On Building Blocks for Distributed Systems

by

Roberto De Prisco

Dottorato in Applied Mathematics and Computer Science
University of Naples, Italy (1998)

M.S. in Electrical Engineering and Computer Science
Massachusetts Institute of Technology (1997)

Laurea in Computer Science
University of Salerno, Italy (1991)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

December 1999

Author ....
Department of Electrical Engineering and Computer Science
December 10, 1999

Certified by ....
Prof. Nancy Lynch
NEC Professor of Software Science and Engineering
Thesis Supervisor

Accepted by ....
Prof. Arthur C. Smith
Chairman, Department Committee on Graduate Theses

On Building Blocks for Distributed Systems

by

Roberto De Prisco

Submitted to the Department of Electrical Engineering and Computer Science
on December 10, 1999, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

In this thesis we have investigated two building blocks for distributed systems: group communication services and distributed consensus services.

Using group communication services is a successful approach in developing fault tolerant distributed applications. Such services provide communication tools that greatly facilitate the development of applications. Though many existing systems are used in real world applications, there is still the need of providing formal specifications for the group communication services offered by these systems. Great efforts are being made by many researchers to provide such specifications. In this thesis we have tackled this problem and have provided specifications for group communication services. One of our specifications considers the notion of primary view; another one generalizes this notion to that of primary configurations (views with quorums). Both specifications are shown to be implementable. The usefulness of both specifications is demonstrated by applications running on top of them. Our specifications are tailored to dynamic systems, where processes join and leave the system even permanently. We also showed how the approach used to develop the specifications can be applied to transform known algorithms, designed for stating settings, in order to make them adaptable to dynamic systems.

Distributed consensus is the abstraction of many coordination problems, which are of fundamental importance in distributed systems. Distributed consensus has been thoroughly studied and one important result showed that it is not possible to solve consensus in asynchronous systems if failures are allowed. However in such systems it is possible to solve the k-set consensus problem, which is a relaxed version of the consensus problem: each participating process begins the protocol with an input value and by the end of the protocol it must decide on one value so that at most k total values are decided by all correct processes (the classical consensus problem requires that there be a unique value decided by all correct processes). In this thesis we have investigated the k-set consensus problem in asynchronous distributed systems. We extended previous work by exploring several variations of the problem definition and model, including for the first time investigation of Byzantine failures. We showed that the precise definition of the validity requirement, which characterizes what decision values are allowed as a function of the input values and whether failures occur, is crucial to the solvability of the problem. We introduced six validity conditions for this problem (all considered in various contexts in the literature), and we demarcated the line between possible and impossible for each case. In many cases this line is different from the one of the originally studied k-set consensus problem.

Thesis Supervisor: Prof. Nancy Lynch
Title: NEC Professor of Software Science and Engineering

Acknowledgments

I would like to thank my advisor Nancy Lynch for her constant support throughout my years at MIT and her guidance in the development of Part I of this thesis. Nancy has been a wonderful advisor, never putting too much pressure but always checking for progress and suggesting on how to continue. I also thank Alan Fekete and Alex Shvartsman for working with me and Nancy on the research presented in Part I of this thesis.

I would also like to thank Dahlia Malkhi and Michael Reiter for working with me on the research presented in Part II of the thesis. I really enjoyed the collaboration with Dahlia and Mike.

Finally I would like to thank the committee of my thesis defense, Idit Keidar and Butler Lampson (and of course Nancy), for providing insightful comments. Special thanks go to Idit for several discussions and useful suggestions.

This is my fourth thesis: I wrote a thesis for my *Laurea* degree at the University of Salerno, an M.S. thesis here at MIT, a thesis for my *Dottorato* degree at the University of Naples and this Ph.D. thesis ... that's it. I will *never* write another thesis[1].

[1]Nancy's comment: Amen!

Contents

1 Introduction **9**

1.1 Group communication services . . . 10

1.1.1 Overview and related work . . . 10

1.1.2 Work in this thesis . . . 13

1.2 Distributed consensus . . . 17

1.2.1 Overview and related work . . . 17

1.2.2 Work in this thesis . . . 19

1.3 Summary of contributions . . . 19

1.4 Thesis roadmap . . . 20

I Group Communication Services **21**

2 Group Communication Services: Overview **22**

3 Mathematical foundations **25**

3.1 Notation and terminology . . . 25

3.1.1 Sets, functions, sequences . . . 25

3.1.2 Processors, views, configurations, identifiers . . . 26

3.2 The I/O automaton (IOA) model . . . 27

3.2.1 IOA definition . . . 27

3.2.2 Composition of IOA . . . 29

4 The VS service **32**

4.1 The VS service . . . 32

4.2 Variations: The CS and LC services . . . 35

5 The DVS service **37**

5.1 The DVS specification . . . 37

5.2 An implementation of DVS . . . 40

5.2.1 Overview . . . 41

5.2.2 Invariants of DVS-IMPL . . . 43

5.2.3 Proof that DVS-IMPL implements DVS . . . 58

5.3 An application of DVS . . . 65

5.3.1 The TO service . . . 66

5.3.2 The implementation of TO . . . 66

5.3.3 Proof that TO-IMPL implements TO . . . 70

5.4 Remarks . . . 73

6 The DC service **75**

6.1 Overview . . . 75

6.2 The DC specification . . . 76

6.3 Invariants of DC . . . 80

6.4 An implementation of DC . . . 81

6.4.1 Overview . . . 81

6.4.2 Invariants of DC-IMPL . . . 83

6.4.3 Proof that DC-IMPL implements DC . . . 92

6.5 An application of DC . . . 98

6.5.1 Overview . . . 98

6.5.2 Proof that ABD-SYS is an atomic register . . . 101

6.6 Remarks . . . 112

7 Dynamic Algorithms **114**

7.1 The DLC specification . . . 114

7.1.1 Differences with DC . . . 114

7.1.2 Full description of DLC . . . 116

7.1.3 Invariant of DLC . . . 118

7.2 Dynamic PAXOS algorithm . . . 119

7.2.1 Distributed Consensus . . . 119

7.2.2 The original PAXOS algorithm . . . 120

7.2.3 The DPAXOS algorithm . . . 122

7.2.4 Proof of correctness for DPAXOS . . . 126

7.2.5 Remarks . . . 132

7.3 A Replicated Atomic Object Algorithm . . . 132

7.3.1 Description of RAB . . . 133

7.3.2 The code of DLC-TO-RAB . . . 133

7.3.3 Sketch of proof of correctness . . . 139

7.4 Remarks 146

8 Conclusions **148**

II Distributed k-set Consensus **152**

9 Distributed k-set Consensus: Overview **153**

10 The problem **156**

11 Crash failures **158**

11.1 Known results 158

11.2 Impossibilities 158

11.3 Protocols 162

11.4 Remarks 163

12 Byzantine failures **164**

12.1 Impossibilities 164

12.2 Protocols 167

12.3 Remarks 171

13 Conclusions **172**

List of Figures

3-1 An I/O automaton. 28

4-1 The VS service 33

4-2 The CS service 36

5-1 The DVS service. 38

5-2 The DVS-IMPL system. 41

5-3 The VS-TO-DVS$_p$ code. 42

5-4 The abstraction function $\mathcal{F}_{dvs}$. 59

5-5 The TO service. 66

5-6 The DVS-TO-TO$_p$ code. 68

5-7 The DVS-TO-TO$_p$ code (cont'd). 69

5-8 The abstraction function $\mathcal{F}_{to}$. 73

6-1 The DC specification 78

6-2 CS-TO-DC$_p$ 84

6-3 CS-TO-DC$_p$ (transitions cont'd) 85

6-4 The abstraction function $\mathcal{F}_{dc}$. 92

6-5 The ABD interface and state. 99

6-6 The ABD-CODE transitions. 100

7-1 The DLC specification 115

7-2 The DLC-TO-PAXOS code. 124

7-3 The DLC-TO-RAB code. 135

7-4 The DLC-TO-RAB code (cont'd). 136

10-1 Validity conditions. 157

11-1 Crash model: Solvable and impossible 159

12-1 Byzantine model: Solvable and impossible. 165

Chapter 1

Introduction

In the last decade the impact of distributed systems on computing has been tremendous. Nowadays no single workstation is stand-alone; even personal computers in homes are connected with the rest of the computing world by means of the Internet. Computer interconnections can be classified at various levels, depending on the kind of interaction required by the components that are connected. The connection can be as simple as a cable connecting two computers and as complex as the Internet which literally connects millions of computers around the world. The more distributed is the system the more complex is the interconnection. Because of this, distributed systems are more subject to failures than stand-alone computers. Also, distributed systems are harder to program because of the difficulties deriving from sharing data, sharing resources, and coordinating work. Thus developing distributed applications is a complex task.

A popular and successful approach to managing this complexity is to decompose the system design, by constructing the system from pre-defined communication, synchronization, and memory *building blocks*. These building blocks may represent global (that is, system-wide) or local services; they may be combined in parallel or may represent different levels of abstraction. The structure they provide makes the systems easier to build, to use, and to modify. Examples of such building blocks already in use are various types of group membership and group communication services; failure detection, leader election, consensus, and atomic commit services; resource allocation and synchronization services; and various forms of strongly consistent and weakly consistent shared memory.

In this thesis we investigate two such building blocks, namely, *group communication services* and *distributed consensus services*.

1.1 Group communication services

1.1.1 Overview and related work

Recently, *view-oriented group communication* services (see [1] for a survey) have been of particular interest. Such a service allows application processes located at different nodes of a distributed network to operate collectively as a group, using the service to multicast messages to all members of the group. The group communication service is based on a *group membership service*, which provides each group member with a *view* of the group, including a list of the processes that are group members. Messages sent by a process in a view are delivered only to processes that are members of that view, and only when they have that view. Within each view, the service offers guarantees about the order and reliability of message delivery. Thus, a view-oriented group communication service manages both consistent delivery of messages within each view and the reconfiguration involved in changing views. Examples of such services are found in the Isis [19], Transis [6, 33], Ensemble [49], Newtop [38] and Relacs [14] systems as well as in other systems. Typical applications that use view-oriented group communication services include state-machine replication (e.g., [46, 45, 57]), distributed transactions and database replication (e.g., [85, 48, 56]), system management (e.g., [4]) and monitoring (e.g., [3]), video-on-demand servers (e.g., [10, 9]), collaborative computing, such as, distance learning, audio and video conferences, application sharing and even distributed musical "jam sessions" (see [87] for more references).

In order to be most useful group communication services (as well as other building blocks) require clear and precise specifications of their guaranteed behavior. Such specifications would allow application programmers to think carefully about the behavior of systems that use the primitives, without having to understand how the primitives themselves are implemented. Unfortunately, providing appropriate specifications for group communication services is not an easy task. Some of these services are rather complicated, and there is still no agreement about exactly what the guarantees should be. Different specifications arise from different implementations of the same service, because of differences in the safety, performance, or fault-tolerance that is provided. Moreover, the specifications that most accurately describe particular implementations may not be the ones that are easiest for application programmers to use.

Hence providing group communication specifications is a serious challenge, and requires theoretical work to support the system development work. Such work has included formal specification of global membership and communication services (e.g., [17, 23, 25, 27, 34, 72, 81, 83]), design and analysis of distributed algorithms that implement or exploit such services (e.g., [57, 7, 88]), and even impossibility results (e.g., [23]).

The Isis system introduced the important concept of *virtual synchrony* [19]. This concept has been interpreted in various ways, but an essential requirement is that if a particular message is delivered to several processes, then all have the same view of the membership when the message is delivered. This allows the recipients to take coordinated action based on the message, the membership set and the rules prescribed by the application.

