
116  {
117     printf ("\nReceived ViewChange:\n");
118     // Dump the new group composition to the screen
119     grp.Dump (stdout);
120  }
121
122  void CArticleDB::Body ()
123  {
124     // Subscribe to the specified group
125     printf ("Joining group\n");
126     this->Join (gName);
127
128     // Since performing a blocking operation (reading a
129     // character from the keyboard), detach thread
130     // from the Phoenix scheduling. That way thread
131     // does not block other operations.
132     PHOENIX::AllowConcurrency ();
133     char c;
134     do {
135        printf ("\n(Q)uit\n");
136        cin>> c;
137     } while (toupper (c) != 'Q');
138     // Before calling a Phoenix function
139     // reenter the Phoenix scheduling
140     PHOENIX::DenyConcurrency ();
141
142     // Leave the previously joined group
143     printf ("Leaving group\n");
144     this->Leave ();
```

```
145
146         // Exit Phoenix main loop
147         PHOENIX::ExitMainLoop ();
148     }
149
150     void CArticleDB::SendReply (PHX_Id& srcid, char *msg)
151     {
152         // Put string in a Phoenix message and send reply
153         int len = strlen (msg) + 1;
154         PHX_Msg rply (len);
155         rply.Put (msg, len);
156         this->Send (srcid, rply);
157     }
158
159     void CArticleDB::Write (PHX_Msg& msg)
160     {
161         int    length;
162         char articletext[MAXARTICLELEN];
163         PHX_Id srcid;
164         int phxsize = srcid.XdrSize ();
165         char *placeholder = (char *) malloc (phxsize);
166
167         // Read in the source of the request
168         msg.Get (placeholder, phxsize);
169         srcid.XdrRead (placeholder);
170         // Read in the article length
171         msg.Get (&length, sizeof (int));
172         // Read in the article text
173         msg.Get (articletext, length);
174         articletext[length] = 0;
175         ++fLastNumber;
176
177         printf ("Write article %d (%d bytes)\n",
178                                 fLastNumber, length);
179
180         // Create the new article in the database
181         CArticle a (fLastNumber, articletext);
182         fArticles.push_back (a);
183         this->SendReply (srcid, "Article accepted");
184
185         if (this->FirstInGroup ()) {
186             // The first member of the group sends
187             // the new state to the sinks
188             SendStateToSinks ();
189         }
190     }
191
192     void CArticleDB::Read (PHX_Msg& msg)
193     {
194         int    nb;
195         char articletext[MAXARTICLELEN];
196         char *res;
197         PHX_Id srcid;
198         int phxsize = srcid.XdrSize ();
199         char *placeholder = (char *) malloc (phxsize);
200
201         // Read in the source of the request
202         msg.Get (placeholder, phxsize);
203         srcid.XdrRead (placeholder);
204         // Read the article number from the incoming message
205         msg.Get (&nb, sizeof (int));
206
207         printf ("Read article %d\n", nb);
208
209         // Look for the article in the database
210         list<CArticle>::iterator i = fArticles.begin ();
211         for ( ; i != fArticles.end (); ++i) {
212             if ((*i).Nb () == nb) {
213                 res = (*i).Read ();  // Get contents of article
214                 // Send article back to the requester
215                 this->SendReply (srcid, res);
216                 break;
217             }
218         }
219         if (i == fArticles.end ()) {
220             // Send a not found reply to the requester
221             printf ("Article %d not found!", nb);
222             this->SendReply (srcid, "Article not found");
223         }
224     }
225
226     void CArticleDB::Receive (PHX_Msg& msg,
227                               PHX_Id& srcid, ULONG msgid)
228     {
229         int id;
230         int writerequest = 2;
231         PHX_Msg *mcastmsg;
232
233         printf ("Received a request\n");
234         msg.Get (&id, sizeof (int));
235         switch (id) {
236             case 0:  // It is a write request
237                 mcastmsg = new PHX_Msg (msg.Length ());
238                 mcastmsg->Put (&writerequest, sizeof (int));
239                 mcastmsg->Put (msg.Data (),
240                                msg.Length () - sizeof (int));
241                 // Multicast first the message before managing
242                 Mcast (*mcastmsg, 0, TOTALORDER);
243                 break;
244             case 1:  // It is a read request
245                 Read (msg);
246                 break;
247             case 2:  // The multicast is received and dispatched
248                 Write (msg);
249                 break;
250             default:
251                 printf ("Unknown message type received\n");
252                 break;
253         }
254     }
255
256
257
258     PHX_Msg * CArticleDB::GetState ()
259     {
260         // Put the state of all articles in a Phoenix message
261         int size = 0;
262         char *text;
263
264         for (list<CArticle>::const_iterator i = fArticles.begin ();
265                     i != fArticles.end (); ++i) {
266             size = size + 2 * sizeof (int) + (*i).Length ();
267         }
268         size = size + 2 * sizeof (int);
269
270         PHX_Msg *msg = new PHX_Msg (size);