The Isis system was designed for an environment where processors might fail and messages might be lost, but where the network does not partition. This assumption might be reasonable for some local area networks, but it is not valid in wide area networks. Therefore, the more recent systems mentioned above allow the possibility that concurrent views of the group might be disjoint.

The first major work on the development of specifications for fault-tolerant group-oriented membership and communication services appears to be that of Ricciardi [81], and the research area is still active (see, e.g., [75, 23, 88, 41]). In particular, there has been a large amount of work on developing specifications for partitionable group services. Some specifications deal just with membership and views [54, 86] while others also cover message services (ordering and reliability properties) [72, 16, 15, 26, 34, 44, 53]. These specifications are all complicated, many are difficult to understand, and some seem to be ambiguous. It is not clear how to tell whether a specification is sufficient for a given application. It is not even clear how to tell whether a specification is implementable at all; impossibility results such as those in [23] demonstrate that this is a significant issue. Vitenberg et al. [87] provide a comprehensive study and comparison of several existing group communication specifications.

Among previous work the specification of a group communication service provided in [41] is particularly relevant to the work done in this thesis. The group communication service specification provided in [41], called VS, captures what seem to be the basic property of a view oriented group communication service: processes are provided with views of the system and communication is view-oriented, meaning that messages sent in a particular view are delivered only within that view. We remark that this is not the only property of existing systems, but it is the most important one. The strength of the VS specification lies in its simplicity. Yet the specification is powerful enough to build applications on top of it.

Providing specifications that are simple enough to be implementable and strong enough to be usable in applications is the key for designing good building blocks. Previous work has shown that this is not an easy task: some specifications are too strong to be implementable, e.g. [23], and some of them fail to capture the non-triviality of existing group communication services. The VS specification has been proven to be implementable and useful as building block for more powerful services. Lesley and Fekete [63] have proved that a version of an algorithm of Cristian and Schmuck [27] implements the VS service. Khazan et al. [58, 59] have used VS in the design of a load-balancing database algorithm and in [41] a totally ordered broadcast service is built on top of VS.

We have mentioned that real distributed systems can partition. When a partition occurs the main problem to be faced is that of maintaining consistency of replicated data. To cope with partitions, usually the application processes perform significant computations only when they have a special type of view called a *primary view*. For example, a replicated database application might only perform a read or write operation within a primary view, in order to ensure that each read receives the result of the last preceding write, in some consistent order of the operations. In this setting, a primary view is typically defined to be one whose membership comprises a *majority* of the universe of processes. The intersection property guaranteed by majorities permits information flow from any previous primary to a newly formed one.

This thesis focuses on primary view group communication services because many real applications do need to maintain consistency of replicated data. However, applications that can tolerate some degree of inconsistency can use partitionable group communication services. For example, in a shared white-board application, a partition would result in users seeing only whatever is written by users in their component. When (possibly) the partition is recovered the white-board can display information written from each component (maybe with some criteria to merge the different white-boards of different components). Another example is a distributed booking system for airline tickets. If the system partitions into two (or more) components each of them could still accept reservations, provided that the airline is willing to face over-booking.

In distributed applications involving replicated data, a well known way to enhance the availability and efficiency of the system is to use *quorums*. A *quorum system* is a set of subsets of the members of the system which satisfy the property that any two sets intersect. We refer to a view with a quorum system defined over the members of the view as a *configuration*. Using configurations an update can be performed with only a quorum available, while with an ordinary view all of the members must be available. The intersection property of quorums permits one to maintain data consistency, within a given configuration. Quorum systems have been extensively studied and used in applications, e.g. [2, 37, 47, 50, 79, 70, 8, 74]. The use of quorums has been proven effective also against Byzantine failures [68, 69].

Pre-defined quorum sets can yield efficient implementations in settings which are relatively static, i.e., failures are transient. However they work less well in settings where processes routinely join and leave the system, or where the system can suffer multiple partitions. For such a setting, a *dynamic* notion of primary is needed. A dynamic notion of primary still needs to maintain some kind of intersection property, in order to permit enough information flow between successive primary views to achieve coherence. For example, each primary view might have to contain at least a majority of the processes in the previous primary view. Several *dynamic voting schemes* have been developed to define primaries adaptively, e.g. [28, 36, 55, 88, 77].

In particular, Yeger Lotem, Keidar, and Dolev [88] have described an implementation of a group membership service that yields only primary views, according to a dynamic notion of primary. An interesting feature of their work is that it points out various subtleties of implementing such a membership service in a distributed manner – subtleties involving different opinions by different processes about what is the previous primary view. These difficulties have led to errors in some of the past work on dynamic voting. The algorithm of [88] copes with these subtleties by maintaining information about a collection of primary views that "might be" the previous primary view. The service deals with group membership only, and not with communication.

1.1.2 Work in this thesis

In this thesis we have provided group communication specifications which handle primary views and primary configurations; the latter required extending the notion of primary view to that of primary configuration. We have proved that the specifications are implementable, by exhibiting algorithms that implement them, and useful as building block for more powerful services, by providing algorithms that implement these more powerful services exploiting the group communication services.

Dynamic views

We have provided a group communication service, called DVS, that integrates the VS group communication service with a dynamic primary view membership service, yielding a dynamic primary view group communication service. The DVS service is inspired by the implementation of [88], but integrates communication with the group membership service.

An important feature of the DVS specification is the careful handling of the interface between the service and the application. When a new view starts, applications generally require some pre-processing, typically, an exchange of information, to prepare for ordinary computation. For example, processes in a coherent database application may need to exchange information about previous updates in order to bring everyone in the new view up to date. We expect each application process to "register" a new view v when it has completed this pre-processing for view v. The DVS service uses registration information when it creates a new view v, in order to determine which previously-created views must satisfy the intersection property with respect to v. When all members have registered v, the application has gathered all information it needs from previous views, and the service no longer needs to ensure intersection in membership between views before v and any subsequent ones that are formed.

Another feature of the DVS specification, compared to that of Yeger Lotem et al. [88], is that our specification is given as an automaton, which maintains state information about the views and the messages sent in each view. This global state can be used in invariants and abstraction functions, leading to assertional proofs of the correctness of implementations of DVS, and also of applications built over DVS. In contrast, Yeger Lotem et al. use a specification given in terms of the whole sequence of events in an execution, and therefore must use operational reasoning about complex sequences of events. Extensive experience with proofs of distributed algorithms suggests that assertional techniques are less error-prone; also they are more amenable to automated checking.

We have demonstrated the value of the DVS specification by showing both how it can be implemented and how it can be used in an application. Both pieces are shown formally, with assertional proofs.

The implementation is a variant of the group membership algorithm of [88]. We have proved that this algorithm implements DVS, in the sense of trace inclusion, that is, the external behavior of the implementation is allowed by the DVS specification. The proof uses a (single-valued) simulation relation and invariant assertions. The key to the proof is an invariant expressing a strong condition about nonempty intersections of views; the proof of this depends on relating a *local* check of *majority* intersection with known views to a *global* check of *nonempty* intersection with existing views.

We have also provided an application algorithm that is a variant of an algorithm in [57, 7, 41], modified to use DVS instead of a static view-oriented service. The modified algorithm uses the registration capability to tell the DVS service that information has been successfully exchanged at the beginning of a new view. We show that it implements a (non-group-oriented) totally-ordered-broadcast service. This proof also uses a simulation relation and invariant assertions.

We have designed our DVS specification to express the guarantees that we think are useful in verifying correctness of applications that use the service. Among previous work, two different sorts of specifications for a primary group service are notable. Work by Ricciardi and others [83] is expressed in terms of temporal logic on consistent cuts; the idea of their specification is that on any cut, there are no disjoint sets of processes such that each set is collectively aware of no members outside that set. Yeger Lotem et al. [88] use a property of an execution, which was previously defined by Cristian [26] for majority groups: any two primary views are linked by a chain of views where every consecutive pair of views includes a process that “knows” it belongs to both views. As far as we know, these previous specifications have not been used to verify properties of applications running above them.

The DVS specification omits some properties of existing dynamic primary view management algorithms. For example, Isis [19] guarantees that processes that move together from one view to the next receive exactly the same messages in the first view. Guaranteeing this property requires state exchange within the view management service. This property is not needed to verify properties of other applications, such as the totally-ordered broadcast service of [41]. Also, our service provides no explicit support for application-level state exchange. Real systems, e.g. Isis, do provide such support, by allowing application-level state exchange messages to be piggy-backed on the lower-level state exchange messages.

Dynamic configurations

Quorum-based methods for managing replicated data are popular because they provide availability of both reads and writes in the presence of faulty behavior by some sites or communication links. A quorum system is also called a configuration. If a system lasts for a very long time, it may become necessary to alter the configuration, perhaps because some sites have failed permanently and others have joined the system, or perhaps because users want a different trade-off between read-availability and write-availability. For example, if more sites join the system, these sites must be included in the quorums in order to use them; If many sites fail permanently, these sites must be taken out of the quorums in order to maintain availability. The most common proposal has been to use a two-phase commit protocol which stops all application operations while all sites are notified of the new configuration. Since two-phase commit is a blocking protocol, this solution is vulnerable to a single failure during the configuration change. An alternative proposal in [66] has reconfiguration directed by a single site, thus this is also not fault-tolerant. In a setting of database transactions, [47] showed how to integrate fault-tolerant updates of replicated information about quorum sizes (using the same quorums for both data item replicas and quorum information replicas).

Herlihy [51] provides algorithms to shrink and enlarge quorums within a static universe of processors; the setting considered in [51] does not allow processors to join and leave the system. Lamport discusses how to modify his PAXOS algorithm [61] in order allow processors to join and leave the system. In this thesis we integrate these aspects in a group communication framework.

There are subtle issues that arise in managing the change of configurations, including how to make sure that any operation using the new configuration is aware of all information from operations that used an old configuration, and how to allow concurrent attempts to alter the configuration.

In this thesis we have addressed this problem by extending dynamic primary views group communication services to handle configurations. The main difficulty in combining configurations with the notion of dynamic primary view is the intersection property required to maintain consistency among data stored at different sites. A dynamic primary view must intersect the previous one in at least a quorum of processors (this property is required, for example, by replicated data applications in order to keep all the replicas consistent). With configurations this intersection property that works for primary views, is no longer enough. Indeed updated information might be only at a quorum and the processors in the intersection might be not in that quorum. A stronger intersection property is required. We have proposed one possible intersection property that allows applications to keep data consistency across changes of primary configurations. Namely, we require that there be a quorum of the old primary configuration which is included in the membership set of the new primary configuration. This guarantees that there is at least one process in the new primary configuration that has the most up to date information. This, similarly to the intersection property of dynamic primary views, allows flow of information from the old configuration to the new one and thus permits one to preserve data consistency.

We actually considered a more specialized version of configurations which uses two sets of quorums, a set of *read* quorums and a set of *write* quorums, with the property that any read quorum intersects any write quorum. (This choice is justified by the application we develop, an atomic read/write register.) With this kind of configuration the intersection property that we require for a new primary configuration is that there be one read quorum and one write quorum both of which are included in the membership set of the new primary configuration. The use of read and write quorums (as opposed to just quorums) can be more efficient in order to balance the load of the system (e.g., [37]).

The resulting dynamic primary configuration group communication service is called DC. This service also integrates support for state exchange into the DC specification. This improves the modularity of the building block. When a new configuration starts, applications generally require some pre-processing, such as an exchange of information, to prepare for ordinary computation. Typically this is needed in order to bring every member of the configuration up to date. For example, processes in a coherent database application may need to exchange information about previous updates in order to bring everyone in the new configuration up to date. We will refer to the up-to-date state of a new configuration as the *starting state* of that configuration. The starting state is the state of the computation that all members should have in order to perform regular computation. The computation of the starting state should be offered by the communication service so that applications do not have to worry about the details of the underlying state exchange. We have demonstrates the value of the DC specification by showing both an algorithm that implements DC and how DC can be used in an application. The implementation is based on a variant of the group membership algorithm of [88]. The application is an atomic read/write shared register, and is similar to the work of of Attiya, Bar-Noy and Dolev [12] and of Lynch and Shvartsman [66].

Dynamic algorithms

We have investigated the use of the technique introduced to design the DVS and DC services to transform services and applications that are designed for "static" settings, into ones that work well in "dynamic" settings.

We used a variant of the DC service to provide a dynamic version of the Lamport's PAXOS algorithm [61]. The PAXOS algorithm solves a fundamental problem of distributed computing: the consensus problem. In such a problem processors of a distributed system start computation with an input value and have to make an irreversible decision guaranteeing *agreement*, which requires that all decisions are the same, and *validity*, which requires that any decision is equal to some input value.

The PAXOS algorithm tolerates many types of failures: timing failures, loss, duplication and reordering of messages and process stopping failures. Process recoveries are considered; some stable storage is needed. PAXOS is guaranteed to work safely, that is, to satisfy agreement and validity, regardless of process, channel and timing failures and process recoveries. When the distributed system stabilizes, meaning that there are no failures nor process recoveries and a majority of the processes are not stopped, for a sufficiently long time, termination is achieved; the performance of the algorithm when the system stabilizes is good.

The original algorithm is designed for static settings, where failures are transient, that is, failed processors recover. If a majority (or a quorum) of the processors is not available the system is blocked. If such a majority or quorum permanently leaves the system, then the system is blocked forever. The variant we have designed adapts well to permanent changes of the underlying distributed system.

The PAXOS algorithm bears many similarities with an earlier algorithm of Liskov and Oki [76]. The work of Liskov and Oki uses a notion of "view" which changes when a new primary site needs to be selected. The notion of "view" and that of "view synchrony" has later been proven very successful (see the overview and related work of this section).

We have also provided a dynamic primary copy data replication algorithm. As the dynamic version of PAXOS also this algorithm is based on a variant of the DC service. This algorithm uses a centralized approach in which a "leader" process is responsible for providing responses to client's queries. In order to keep consistency this leader process replicates (part of) its own state to a quorum of processes. The algorithm exploits the quorum-oriented framework provided by the DC service. We sketch the proof of correctness of this algorithm; the technique used to prove correct other applications developed in this thesis should apply also to this algorithm.

1.2 Distributed consensus

1.2.1 Overview and related work

Another important building block for distributed systems is *distributed consensus*. Such a problem arises in many forms and various contexts, such as, for example, distributed data replication, distributed databases, flight control systems. Data replication is used in practice to provide high availability: having more than one copy of the data allows easier access to the data, i.e., the nearest copy of the data can be used. However, consistency among the copies must be maintained. A consensus algorithm can be used to maintain consistency. A practical example of the use of data replication is an airline reservation system. The data consists of the current booking information for the flights and it can be replicated at agencies spread over the world. The current booking information can be accessed at any of the replicas. Reservations or cancellations must be agreed upon by all the copies.

In a distributed database, the consensus problem arises when a collection of processes participating in the processing of a distributed transaction has to agree on whether to commit or abort the transaction, that is, make the changes due to the transaction permanent or discard the changes. A common decision must be taken to avoid inconsistencies. A practical example of the use of distributed transactions is a banking system. Transactions can be done at any bank location or ATM machine, and the commitment or abortion of each transaction must be agreed upon by all the bank locations or ATM machines involved.
In a flight control system, the consensus problem arises when the flight surface and airplane control systems have to agree on whether to continue or abort a landing in progress or when the control systems of two approaching airplanes need to modify the air routes to avoid collision.

Distributed consensus has been extensively studied; a good survey of early results is provided in [42]. We refer the reader to [65] for a more up-to-date treatment of consensus problems.

One of the most celebrated result about distributed consensus is the impossibility result of Fischer, Lynch and Paterson [43]. This impossibility result, popularly known as FLP, states that it is impossible to achieve distributed consensus in asynchronous systems even if only one stop failures is possible. This surprising result sparked various directions of research aimed to solve the problem by either restricting the asynchrony of the computation model (e.g. [31, 35]) or using randomized protocols (e.g. [18, 21, 80]) or weakening the problem definition (e.g. [24, 32, 39, 40]).

The last of these three directions of research falls in the more general research area of demarcating what is deterministically computable and what is deterministically impossible in asynchronous distributed systems in the presence of failures. The FLP impossibility seemed to suggest that no nontrivial problem could be solved deterministically and asynchronously in the presence of faults. Attiya, Bar-Noy, Dolev, Peleg and Reischuk [13] showed that the renaming problem can be solved in a deterministic way in asynchronous system in the presence of failures. Informally, in the renaming problem processors start the computation with a "name" taken from some unbounded ordered name space and have to "rename" themselves with names chosen from a new small name space. This result revived the research trend of exploring computable and impossible in deterministic asynchronous distributed systems subject to failures. Following this direction, Chaudhuri [24] defined the k-consensus problem, which is a natural generalization of the consensus problem obtained by allowing processes to decide on k different values, instead of requiring them to agree on a single value. The 1-consensus problem is the classical consensus problem.

Chaudhuri provided an algorithm to solve the k-consensus problem that tolerates up to a threshold t of process failures strictly smaller than k. This result proved that the k-consensus problem, for $k \geq 2$, allows more resilience than the 1-consensus problem. Chaudhuri conjectured that the k-consensus problem was impossible to solve while tolerating k or more failures. This conjecture was proven true by three independent research teams: Borowsky and Gafni [20], Herlihy and Shavit [52] and Saks and Zaharoglou [84]. Attiya [11] provided an alternative proof of the same result.

The results of [24, 20, 52, 84] completely characterize the k-consensus problem in asynchronous systems with stop failures. In such a model the k-consensus problem is solvable if and only if $t < k$.

The formal definition of the k-consensus problem requires three conditions to be satisfied: *agreement*, *termination* and *validity*. The agreement condition requires that each process decide on a value in such a way that the set of decided values has cardinality at most k. The termination condition simply requires that each (correct) process decide. For what concern the validity condition, several variants have been considered in the literature. The validity condition used in [24, 20, 52, 84] requires that each of the decision be equal to some input value.

An alternative definition of the validity condition considered for the 1-consensus problem with stop failures requires that if all the inputs to the processes of the systems are equal then any decision must be equal to the input (see, for example, Chapter 6 of [65]).

In a Byzantine environment faulty processes can "mask" their inputs. Hence a more suitable validity condition considered for the 1-consensus problem with Byzantine failures requires that if all the correct processes have the same input then any decision be the input of a correct process [62, 78].

1.2.2 Work in this thesis

In this thesis we have explored several alternative validity conditions and we consider the k-consensus problem in asynchronous systems both with stop failures and with Byzantine failures. We have considered six different definitions for the validity condition of the k-consensus problem. In many cases the validity condition makes a difference. We have considered the six variations of the k-consensus problem both in the stop failure case and in the Byzantine failure case. This lead to twelve different problems. One of this is the k-consensus problem considered in [24, 20, 52, 84]. Hence for this problem we already know the line that separate solvable from impossible (the problem is solvable if and only if the number of allowed failures is strictly less than k). For the other variations of the problem and in particular for the Byzantine settings, the line between impossible and possible was not known. We have demarcated these lines.

1.3 Summary of contributions

This thesis provided new formal specifications for group communication services. The specifications are shown to be implementable and useful to build applications. The significance of this work is two-fold: on one hand it is a contribution in the identification of useful formal specifications for group communication services, a research area very active recently; on the other hand we have explored the possibility of integrating into a single group communication building block the notion of primary view and that of configuration, both of which are well known but never have been used together. The specifications we have provided are tailored to dynamic systems, where processors join and leave the system routinely and possibly permanently. The approach used to design such a dynamic services has been applied also to transform known algorithms, designed for stating settings, in order to make them adaptable to dynamic systems.

This thesis investigated also some theoretical aspects of another important building block for distributed systems: distributed consensus. We extended previous work by exploring several variations of the problem definition and model, including for the first time investigation of Byzantine failures. We showed that the precise definition of the validity requirement, which characterizes what decision values are allowed as a function of the input values and whether failures occur, is crucial to the solvability of the problem. We introduced six validity conditions for this problem (all considered in various contexts in the literature), and we demarcated the line between possible and impossible for each case. In many cases this line is different from the one of the originally studied k-set consensus problem.

1.4 Thesis roadmap

The rest of thesis is divided into two parts. The first part is dedicated to group communication services while the second part studies the consensus problem.

Part I (group communication services) is structured as follows. Chapter 2 contains an overview of group communication services. Chapter 3 contains notation and terminology used throughout the rest of the part and introduces the I/O automaton model, which is used to provide the specifications, the implementations and the applications. Chapter 4 describes the VS service of [41]; such a service is used as building block for the implementations of the group communication services provided in this thesis. Chapter 5 contains the DVS specifications, a specification for a dynamic primary view group communication service, together with an implementation and a totally ordered broadcast service running on top of DVS. Chapter 6 contains the DC specifications, a specification for a dynamic primary configuration group communication service, together with an atomic read/write register implemented top of DC. Chapter 7 provides a version of Lamport's PAXOS algorithm implemented on top of a variation of the DC service. Finally Chapter 8 provides concluding remarks for Part I.

Part II (distributed consensus) is structured as follows. Chapter 9 contains an introduction to the problem. Chapter 10 describes the model of computation and provides a formal definition of the problem. Chapters 11 and 12 study the k-set consensus problem in the crash failures and Byzantine failures models, respectively. Finally, Chapter 13 provides concluding remarks for Part II.

Part I

Group Communication Services

Chapter 2

Group Communication Services: Overview

Developing distributed applications is a difficult task, because of the complexities of the applications themselves and of the fault-prone distributed settings in which they run. Considerable effort is devoted to making distributed applications robust in the face of typical processor and communication failures. A successful approach to overcome these difficulties is to modularize the system by implementing suitable building blocks that provide powerful general-purpose distributed computation services.

Among the most important examples of building blocks are *group communication services.* Group communication services enable processes located at different nodes of a distributed network to operate collectively as a group. The processes do this by using a group communication service to multicast messages to all members of the group. Different group communication services offer different guarantees about the order and reliability of message delivery. Examples are found in Isis [19], Transis [33], Totem [73], Newtop [38], Relacs [14] and Horus [86].

The basis of a group communication service is a *group membership service.* Each process, at each time, has a unique *view* of the membership of the group. The view includes a list of the processes that are members of the group. Views can change from time to time, and may become different at different processes. Isis introduced the important concept of *virtual synchrony* [19]. This concept has been interpreted in various ways, but an essential requirement is that if a particular message is delivered to several processes, then all have the same view of the membership when the message is delivered which is also the view where the message was sent. This allows the recipients to take coordinated action based on the message, the membership set and, obviously, the application.