271         msg->Put (&fLastNumber, sizeof (int));
272         int len = fArticles.size ();
273         msg->Put (&len, sizeof (int));
274         for (list<CArticle>::const_iterator i = fArticles.begin ();
275                 i != fArticles.end (); ++i) {
276             int nb = (*i).Nb ();
277             int len = (*i).Length ();
278             char *text = (*i).Read ();
279             msg->Put (&nb, sizeof (int));
280             msg->Put (&len, sizeof (int));
281             msg->Put (text, strlen (text));
282         }
283         return (msg);
284     }
285
286     void CArticleDB::PutState (PHX_Msg& state)
287     {
288         // If joining back a group, ensure
289         // that article list is empty
290         fArticles.erase (fArticles.begin (), fArticles.end ());
291
292         // Get the state of all articles from the message
```

```
293         state.Get (&fLastNumber, sizeof (int));
294         int len;
295         state.Get (&len, sizeof (int));
296         for (int i = 0; i < len; i++) {
297             int nb;
298             int len;
299             char text[MAXARTICLELEN];
300             state.Get (&nb, sizeof (int));
301             state.Get (&len, sizeof (int));
302             state.Get (&text, len);
303             text[len] = 0;
304             CArticle a (nb, text);
305             fArticles.push_back (a);
306         }
307         printf ("Received new state:\n");
308         this->Dump (stdout);
309     }
310
311     void CArticleDB::SendStateToSinks ()
312     {
313         // Send internal state (i.e. all articles) to sink members
314         PHX_Msg *msg = this->GetState ();
315         this->SendToSinks (*msg);
316         delete msg;
317     }
318
319     void CArticleDB::Dump (FILE *f)
320     {
321         // Dump the internal state of the objects (i.e. all articles)
322         for (list<CArticle>::const_iterator i = fArticles.begin ();
323                 i != fArticles.end (); ++i)
324             fprintf (f, "Article %4d:\n%s\n",
325                             (*i).Nb (), (*i).Read ());
326     }
327
328     int main (int argc, char **argv)
329     {
330         if (argc != 2) {
331             printf ("Usage: %s groupname\n", argv[0]);
332             exit (1);
333         }
334
335         // Store the group name
336         gName = argv[1];
337
338         // Initialize Phoenix
339         PHOENIX::Init ();
340
341         // Create an article database object
342         CArticleDB *articles = new CArticleDB;
343
344         // Call Phoenix main loop
345         PHOENIX::MainLoop ();
346
347         delete articles;
348         // Leave Phoenix cleanly
349         PHOENIX::Exit ();
350
351         printf ("End of program\n");
352         return (0);
353     }
```

C.2.3 News Sink

The news sink application defines (besides the *CArticle* class already introduced in news base application) a class *CNewsSink* (lines 56 to 78) inheriting from the Phoenix *PHX_Sink* class. An object of this class subscribes to the group built by the objects of the news base application.

The *Receive* method (lines 91 to 117) is called each time information is received from one of the group members. As seen in the previous subsection, one of the group members is responsible for sending the whole article data base to the sinks. The information contained in this message is extracted and dumped to the standard output (line 116) using the *Dump* method (lines 151 to 158).

In the *Body* method (lines 123 to 149), the object subscribes to the specified group (line 127), and enters a loop (lines 135 to 138). The loop stops upon user request, leading the object to unsubscribe from the group.

The main procedure is almost identical to the main procedure of the former examples.

newssink.cc

```
1   #include "phoenix.h"
2   #include "debug.h"
3
4   #include <ctype.h>
5
6   #define MAXARTICLELEN 1000
7
8    ////////////////////////////////////////
9    // Type definitions
10   ////////////////////////////////////////
11
12   // A simple class for representing an article
13
14  class CArticle  {
15
16  public:
17   // Constructor & Destructor
18      CArticle ()
19          { fNb = 0; }
20      CArticle (int nb)
21          { fNb = nb; }
22      CArticle (int nb, char *text);
23
24   // Operators
25       // Compare 2 articles for equality
26      bool operator ==(const CArticle & a) const
27          { return (fNb == a.fNb); }
28
29   // Access
30       // Read contents of an article
31      char * Read () const { return (fText); }
32       // Return the article number
33      int Nb () const { return (fNb); }
34       // Return the length of an article
35      int Length () const { return (fLength); }
36
37  private:
38       // Article number
39      int fNb;
40       // Article length
41      int fLength;
42       // Article contents
43      char *fText;
44  } ;
45
46  CArticle::CArticle (int nb, char *text)
47  {
48      fNb = nb;
49      fLength = strlen (text);
50      fText = (char *) malloc (fLength);
51      strcpy (fText, text);
52  }
53
54   // A class for consulting articles
55
56  class CNewsSink : public PHX_Sink  {
57
58  public:
59   // Constructor & Destructor
60      CNewsSink () : PHX_Sink (ACTIVE) {}
61      ~CNewsSink () {}
62
63  protected:
64   // Event handling
65       // Called upon message reception.