To be most useful to application programmers, system building blocks should come equipped with simple and precise specifications of their guaranteed behavior. Such specifications would allow application programmers to think carefully about the behavior of systems that use the primitives, without having to understand how the primitives themselves are implemented. Unfortunately, providing appropriate specifications for group communication services is not an easy task. Some of these services are rather complicated, and there is still no agreement about exactly what the guarantees should be. Different specifications arise from different implementations of the same service, because of differences in the safety, performance, or fault-tolerance that is provided. Moreover, the specifications that most accurately describe particular implementations may not be the ones that are easiest for application programmers to use. Example of specifications for group membership and communication services can be found in [17, 23, 25, 27, 34, 72, 81, 83]).

In distributed application involving replicated data, a well known way to enhance the availability and efficiency of the system is to use *quorums*. A quorum system is a set of subsets of the members of the system which satisfy the property that any two sets intersect. We refer to a view with a quorum system as a *configuration*. Using configurations an update can be performed with only a quorum available, while with an ordinary view all of the members must be available. The intersection property of quorums guarantees consistency within a given configuration. Quorum systems have been extensively studied and used in applications (e.g., [2, 37, 47, 50, 70, 74]).

Pre-defined quorum sets can yield efficient implementations in settings where the system is relatively static, that is, failures are transient. However, they work less well in settings where the set of processors in the network evolves over time, with processes joining and leaving the system. For such a setting, a *dynamic* notion of primary is needed. A dynamic notion of primary still needs to maintain some kind of intersection property, in order to permit enough information flow between successive primary views to achieve coherence. For example, each primary view might have to contain at least a majority of the processes in the previous primary view. Several *dynamic voting schemes* have been developed to define primaries adaptively, e.g. [28, 36, 55, 88, 71, 77].

In particular, Yeger Lotem, Keidar, and Dolev [88] have described an implementation of a group membership service that yields only primary views, according to a dynamic notion of primary. An interesting feature of their work is that it points out various subtleties of implementing such a membership service in a distributed manner – subtleties involving different opinions by different processes about what is the previous primary view. These difficulties have led to errors in some of the past work on dynamic voting. The algorithm of [88] copes with these subtleties by maintaining information about a collection of primary views that "might be" the previous primary view. The service deals with group membership only, and not with communication. Yeger Lotem et al. prove that their protocol satisfies the following condition on system executions: any two (primary) views that occur in an execution are linked by a chain of views where for every consecutive pair of views in the chain, there is some process that "knows" it belongs to both views.

In Chapter 5 we provide a group communication service, called DVS, that integrates the VS group communication service with a dynamic primary view membership service, yielding a dynamic primary view group communication service. The DVS service is inspired by the implementation of [88], but integrates communication with the group membership service. We also show how the DVS specification can be implemented and used for an application.

In Chapter 6 we extend the notion of "primary view" to that of "primary configuration". The main difficulty in making this step is to identify the intersection property between two successive primary configurations that allows to maintain consistency. We propose one possible such a property. Namely, we require that there be a quorum of the old primary configuration which is included in the membership set of the new primary configuration. This guarantees that there is at least one process in the new primary configuration that has the most up to date information. This, similarly to the intersection property of dynamic primary views, allows flow of information from the old configuration to the new one and thus permits to preserve consistency.

We actually consider a more specialized version of configurations which uses two sets of quorums, a set of *read* quorums and a set of *write* quorums, with the property that any read quorum intersects any write quorum. (This choice is justified by the application we develop, an atomic read/write register.) With this kind of configuration the intersection property that we require for a new primary configuration is that there be one read quorum and one write quorum both of which are included in the membership set of the new primary configuration. The use of read and write quorums (as opposed to just quorums) can be more efficient in order to balance the load of the system (e.g., [37]).

We provide a a group communication service, called DC, that integrates a group communication service with a dynamic primary configuration membership service. We prove that the DC service is implementable and can be used for applications.

Finally, in Chapter 7, we investigate the use of the technique introduced to design DVS and DC to transform services and applications that are designed for "static" settings, into ones that work well in "dynamic" settings. Specifically, we design a service similar to DC and we use that service to provide a dynamic version of the Lamport's PAXOS algorithm [61]. The original algorithm is designed for system that are relatively static: if a majority (o more generally a quorum) of the processors is not available then the algorithm blocks. The dynamic version adapts well to permanent changes of the system. We also design a primary copy data replication algorithm; this algorithm is simlar to the Liskov-Oki algorithm [76] but considers dynamic settings, while the Liskov-Oki is designed for static settings.

Chapter 3

Mathematical foundations

In this chapter we introduce some terminology and notation, and then we provide the underlying formal model used to specify our group communication services and applications. Section 3.1 provides terminology and notation and Section 3.2 describes the IOA model.

3.1 Notation and terminology

3.1.1 Sets, functions, sequences

We write λ for the empty sequence. If a is a sequence then $|a|$ denotes the length of a. If a is a sequence and $1 \leq i \leq j \leq |a|$ then $a(i)$ denotes the ith element of a and $a(i..j)$ denotes the subsequence $a(i), a(i+1), ..., a(j)$ of a. The *head* of a nonempty sequence a is $a(1)$. A sequence can be used as a queue: the *append* operation modifies the sequence by concatenating it with a new element and the *remove* operation modifies the sequence by deleting its head.

If a and b are sequences, a finite, then $a \circ b$ denotes the concatenation of a and b. We sometimes abuse this notation by letting a or b be a single element. We say that sequence a is a *prefix* of sequence b, written $a \leq b$, provided that there exists c such that $a \circ c = b$. A collection A of sequences is *consistent* provided that $a \leq b$ or $b \leq a$ for all $a, b \in A$. If A is a consistent collection of sequences, we define $lub(A)$ to be the minimum sequence b such that $a \leq b$ for all $a \in A$.

If S is a set, then $seqof(S)$ denotes the set of all finite sequences of elements of S. If $a \in seqof(S)$ and f is a partial function from S to T whose domain includes the set of all elements of S appearing in a, then $applytoall(f, a)$ denotes the sequence b such that $length(b) = length(a)$ and, for $i \leq length(b)$, $b(i) = f(a(i))$.

If S is a set, the notation $S_\perp$ refers to the set $S \cup \{\perp\}$. Whenever S is ordered, we order $S_\perp$ by extending the order on S, and making $\perp$ less than all elements of S. If R is a binary relation, then we define $dom(R)$, the *domain* of R, to be the set (without repetitions), of first elements of the ordered pairs comprising relation R. If f is a partial function from S to T, and $\langle s,t\rangle \in S \times T$, then $f \oplus \langle s,t\rangle$ is defined to be the partial function that is identical to f except that $f(s) = t$.

We denote by *arrayof* (S) the set of all arrays, indexed by positive integers, whose entries consists of elements of $S_\perp$.

3.1.2 Processors, views, configurations, identifiers

$\mathcal{P}$ denotes the universe of all processors,[1] and $\mathcal{M}$ the universe of all possible messages. $\mathcal{G}$ is a totally ordered set of identifiers used to distinguish views or configurations, with a distinguished least element g_0.

A *view* $v = \langle g, P\rangle$ consists of a view identifier $g \in \mathcal{G}$ and a nonempty membership set $P \subseteq \mathcal{P}$; we write $v.id$ and $v.set$ to denote the view identifier and membership set components of v, respectively. $\mathcal{V}$ denotes the set of all views, and $v_0 = \langle g_0, P_0\rangle$ is a distinguished *initial view*.

The notion of view can be generalized to that of *configuration*. A configuration is a view with a structure defined on the view. For example a configuration can be a view with a set of quorums defined over the memebrship set of the view. However configurations can be tailored to applications. For example, applications that use read and write quorums, use configurations which are views with a set of read quorums and a set of write quorums; applications that use a "leader" processor use configurations with a leader processor. Next we define several types of configurations. We will specify the type of configuration we use in the chapter where we use it.

A configuration is a triple, $c = \langle g, P, \mathcal{Q}\rangle$, where $g \in \mathcal{G}$ is a unique identifier, $P \subseteq \mathcal{P}$ is a nonempty set of processors, and $\mathcal{Q}$ is a nonempty sets of nonempty subsets of P, such that any two subsets intersects. Each element of $\mathcal{Q}$ is called a *quorum* of c.

A more specialized type of configuration is a quadruple, $c = \langle g, P, \mathcal{R}, \mathcal{W}\rangle$, where $g \in \mathcal{G}$ is a unique identifier, $P \subseteq \mathcal{P}$ is a nonempty set of processors, and $\mathcal{R}$ and $\mathcal{W}$ are nonempty sets of nonempty subsets of P, such that $R \cap W \neq \emptyset$ for all $R \in \mathcal{R}$, $W \in \mathcal{W}$. Each element of $\mathcal{R}$ is called a *read quorum* of c, and each element of $\mathcal{W}$ a *write quorum*. We refer to this type of configuration as read-write quorum configuration.

Another type of configuration is a quadruple, $c = \langle g, P, \mathcal{Q}, p\rangle$, where $g \in \mathcal{G}$ is a unique identifier, $P \subseteq \mathcal{P}$ is a nonempty set of processors, and $\mathcal{Q}$ is a quorum system and $p \in P$ is a distinguished processor, called the leader of the configuration. We refer to this type of configuration as leader configuration.

Once fixed a particular type of configuration, welet $\mathcal{C}$ denote the set of all configurations. Given a configuration c, the notation $c.id$ refers to the configuration identifier g, the notation $c.set$ refers to the membership set P; the notation $c.qrms$ refers to the quorum system $\mathcal{Q}$ while $c.rqrms$ and

[1] We use "processor" and "process" interchangeably.

$c.wqrms$ refer to the read quorums set $\mathcal{R}$ and the write quorum sets $\mathcal{W}$, respectively; the notation $c.ldr$ refers to the leader p of configuration c.

We distinguish an initial configuration $c_0 = \langle g_0, P_0, \mathcal{R}_0, \mathcal{W}_0 \rangle$ (or $c_0 = \langle g_0, P_0, \mathcal{Q}_0 \rangle$, or $c_0 = \langle g_0, P_0, \mathcal{Q}_0, p_0 \rangle$, depending on the type of configuration that we are using), where g_0 is a distinguished configuration identifier.

3.2 The I/O automaton (IOA) model

We describe our services and algorithms using the I/O automaton model of Lynch and Tuttle [67] (without fairness).

The I/O automata (IOA for short) model is a formal model suitable for describing asynchronous distributed systems. The basic I/O automaton model was introduced by Lynch and Tuttle [67].

Various extensions of the basic model have been developed. For example two extensions provide formal mechanisms to handle the passage of time and thus are suitable for describing partially synchronous distributed systems; these models are the MMT automaton (MMTA for short) and the general timed automaton (GT automaton or GTA for short). The MMTA is a special case of GTA, and thus it can be regarded as a notation for describing some GT automata.

For the purpose of this thesis, we will use this basic I/O automaton model, which we describe in Section 3.2.1. Section 3.2.2 describes the "composition" operation on automata. The interested reader can find more information about IOA in [65].

3.2.1 IOA definition

An I/O automaton is a simple type of state machine in which transitions are associated with named *actions*. These actions are classified into categories, namely *input, output, internal* and, for the timed models, *time-passage*. Input and output actions are used for communication with the external environment, while internal actions are local to the automaton. The time-passage actions are intended to model the passage of time. The input actions are assumed not to be under the control of the automaton, that is, they are controlled by the external environment which can force the automaton to execute the input actions. Internal and output actions are controlled by the automaton. The time-passage actions are also controlled by the automaton (though this may at first seem somewhat strange, it is just a formal way of modeling the fact that the automaton must perform some action before some amount of time elapses).

As an example, we can consider an I/O automaton that models the behavior of a process involved in a consensus problem. Figure 3-1 shows the typical interface (that is, input and output actions) of such an automaton. The automaton is drawn as a circle, input actions are depicted as incoming arrows and output actions as outcoming arrows (internal actions are hidden since they are local

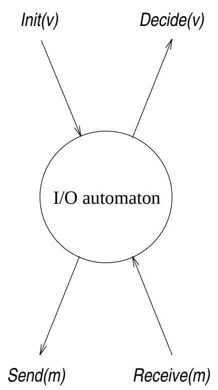

Figure 3-1: An I/O automaton.

to the automaton). The automaton receives inputs from the external world by means of action INIT(v), which represents the receipt of an input value v and conveys outputs by means of action DECIDE(v) which represents a decision of v. Actions SEND(m) and RECEIVE(m) are supposed to model the communication with other automata.

A *signature* S is a triple consisting of three disjoint sets of actions: the input actions, $in(S)$, the output actions, $out(S)$, and the internal actions, $int(S)$. The external actions, $ext(S)$, are $in(S) \cup out(S)$; the locally controlled actions, $local(S)$, are $out(S) \cup int(S)$; and $acts(S)$ consists of all the actions of S. The external signature, $extsig(S)$, is defined to be the signature $(in(S), out(S), \emptyset)$. The external signature is also referred to as the external interface.

An *I/O automaton* (IOA for short) A, consists of five components:

- $sig(A)$, a signature
- $states(A)$, a (not necessarily finite) set of *states*
- $start(A)$, a nonempty subset of $states(A)$ known as the *start states* or *initial states*
- $trans(A)$, a *state-transition relation*, where $trans(A) \subseteq states(A) \times acts(sig(A)) \times states(A)$; this must have the property that for every state s and every input action π, there is a transition $(s, \pi, s') \in trans(A)$
- $tasks(A)$, a *task partition*, which is an equivalence relation on $local(sig(A))$ having at most countably many equivalence classes

Often $acts(A)$ is used as shorthand for $acts(sig(A))$, and similarly $in(A)$, and so on.

An element (s, π, s') of $trans(A)$ is called a *transition*, or *step*, of A. If for a particular state s and action π, A has some transition of the form (s, π, s'), then we say that π is *enabled* in s. Input actions are enabled in every state.

The fifth component of the I/O automaton definition, the task partition $tasks(A)$, should be thought of as an abstract description of "tasks," or "threads of control," within the automaton. This partition is used to define fairness conditions on an execution of the automaton; roughly speaking, the fairness conditions say that the automaton must continue, during its execution, to give fair turns to each of its tasks.

An *execution fragment* of A is either a finite sequence, $s_0, \pi_1, s_1, \pi_2, \ldots, \pi_r, s_r$, or an infinite sequence, $s_0, \pi_1, s_1, \pi_2, \ldots, \pi_r, s_r, \ldots$, of alternating states and actions of A such that $(s_k, \pi_{k+1}, s_{k+1})$ is a transition of A for every $k \geq 0$. Note that if the sequence is finite, it must end with a state. An execution fragment beginning with a start state is called an *execution*. The *length* of a finite execution fragment $\alpha = s_0, \pi_1, s_1, \pi_2, \ldots, \pi_r, s_r$ is r. The set of executions of A is denoted by $execs(A)$. A state is said to be *reachable* in A if it is the final state of a finite execution of A.

The *trace* of an execution α of A, denoted by $trace(\alpha)$, is the subsequence of α consisting of all the external actions. A *trace* β of A is a trace β of an execution of A. The set of traces of A is denoted by $traces(A)$.

3.2.2 Composition of IOA

The composition operation allows an automaton representing a complex system to be constructed by composing automata representing simpler system components. The most important characteristic of the composition of automata is that properties of isolated system components still hold when those isolated components are composed with other components. The composition identifies actions with the same name in different component automata. When any component automaton performs a step involving π, so do all component automata that have π in their signatures. Since internal actions of an automaton A are intended to be unobservable by any other automaton B, automaton A cannot be composed with automaton B unless the internal actions of A are disjoint from the actions of B. (Otherwise, A's performance of an internal action could force B to take a step.) Moreover, A and B cannot be composed unless the sets of output actions of A and B are disjoint. (Otherwise two automata would have the control of an output action.)

Let I be an arbitrary finite index set[2]. A finite countable collection $\{S_i\}_{i \in I}$ of signatures is said to be *compatible* if for all $i, j \in I$, $i \neq j$, the following hold[3]:

1. $int(S_i) \cap acts(S_j) = \emptyset$

2. $out(S_i) \cap out(S_j) = \emptyset$

A finite collection of automata is said to be *compatible* if their signatures are compatible.

[2]The composition operation for IOA is defined also for an infinite but countable collection of automata [65], but we only consider the composition of a finite number of automata.

[3]We remark that for the composition of an infinite countable collection of automata, there is a third condition on the definition of compatible signature [65]. However this third condition is automatically satisfied when considering only finite sets of automata.

When we compose a collection of automata, output actions of the components become output actions of the composition, internal actions of the components become internal actions of the composition, and actions that are inputs to some components but outputs of none become input actions of the composition. Formally, the *composition* $S = \prod_{i \in I} S_i$ of a finite compatible collection of signatures $\{S_i\}_{i \in I}$ is defined to be the signature with

- $out(S) = \cup_{i \in I} out(S_i)$
- $int(S) = \cup_{i \in I} int(S_i)$
- $in(S) = \cup_{i \in I} in(S_i) - \cup_{i \in I} out(S_i)$

The *composition* $A = \prod_{i \in I} A_i$ of a finite collection of automata, is defined as follows:[4]

- $sig(A) = \prod_{i \in I} sig(A_i)$
- $states(A) = \prod_{i \in I} states(A_i)$
- $start(A) = \prod_{i \in I} start(A_i)$
- $trans(A)$ is the set of triples (s, π, s') such that, for all $i \in I$, if $\pi \in acts(A_i)$, then $(s_i, \pi, s'_i) \in trans(A_i)$; otherwise $s_i = s'_i$
- $tasks(A) = \cup_{i \in I} tasks(A_i)$

Thus, the states and start states of the composition automaton are vectors of states and start states, respectively, of the component automata. The transitions of the composition are obtained by allowing all the component automata that have a particular action π in their signature to participate simultaneously in steps involving π, while all the other component automata do nothing. The task partition of the composition's locally controlled actions is formed by taking the union of the components' task partitions; that is, each equivalence class of each component automaton becomes an equivalence class of the composition. This means that the task structure of individual components is preserved when the components are composed. Notice that since the automata A_i are input-enabled, so is their composition. The following theorem follows from the definition of composition.

Theorem 3.2.1 *The composition of a compatible collection of I/O automata is an I/O automaton.*

The following theorems relate the executions and traces of a composition to those of the component automata. The first says that an execution or trace of a composition "projects" to yield executions or traces of the component automata. Given an execution, $\alpha = s_0, \pi_1, s_1, \ldots,$ of A, let

[4]The Π notation in the definition of $start(A)$ and $states(A)$ refers to the ordinary Cartesian product, while the Π notation in the definition of $sig(A)$ refers to the composition operation just defined, for signatures. Also, the notation s_i denotes the ith component of the state vector s.

$\alpha|A_i$ be the sequence obtained by deleting each pair π_r, s_r for which π_r is not an action of A_i and replacing each remaining s_r by $(s_r)_i$, that is, automaton A_i's piece of the state s_r. Also, given a trace β of A (or, more generally, any sequence of actions), let $\beta|A_i$ be the subsequence of β consisting of all the actions of A_i in β. Also, $|$ represents the subsequence of a sequence β of actions consisting of all the actions in a given set in β.

Theorem 3.2.2 *Let $\{A_i\}_{i \in I}$ be a compatible collection of automata and let $A = \prod_{i \in I} A_i$.*

1. *If $\alpha \in execs(A)$, then $\alpha|A_i \in execs(A_i)$ for every $i \in I$.*

2. *If $\beta \in traces(A)$, then $\beta|A_i \in traces(A_i)$ for every $i \in I$.*

The other two are converses of Theorem 3.2.2. The next theorem says that, under certain conditions, executions of component automata can be "pasted together" to form an execution of the composition.

Theorem 3.2.3 *Let $\{A_i\}_{i \in I}$ be a compatible collection of automata and let $A = \prod_{i \in I} A_i$. Suppose α_i is an execution of A_i for every $i \in I$, and suppose β is a sequence of actions in $ext(A)$ such that $\beta|A_i = trace(\alpha_i)$ for every $i \in I$. Then there is an execution α of A such that $\beta = trace(\alpha)$ and $\alpha_i = \alpha|A_i$ for every $i \in I$.*

The final theorem says that traces of component automata can also be pasted together to form a trace of the composition.

Theorem 3.2.4 *Let $\{A_i\}_{i \in I}$ be a compatible collection of automata and let $A = \prod_{i \in I} A_i$. Suppose β is a sequence of actions in $ext(A)$. If $\beta|A_i \in traces(A_i)$ for every $i \in I$, then $\beta \in traces(A)$.*

Theorem 3.2.4 implies that in order to show that a sequence is a trace of a system, it is enough to show that its projection on each individual system component is a trace of that component.

Chapter 4

The VS service

In this chapter we describe the view-oriented group communication service VS introduced in [41]. The VS service deals with views. We describe "variations" of the service, which deal with configurations.

4.1 The VS service

The VS service is a view-oriented group communication service. The name VS stands for "view synchrony" and refers to the property that a message sent within a particular view is delivered only to members of that view and only if they are still in that view. This seems to be the most important property of group communication services that go under the label of "virtual synchrony" (the expression "virtual synchrony" has been actually semantically overloaded and several virtual synchronous services provide different guarantees).

Another important feature of the VS specification is that it requires that the sequence of messages received by two different processes within a given view are such that one is the prefix of the other. Finally, new views are reported to their members in order of view identifier.

The external actions of the VS specification include VS-GPSND$(m)_p$ actions, representing the client at p sending a message m, and VS-GPRCV$(m)_{p,q}$ actions, representing the delivery to q of the message m sent by p. Output actions VS-SAFE$(m)_{p,q}$ are also provided at q to report that the earlier message m from p has been delivered to all locations in the current view as known by q.

The VS service informs its clients of group status changes through VS-NEWVIEW$(\langle g, P\rangle)_p$ actions (with $p \in P$), which tells p that the view identifier g is associated with membership set P and that, until another VS-NEWVIEW occurs, the following messages will be in this view. After any finite execution, the *current view* at p is defined as the argument v in the last NEWVIEW$(v)_p$ event, if any, otherwise it is either the initial view $\langle g_0, P_0\rangle$ if $p \in P_0$, or $\perp$ if $p \notin P_0$. This reflects the concept that the system starts with the processors in P_0 forming the group, and other processors unaware of the group.