66      void Receive (PHX_Msg& msg, PHX_Id& srcid,
67                    ULONG msgid);  // Override
68
69       // Thread associated with object (object is ACTIVE).
70      void Body ();  // Override
71
72       // Dump internal state of objects (i.e. all articles).
73      void Dump (FILE *f);
74
75  private:
76       // List of all article.
77      list<CArticle> fArticles;
78  } ;
79
80   ////////////////////////////////////
81   // Local Scope Variables
82   ////////////////////////////////////
83
84   // Name of group of articles databases.
85  char *gName;
86
87   ////////////////////////////////////
88   // Class Methods
89   ////////////////////////////////////
90
91  void CNewsSink::Receive (PHX_Msg& msg, PHX_Id& srcid,
92                   ULONG msgid)
93  {
94      int dummy;
95
96       // Since receiving state of all articles,
97       // it is necessary to free previous state.
98      fArticles.erase (fArticles.begin (), fArticles.end ());
99
100      // Get state of all articles from the message
101     msg.Get (&dummy, sizeof (int));
102     int len;
103     msg.Get (&len, sizeof (int));
104     for (int i = 0; i < len; i++) {
105         int nb;
106         int len;
107         char text[MAXARTICLELEN];
108         msg.Get (&nb,     sizeof (int));
109         msg.Get (&len,    sizeof (int));
110         msg.Get (&text, len);
111         text[len] = 0;
112         CArticle a (nb, text);
113         fArticles.push_back (a);
114     }
115     printf ("Received new sink information:\n");
116     this->Dump (stdout);
117 }
118
119  ////////////////////////////////////////
120  // Class Methods
121  ////////////////////////////////////////
122
123 void CNewsSink::Body ()
124 {
125      // Subscribe the specified group as a sink.
126     printf ("Subscribe group\n");
127     this->SinkSubscribe (gName);
128
129      // Since performing a blocking operation (reading a
130      // character from the keyboard), detach thread
131      // from the Phoenix scheduling. That way thread
132      // does not block other operations.
133     PHOENIX::AllowConcurrency ();
134     char c;
135     do {
136         printf ("\n(Q)uit\n");
137         cin>> c;
138     } while (toupper (c) != 'Q');
139      // Before calling a Phoenix functiond
140      // reenter the Phoenix scheduling.
141     PHOENIX::DenyConcurrency ();
142
143      // Leave the previously subscribed group.
144     printf ("Unsubscribe group\n");
```

```
145         this->SinkUnsubscribe (gName);
146
147         // Exit Phoenix main loop.
148         PHOENIX::ExitMainLoop ();
149     }
150
151     void CNewsSink::Dump (FILE *f)
152     {
153         // Dump internal state of the objects (i.e. all articles).
154         for (list<CArticle>::const_iterator i = fArticles.begin ();
155                 i != fArticles.end (); ++i)
156             fprintf (f, "Article %4d:\n%s\n",
157                             (*i).Nb (), (*i).Read ());
158     }
159
160     int main (int argc, char **argv)
161     {
162         if (argc != 2) {
163             printf ("Usage: %s groupname\n", argv[0]);
164             exit (1);
165         }
166
167         // Store the group name.
168         gName = argv[1];
169
170         // Initialize Phoenix.
171         PHOENIX::Init ();
172
173         // Create a CNewsSink object.
174         CNewsSink *cp = new CNewsSink;
175
176         // Call Phoenix main loop.
177         PHOENIX::MainLoop ();
178
179         delete cp;
180         // Leave Phoenix cleanly.
181         PHOENIX::Exit ();
182
183         return (0);
184     }
```

Curriculum Vitae

Christoph Malloth was born in St. Moritz (Switzerland) in 1966. He attended primary school in St. Moritz and then the Lyceum Alpinum at Zuoz (Switzerland) from 1979, obtaining his federal maturity in 1986. From 1986 to 1991, he studied at the Swiss Federal Institute of Technology in Zürich (ETHZ, Switzerland) obtaining a diploma as a computer science engineer and the ETHZ award for an exceptional diploma thesis. Since 1991 he has been working in the Operating Systems Laboratory at the Swiss Federal Institute of Technology in Lausanne (EPFL, Switzerland) as an assistant and Ph.D. student, under the supervision of Professor André Schiper, first on parallel, and later on fault-tolerant, distributed systems.