VS

Signature:

Input: $\text{VS-GPSND}(m)_p$, $m \in \mathcal{M}$, $p \in \mathcal{P}$

Internal: $\text{VS-CREATEVIEW}(v)$, $v \in \mathcal{V}$
$\text{VS-ORDER}(m, p, g)$, $m \in \mathcal{M}$, $p \in \mathcal{P}$, $g \in \mathcal{G}$

Output: $\text{VS-GPRCV}(m)_{p,q}$, $m \in \mathcal{M}$, $p, q \in \mathcal{P}$
$\text{VS-SAFE}(m)_{p,q}$, $m \in \mathcal{M}$, $p, q \in \mathcal{P}$,
$\text{VS-NEWVIEW}(v)_p$, $v \in \mathcal{V}$, $p \in v.set$

State:

created $\in 2^{\mathcal{V}}$, init $\{v_0\}$
for each $p \in \mathcal{P}$:
  current-viewid$[p] \in \mathcal{G}_\perp$, init g_0 if $p \in P_0$, $\perp$ else
for each $g \in \mathcal{G}$:
  queue$[g] \in$ *seqof*$(\mathcal{M} \times \mathcal{P})$, init λ

for each $p \in \mathcal{P}$, $g \in \mathcal{G}$:
  pending$[p, g] \in$ *seqof*$(\mathcal{M})$, init λ
  next$[p, g] \in \mathbf{N}^{>0}$, init 1
  next-safe$[p, g] \in \mathbf{N}^{>0}$, init 1

Transitions:

internal $\text{VS-CREATEVIEW}(v)$
  Pre: $\forall w \in$ *created* : $v.id > w.id$
  Eff: *created* := *created* $\cup \{v\}$

output $\text{VS-NEWVIEW}(v)_p$
  Pre: $v \in$ *created*
    $v.id >$ *current-viewid*$[p]$
  Eff: *current-viewid*$[p] := v.id$

input $\text{VS-GPSND}(m)_p$
  Eff: if *current-viewid*$[p] \neq \perp$ then
      append m to *pending*$[p,$ *current-viewid*$[p]]$

internal $\text{VS-ORDER}(m, p, g)$
  Pre: m is head of *pending*$[p, g]$
  Eff: remove head of *pending*$[p, g]$
    append $\langle m, p\rangle$ to *queue*$[g]$

output $\text{VS-GPRCV}(m)_{p,q}$, choose g
  Pre: $g \neq \perp$
    $g =$ *current-viewid*$[q]$
    queue$[g]($*next*$[q, g]) = \langle m, p\rangle$
  Eff: *next*$[q, g] :=$ *next*$[q, g] + 1$

output $\text{VS-SAFE}(m)_{p,q}$, choose g, P
  Pre: $g \neq \perp$
    $g =$ *current-viewid*$[q]$
    $\langle g, P\rangle \in$ *created*
    queue$[g]($*next-safe*$[q, g]) = \langle m, p\rangle$
    for all $r \in P$:
      next$[r, g] >$ *next-safe*$[q, g]$
  Eff: *next-safe*$[q, g] :=$ *next-safe*$[q, g] + 1$

Figure 4-1: The VS service

The code is given in Figure 4-1.

The state of the VS service keeps track of the created views in variable *created*, and for each processor p, it keeps track of the current view at p, in variable *current-view*$[p]$. For each view, incoming message from a client at p are first buffered into a queue for processor p, *pending*$[p, g]$, and then they are "ordered" into a global queue for the view, *queue*$[g]$. The pointers *next*$[p, g]$ and *next-safe*$[p, g]$ point to, respectively, the next message of the gloabal queue that has to be delivered to the client at p and the next safe indications that has to be delivered to the client at p. In any trace of the VS service, there is a natural correspondence between VS-GPRCV events and the VS-GPSND events that cause them, and between VS-SAFE events and the VS-GPSND events that cause them.

The actions for creating a view and for informing a processor of a new view are straightforward (recall that the signature ensures that only members, but not necessarily all members, receive notification of a new view).

A message that is sent before the sender knows of any view (when the current view is $\perp$) is simply ignored, and never delivered anywhere.

Note that VS specification does not include any restrictions on when a new view might be formed. Clearly it is possible to analyze the service conditionally to some restrictions on the execution. Fir example, the performance and fault-tolerance property analysis provided in [41], does consider some restrictions: it implies that "capricious" view changes must stop shortly after the behavior of the underlying physical system stabilizes.

We note that the fact that VS allows views to be created only in order of view identifier is unimportant: weakening this requirement to allow out-of-order view creation would not change the external behavior, because VS-NEWVIEW actions are constrained to occur in such a way that views are delivered in order of view indentifiers anyway.

The following are safety properties of the VS service which we will be using in Chapter 5.

- New views are reported in increasing order of view identifier (Monotone views property);
- Messages sent in a view are delivered only within that view (View synchrony property);
- The sequences of messages delivered in a view at any two processors are such that one sequence is a prefix of the other (Prefix order property).

The following invariant holds.

Invariant 4.1.1 (VS)
In any reachable state, if $v, v' \in$ *created and* $v.id = v'.id$*, then* $v = v'$.

4.2 Variations: The CS and LC services

In many applications involving shared data, updates to the shared data have to be agreed upon by all the members of a view. In order to improve availability of the service and to balance the load of the system it is desirable to make updates without involving all the members of a view while still preserving consistency. This is achieved by using *configurations*.

A configuration is different from an ordinary view in that it is an ordinary view equipped with a set of subsets of the members of the view which satisfy the property that any two sets intersect. Such sets are called *quorums*. Hence a configuration is a view plus a set of quorums. The intersection property of quorums guarantees consistency within a given configuration: indeed for any given quorum there is always at least one process that has the latest update.

We will consider two more specialized types of configurations, which have been introduced in Chapter 3.

On such a configuration is the read-write-quorum configuration. Recalling the definition from Chapter 3, we have that a configuration is $c = \langle g, P, \mathcal{R}, \mathcal{W} \rangle$, where g is a configuration identifier, P is the set of members and $\mathcal{R}$ and $\mathcal{W}$ are the sets of read and write quorums.

Another such a configuration is the leader configuration. Recalling the definition from Chapter 3, we have that, in this case, a configuration is $c = \langle g, P, \mathcal{Q}, p \rangle$, where g is a configuration identifier, P is the set of members and $\mathcal{Q}$ is the sets of quorums and p is the leader of the configuration.

The VS service supports ordinary views $v = \langle g, P \rangle$ but can be easily generalized to configurations. We call these generalizations CS and LC, respectively, for the read-write-quorums configurations and for the leader configurations. The only difference between CS and VS, as well as LC and VS, is that CS and LC announce configurations while VS announces ordinary views. The code of CS, as well as that of LC, is exactly the code of VS. Indeed configurations are treated as a single entity, as are ordinary views. The reason we "rename" the code is because the two services are actually different (one provides views and the other provides configurations), thus we must distinguish them.

Figure 4-2 shows the code of the CS and LC specifications. Since these codes are the same as VS all the properties and invariants of VS apply to CS and LC too. In particular we have that configurations are reported in increasing order of configuration identifier (Monotone configurations property), messages sent in a configuration are delivered only within that configuration (Configuration synchrony property), and the sequence of messages delivered in a configuration at any two processes are such that one is a prefix of the other.

CS and LC

Signature:

Input: $\text{CS-GPSND}(m)_p$, $m \in \mathcal{M}$, $p \in \mathcal{P}$

Internal: $\text{CS-CREATECONF}(c)$, $c \in \mathcal{C}$

$\text{CS-ORDER}(m, p, g)$, $m \in \mathcal{M}$, $p \in \mathcal{P}$, $g \in \mathcal{G}$

Output: $\text{CS-GPRCV}(m)_{p,q}$, $m \in \mathcal{M}$, $p, q \in \mathcal{P}$

$\text{CS-SAFE}(m)_{p,q}$, $m \in \mathcal{M}$, $p, q \in \mathcal{P}$,

$\text{CS-NEWCONF}(c)_p$, $c \in \mathcal{C}$, $p \in c.set$

State:

created $\in 2^{\mathcal{C}}$, init $\{c_0\}$

for each $p \in \mathcal{P}$:

  current-confid$[p] \in \mathcal{G}_\perp$, init g_0 if $p \in P_0$, $\perp$ else

for each $g \in \mathcal{G}$:

  queue$[g] \in$ *seqof* $(\mathcal{M} \times \mathcal{P})$, init λ

for each $p \in \mathcal{P}$, $g \in \mathcal{G}$:

  pending$[p, g] \in$ *seqof* $(\mathcal{M})$, init λ

  next$[p, g] \in \mathbf{N}^{>0}$, init 1

  next-safe$[p, g] \in \mathbf{N}^{>0}$, init 1

Transitions:

internal $\text{CS-CREATECONF}(c)$

  Pre: $\forall w \in$ *created* : $c.id > w.id$

  Eff: *created* := *created* $\cup \{c\}$

output $\text{CS-NEWCONF}(c)_p$

  Pre: $c \in$ *created*

  $c.id >$ *current-confid*$[p]$

  Eff: *current-confid*$[p] := c.id$

input $\text{CS-GPSND}(m)_p$

  Eff: if *current-confid*$[p] \neq \perp$ then

    append m to *pending*$[p,$ *current-confid*$[p]]$

internal $\text{CS-ORDER}(m, p, g)$

  Pre: m is head of *pending*$[p, g]$

  Eff: remove head of *pending*$[p, g]$

  append $\langle m, p \rangle$ to *queue*$[g]$

output $\text{CS-GPRCV}(m)_{p,q}$, choose g

  Pre: $g =$ *current-confid*$[q]$

  $g \neq \perp$

  queue$[g]($*next*$[q, g]) = \langle m, p \rangle$

  Eff: *next*$[q, g] :=$ *next*$[q, g] + 1$

output $\text{CS-SAFE}(m)_{p,q}$, choose g, P

  Pre: $g =$ *current-confid*$[q]$

  $g \neq \perp$

  $\langle g, P \rangle \in$ *created*

  queue$[g]($*next-safe*$[q, g]) = \langle m, p \rangle$

  for all $r \in P$:

    next$[r, g] >$ *next-safe*$[q, g]$

  Eff: *next-safe*$[q, g] :=$ *next-safe*$[q, g] + 1$

Figure 4-2: The CS service

Chapter 5

The DVS service

In this chapter we present DVS, a specification for a dynamic primary view group communication service. Section 5.1 provides the DVS specification, Section 5.2 provides an implementation of DVS and finally Section 5.3 describes an application that uses DVS as building block. Section 5.4 closes the chapter with some remarks.

5.1 The DVS specification

The DVS service works as follows. Each client of the service has a "current" view of the group of processes. A process can send a message to all other members of its current view and the service guarantees that messages sent within a view are delivered only within that view and each member of the view receives messages in the same order as other members. However, not all messages need to be delivered to all members. The service also provides a "safe" notification for a particular message m that tells the recipient that message m has been received by all the members of the current view. New views are announced to all members of the new view and new views are guaranteed to be "primary" views. Primary views are defined according to a dynamic notion [55]: a new primary needs to contain a majority of the members of the previous primary. The DVS service allows the clients to "register" a new view after completing the pre-processing for that view.

The specification is given in Figure 5-1. In this specification, $\mathcal{M}_c \subseteq \mathcal{M}$ denotes the set of messages that clients may use for communication. The most interesting part of the DVS specification is the transition definition for DVS-CREATEVIEW(v). The precondition specifies the properties that a view must satisfy in order to be considered primary. The precondition says that $v.set$ must intersect the membership set of all previously-created smaller-id views w for which there is no intervening totally registered view – that is, the set of all "possible previous primary views". Since (for convenience) we allow out-of-order view creation in DVS, we also include a symmetric condition for previously-created larger-id views. All created views are recorded in *created*.

DVS

Signature:

Input: $\text{DVS-GPSND}(m)_p$, $m \in \mathcal{M}_c$, $p \in \mathcal{P}$
DVS-REGISTER_p, $p \in \mathcal{P}$

Internal: $\text{DVS-CREATEVIEW}(v)$, $v \in \mathcal{V}$
$\text{DVS-ORDER}(m, p, g)$, $m \in \mathcal{M}_c$, $p \in \mathcal{P}$, $g \in \mathcal{G}$

Output: $\text{DVS-GPRCV}(m)_{p,q}$, $m \in \mathcal{M}_c$, $p, q \in \mathcal{P}$
$\text{DVS-SAFE}(m)_{p,q}$, $m \in \mathcal{M}_c$, $p, q \in \mathcal{P}$
$\text{DVS-NEWVIEW}(v)_p$, $v \in \mathcal{V}$, $p \in v.set$

State:

$created \in 2^{\mathcal{V}}$, init $\{v_0\}$
for each $p \in \mathcal{P}$:
  $current\text{-}viewid[p] \in \mathcal{G}_\perp$, init g_0 if $p \in P_0$, $\perp$ else
for each $g \in \mathcal{G}$:
  $queue[g] \in seqof(\mathcal{M}_c \times \mathcal{P})$, init λ
  $attempted[g] \in 2^{\mathcal{P}}$, init P_0 if $g = g_0$, $\emptyset$ else
  $registered[g] \in 2^{\mathcal{P}}$, init P_0 if $g = g_0$, $\emptyset$ else

for each $p \in \mathcal{P}$, $g \in \mathcal{G}$:
  $pending[p, g] \in seqof(\mathcal{M}_c)$, init λ
  $next[p, g] \in \mathbf{N}^{>0}$, init 1
  $next\text{-}safe[p, g] \in \mathbf{N}^{>0}$, init 1

Derived variables:

$\mathcal{A}tt \in 2^{\mathcal{V}}$, defined as $\{v \in created \mid attempted[v.id] \neq \emptyset\}$
$\mathcal{T}ot\mathcal{A}tt \in 2^{\mathcal{V}}$, defined as $\{v \in created \mid v.set \subseteq attempted[v.id]\}$
$\mathcal{R}eg \in 2^{\mathcal{V}}$, defined as $\{v \in created \mid registered[v.id] \neq \emptyset\}$
$\mathcal{T}ot\mathcal{R}eg \in 2^{\mathcal{V}}$, defined as $\{v \in created \mid v.set \subseteq registered[v.id]\}$

Transitions:

internal $\text{DVS-CREATEVIEW}(v)$
Pre: $\forall w \in created : v.id \neq w.id$
  $\forall w \in created$:
    $\exists x \in \mathcal{T}ot\mathcal{R}eg : w.id < x.id < v.id$
    or $\exists x \in \mathcal{T}ot\mathcal{R}eg : v.id < x.id < w.id$
    or $v.set \cap w.set \neq \emptyset$
Eff: $created := created \cup \{v\}$

output $\text{DVS-NEWVIEW}(v)_p$
Pre: $v \in created$
  $v.id > current\text{-}viewid[p]$
Eff: $current\text{-}viewid[p] := v.id$
  $attempted[v.id] := attempted[v.id] \cup \{p\}$

input DVS-REGISTER_p
Eff: if $current\text{-}viewid[p] \neq \perp$ then
  $registered[current\text{-}viewid[p]] :=$
    $registered[current\text{-}viewid[p]] \cup \{p\}$

input $\text{DVS-GPSND}(m)_p$
Eff: if $current\text{-}viewid[p] \neq \perp$ then
  append m to $pending[p, current\text{-}viewid[p]]$

internal $\text{DVS-ORDER}(m, p, g)$
Pre: m is head of $pending[p, g]$
Eff: remove head of $pending[p, g]$
  append $\langle m, p \rangle$ to $queue[g]$

output $\text{DVS-GPRCV}(m)_{p,q}$, choose g
Pre: $g = current\text{-}viewid[q]$
  $queue[g](next[q, g]) = \langle m, p \rangle$
Eff: $next[q, g] := next[q, g] + 1$

output $\text{DVS-SAFE}(m)_{p,q}$, choose g, P
Pre: $g = current\text{-}viewid[q]$
  $\langle g, P \rangle \in created$
  $queue[g](next\text{-}safe[q, g]) = \langle m, p \rangle$
  for all $r \in P$:
    $next[r, g] > next\text{-}safe[q, g]$
Eff: $next\text{-}safe[q, g] := next\text{-}safe[q, g] + 1$

Figure 5-1: The DVS service.

The DVS service informs its clients of view changes using DVS-NEWVIEW$(\langle g, P\rangle)_p$ actions; such an action informs processor p that the view identifier g is associated with membership set P and that the current group of processors connected to p is P. After any finite execution, we define the *current view* at p to be the argument v in the last DVS-NEWVIEW$(v)_p$ event, if any, otherwise it is the initial view v_0 for processors in P_0 and is undefined for other processors. Even though views can be created out of view identifier order, the notification to each client is consistent with that order. Not every client needs to see every view. The variable *attempted* records, for each view, which processes have been notified of that view. Variable *attempted* is only used in proving the correctness of an implementation of DVS.

With the DVS-REGISTER$_p$ action, the client at p informs the service that it has obtained whatever information the application needs to begin operating in the new view v. For many applications, this will mean that p has received messages from every other member of view v, reporting its state at the start of v. The variable *registered* records, for each view, which process have registered that view. Variable *registered* is only used in proving the correctness of an implementation of DVS.

The DVS service allows a processor p to broadcast a message m using a DVS-GPSND$(m)_p$ action, and delivers the message to a processor q using a DVS-GPRCV$(m)_{p,q}$ action. DVS also uses a DVS-SAFE$(m)_{p,q}$ action to report to processor q that the earlier message m from p has been delivered to all members of the current view of q. DVS guarantees that messages sent by a processor p when the current view of p is v are delivered only within view v (i.e., only to processors in $v.set$ whose current view is v). Moreover, each processor receives messages in the same order as other processor and without gaps in the sequence of received messages; however, a processor may receive only a prefix of the sequence of messages received by another processor. Variables *queue*, *pending*, *next* and *next-safe* are used for handling the messages. Their use should be clear from the code.

There are four derived variables, $\mathcal{A}tt$, $\mathcal{T}ot\mathcal{A}tt$, $\mathcal{R}eg$ and $\mathcal{T}ot\mathcal{R}eg$. Informally, a view belongs to the set $\mathcal{A}tt$ if it has been reported to at least one member of the view (we say that it is *attempted*). A view belongs to the set $\mathcal{T}ot\mathcal{A}tt$ if it has been reported to all members of the view (we say that the view is *totally attempted*). Similarly, a view belongs to the set $\mathcal{R}eg$ if at least one member of the view has registered the view (we say that it is *registered*) and belongs to the set $\mathcal{T}ot\mathcal{R}eg$, if all members of the view have registered the view (we say that the view is *totally registered*).

We close this section with some invariants stating properties of DVS.

Invariant 5.1.1 (DVS)
In any reachable state, $\mathcal{T}ot\mathcal{A}tt \subseteq \mathcal{A}tt$, $\mathcal{T}ot\mathcal{R}eg \subseteq \mathcal{R}eg$, $\mathcal{R}eg \subseteq \mathcal{A}tt$, *and* $\mathcal{T}ot\mathcal{R}eg \subseteq \mathcal{T}ot\mathcal{A}tt$.

Invariant 5.1.2 (DVS)
In any reachable state if, $p \in attempted[g]$ *then* $current\text{-}viewid[p] \geq g$.

Invariant 5.1.3 expresses the key intersection property guaranteed by DVS; this is weaker than the intersection property required by static definitions of primary views, which says that all primary components must intersect. This invariant is our version of the correctness requirement for dynamic view services that two consecutive primary views intersect.

Invariant 5.1.3 (DVS)
In any reachable state, if $v, w \in created$, $v.id < w.id$, and there is no $x \in TotReg$ such that $v.id < x.id < w.id$, then $v.set \cap w.set \neq \emptyset$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. In the initial state $created = \{v_0\}$ and thus the invariant is vacuously true.

For the inductive step assume the invariant is true in s. We need to prove that it is true in s' for any possible step (s, π, s'). The only steps that can change the hypothesis from false to true are DVS-CREATEVIEW(v) and DVS-CREATEVIEW(w). The preconditions of these actions show that the needed conclusion holds. No step changes the conclusion from true to false. ∎

Invariant 5.1.4 says that if a view w is totally attempted, then any earlier view v has a member whose current view is later than v.

Invariant 5.1.4 (DVS)
In any reachable state, if $v \in created$, $w \in TotAtt$, and $v.id < w.id$, then there exists $p \in v.set$ with $current\text{-}viewid[p] > v.id$.

Proof: Consider any particular reachable state. Assume that $v \in created$, $w \in TotAtt$, and $v.id < w.id$. Then let y be the view in $TotAtt$ having the smallest viewid strictly greater than $v.id$. Then there is no $x \in TotAtt$ with $v.id < x.id < y.id$. Then Invariant 5.1.1 implies that there is no $x \in TotReg$ with $v.id < x.id < y.id$. Then Invariant 5.1.3 implies that $v.set \cap y.set \neq \emptyset$. Let $p \in v.set \cap y.set$; then $p \in attempted[y.id]$. Then Invariant 5.1.2 implies that $current\text{-}viewid[p] \geq y.id$. This implies $current\text{-}viewid[p] > v.id$. ∎

5.2 An implementation of DVS

We now give an implementation of the DVS service which we call DVS-IMPL. In Section 5.2.1 we describe DVS-IMPL, in Section 5.2.2 we provide some invariants of DVS-IMPL and finally in Section 5.2.3 we prove that DVS-IMPL implements DVS, in the sense of inclusion of sets of traces.

5.2.1 Overview

The implementation uses as a building block the group communication service VS (see Chapter 4) and uses ideas from [88]. The overall system is the composition of an automaton VS-TO-DVS$_p$ for each $p \in \mathcal{P}$, and VS, with the external actions of VS hidden in the composition. This system is called DVS-IMPL and is illustrated in Figure 5-2.

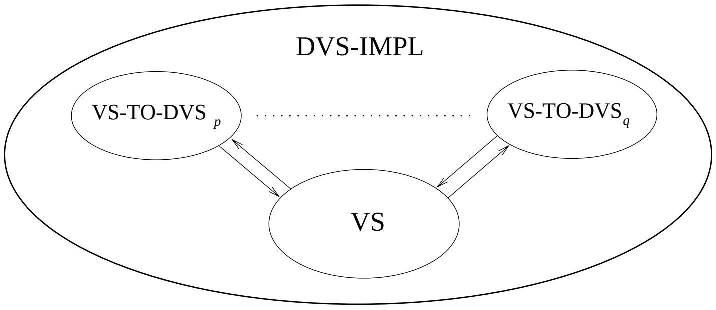

Figure 5-2: The DVS-IMPL system.

The automaton VS-TO-DVS$_p$ is given in Figure 5-3. VS-TO-DVS$_p$ uses special non-client messages, tagged either with *"info"* or *"registered"*. Thus, we use $\mathcal{M} = \mathcal{M}_c \cup \{(\text{"}\mathit{info}\text{"} \times \mathcal{V} \times 2^{\mathcal{V}})\} \cup \{\text{"}\mathit{registered}\text{"}\}$, where $\mathcal{M}_c$ is the set of all client messages and $\mathcal{M}$ is the universe of all messages. The *attempted*, *reg*, and *info-sent* state variables are not needed for the algorithm, but only for the correctness proof.

Automaton VS-TO-DVS$_p$ acts as a "filter", receiving VS-NEWVIEW inputs from the underlying VS service and deciding whether to accept the proposed views as primary views. If VS-TO-DVS$_p$ decides to accept some such view v, it "attempts" the view by performing a DVS-NEWVIEW(v) output. For each v, we think of the DVS internal DVS-CREATEVIEW(v) action as occurring at the time of the first DVS-NEWVIEW(v) event.

According to the DVS specification, the algorithm is supposed to guarantee nonempty intersection of each newly-created primary view v with any previously-created view w having no intervening totally registered view – a *global* condition involving *nonempty* intersection. The VS-TO-DVS$_p$ processors, however, do not have accurate knowledge of which primary views have been created by other processors, nor of which views are totally registered. Therefore, the processors employ a *local* check of *majority* intersection with known views, rather than a global check of nonempty intersection with existing views. Specifically, each VS-TO-DVS$_p$ keeps track of an "active" view *act*, which is the latest view that it knows to be totally registered, plus a set of "ambiguous" views *amb*, which are all the views that it knows have been attempted (i.e., have had a DVS-NEWVIEW action performed someplace), and whose ids are greater than $act.id$. We define $use = \{act\} \cup amb$. When VS-TO-DVS$_p$ receives a VS-NEWVIEW(v) input, it sends out *"info"* messages containing its current *act* and *amb* values to all the other processors in the new view, using the VS service, and then waits to receive corresponding *"info"* messages for view v from all the other processors in the view. After receiving this information (and

VS-TO-DVS

Signature:

Input: DVS-GPSND$(m)_p$, $m \in \mathcal{M}_c$
DVS-REGISTER$_p$
VS-NEWVIEW$(v)_p$, $v \in \mathcal{V}$, $p \in v.set$
VS-GPRCV$(m)_{q,p}$, $m \in \mathcal{M}$, $q \in \mathcal{P}$
VS-SAFE$(m)_{q,p}$, $m \in \mathcal{M}$, $q \in \mathcal{P}$

Internal: DVS-GARBAGE-COLLECT$(v)_p$, $v \in \mathcal{V}$

Output: VS-GPSND$(m)_p$, $m \in \mathcal{M}$
DVS-NEWVIEW$(v)_p$, $v \in \mathcal{V}$, $p \in v.set$
DVS-GPRCV$(m)_{q,p}$, $m \in \mathcal{M}_c$, $q \in \mathcal{P}$
DVS-SAFE$(m)_{q,p}$, $m \in \mathcal{M}_c$, $q \in \mathcal{P}$

State:

$cur \in \mathcal{V}_\perp$, init v_0 if $p \in P_0$, $\perp$ else
$client\text{-}cur \in \mathcal{V}_\perp$, init v_0 if $p \in P_0$, $\perp$ else
$act \in \mathcal{V}$, init v_0
$amb \in 2^{\mathcal{V}}$, init $\emptyset$
$attempted \in 2^{\mathcal{V}}$, init $\{v_0\}$ if $p \in P_0$,$\emptyset$ else
for each $g \in \mathcal{G}$, $q \in \mathcal{P}$
  $info\text{-}rcvd[q,g] \in (\mathcal{V} \times 2^{\mathcal{V}})_\perp$, init $\perp$
  $rcvd\text{-}rgst[q,g]$ a bool, init `false`

for each $g \in \mathcal{G}$
  $msgs\text{-}to\text{-}vs[g] \in seqof(\mathcal{M})$, init λ
  $msgs\text{-}from\text{-}vs[g] \in seqof(\mathcal{M}_c \times P)$, init λ
  $safe\text{-}from\text{-}vs[g] \in seqof(\mathcal{M}_c \times P)$, init λ
  $reg[g]$ a bool, init `true` if $p \in P_0$ and $g = g_0$, `false` else
  $info\text{-}sent[g] \in (\mathcal{V} \times 2^{\mathcal{V}})_\perp$, init $\perp$

Derived variables:
$\mathcal{A}tt \in 2^{\mathcal{V}}$, defined as $\mathcal{A}tt = \{v \in created \mid (\exists p \in v.set) v \in attempted_p\}$;
$\mathcal{R}eg \in 2^{\mathcal{V}}$, defined as $\mathcal{R}eg = \{v \in created \mid (\exists p \in v.set) reg[v.id]_p = \texttt{true}\}$;
$\mathcal{T}ot\mathcal{A}tt \in 2^{\mathcal{V}}$, defined as $\mathcal{T}ot\mathcal{A}tt = \{v \in created \mid (\forall p \in v.set) v \in attempted_p\}$;
$\mathcal{T}ot\mathcal{R}eg \in 2^{\mathcal{V}}$, defined as $\mathcal{T}ot\mathcal{R}eg = \{v \in created \mid (\forall p \in v.set) reg[v.id]_p = \texttt{true}\}$.
$use \in 2^{\mathcal{V}}$, defined as $use = \{act\} \cup amb$

Transitions:

input VS-NEWVIEW$(v)_p$
Eff: $cur := v$
  append $\langle$"$info$", $act, amb\rangle$ to $msgs\text{-}to\text{-}vs[cur.id]$
  $info\text{-}sent[cur.id] := \langle act, amb\rangle$

input VS-GPRCV$(\langle$"$info$"$, v, V\rangle)_{q,p}$
Eff: $info\text{-}rcvd[q, cur.id] := \langle v, V\rangle$
  if $v.id > act.id$ then $act := v$
  $amb := \{w \in amb \cup V \mid w.id > act.id\}$

input VS-SAFE$(\langle$"$info$"$, v, V\rangle)_{q,p}$
Eff: none

output DVS-NEWVIEW$(v)_p$
Pre: $v = cur$
  $v.id > client\text{-}cur.id$
  $\forall q \in v.set, q \neq p : info\text{-}rcvd[q, v.id] \neq \perp$
  $\forall w \in use : |v.set \cap w.set| > |w.set|/2$
Eff: $amb := amb \cup \{v\}$
  $attempted := attempted \cup \{v\}$
  $client\text{-}cur := v$

input DVS-REGISTER$_p$
Eff: if $client\text{-}cur \neq \perp$ then
    $reg[client\text{-}cur] :=$ `true`
    append $\langle$"$registered$"$\rangle$ to $msgs\text{-}to\text{-}vs[client\text{-}cur.id]$

input VS-GPRCV$(\langle$"$registered$"$\rangle)_{q,p}$
Eff: $rcvd\text{-}rgst[cur.id, q] :=$ `true`

input VS-SAFE$(\langle$"$registered$"$\rangle)_{q,p}$
Eff: none

internal DVS-GARBAGE-COLLECT$(v)_p$
Pre: $\forall q \in v.set : rcvd\text{-}rgst[q, v.id] = \texttt{true}$
  $v.id > act.id$
Eff: $act := v$
  $amb := \{w \in amb \mid w.id > act.id\}$

input DVS-GPSND$(m)_p$
Eff: if $client\text{-}cur.id_p \neq \perp$ then
    append m to $msgs\text{-}to\text{-}vs[client\text{-}cur.id]$

output VS-GPSND$(m)_p$
Pre: m is head of $msgs\text{-}to\text{-}vs[cur.id]$
Eff: remove head of $msgs\text{-}to\text{-}vs[cur.id]$

input VS-GPRCV$(m)_{q,p}$, where $m \in \mathcal{M}_c$
Eff: append $\langle m, q\rangle$ to $msgs\text{-}from\text{-}vs[cur.id]$

output DVS-GPRCV$(m)_{q,p}$
Pre: $\langle m, q\rangle$ is head of $msgs\text{-}from\text{-}vs[client\text{-}cur.id]$
Eff: remove head of $msgs\text{-}from\text{-}vs[client\text{-}cur.id]$

input VS-SAFE$(m)_{q,p}$, where $m \in \mathcal{M}_c$
Eff: append $\langle m, q\rangle$ to $safe\text{-}from\text{-}vs[cur.id]$

output DVS-SAFE$(m)_p$
Pre: $\langle m, q\rangle$ is head of $safe\text{-}from\text{-}vs[client\text{-}cur.id]$
Eff: remove head of $safe\text{-}from\text{-}vs[client\text{-}cur.id]$

Figure 5-3: The VS-TO-DVS$_p$ code.

updating its own *act* and *amb* accordingly), VS-TO-DVS$_p$ checks that v has a majority intersection with each view in *use*. If so, VS-TO-DVS$_p$ performs a DVS-NEWVIEW$_p$ output.

Then the clients can use the communication system to exchange state information as needed for processing in view v. When client at p has obtained enough information, it "registers" the view by means of action DVS-REGISTER$_p$, which causes processor p to send *"registered"* messages to the other members. When a processor receives *"registered"* messages for a view v from all members, it may perform garbage collection by discarding information about views with ids smaller than that of v. VS-TO-DVS uses VS to send and receive messages.

The system DVS-IMPL is defined as composition of all the VS-TO-DVS$_p$ automata and VS with all the external actions of VS hidden.

There are four derived variables for DVS-IMPL analogous to those of DVS, indicating the attempted, totally attempted, registered, and totally registered views, respectively. Another derived variable, use_p is defined in the code.

5.2.2 Invariants of DVS-IMPL

This section contains invariants of DVS-IMPL needed for the proof that DVS-IMPL implements DVS in Section 5.2.3. The first invariants state simple facts about DVS.

Invariant 5.2.1 (DVS-IMPL)
In any reachable state, if $cur_p \neq \perp$ then $current\text{-}viewid[p] = cur.id_p$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Fix p. In the initial state we have that $cur_p = \perp$.

For the inductive step assume the invariant is true in s. We need to prove that it is true in s' for any possible step (s, π, s'). Fix p. We prove the invariant considering each possible action π.

1. $\pi =$ VS-NEWVIEW$(v)_p$.

   By the code of π in VS, we have that $current\text{-}viewid[p] = v.id$. By the code of π in DVS-IMPL, we have that $cur.id_p = v.id$.

2. Other actions.

   Variables $current\text{-}viewid[p]$ and $cur.id_p$ are not modified. Hence the assertion cannot be made false.

∎

Invariant 5.2.2 (DVS-IMPL)
In any reachable state, if $v \in attempted_p$ then $client\text{-}cur.id_p \geq v.id$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Fix v, p. In the initial state we have that $attempted_p = \{v_0\}$ for $p \in P_0$ and $attempted_p = \perp$ for $p \notin P_0$. So assume that $v = v_0$ and $p \in P_0$. Then $client\text{-}cur_p = v_0$. Hence the invariant is true.

For the inductive step assume the invariant is true in s. We need to prove that it is true in s' for any possible step (s, π, s'). Fix v, p and assume that $v \in s'.attempted_p$. We distinguish two possible cases.

1. $v \in s.attempted_p$.

   By the inductive hypothesis we have that $s.client\text{-}cur_p \geq v.id$. By the monotonicity of $client\text{-}cur_p$ we have that $s'.client\text{-}cur_p \geq s.client\text{-}cur_p$.

2. $v \notin s.attempted_p$.

   Then it must be $\pi =$DVS-NEWVIEW$(v)_p$. The invariant follows from the code which sets $client\text{-}cur_p$ to v.

□

Invariant 5.2.3 (DVS-IMPL)
In any reachable state, if $v \in info\text{-}sent[g]_p = \langle x, X\rangle$ then $cur.id_p \geq g$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Fix v, p. In the initial state we have that $info\text{-}sent_p = \perp$ and thus the invariant is vacuously true.

For the inductive step assume the invariant is true in s. We need to prove that it is true in s' for any possible step (s, π, s'). Fix p, g, x, X and assume that $s'.info\text{-}sent[g]_p = \langle x, X\rangle$. We distinguish two possible cases.

1. $s.info\text{-}sent[g]_p = \langle x, X\rangle$

   By the inductive hypothesis we have that $s.cur_p \geq g$. By the monotonicity of cur_p we have that $s'.cur_p \geq s.cur_p$. Hence the invariant is true.

2. $s.info\text{-}sent[g]_p \neq \langle x, X\rangle$

   Then it must be $\pi =$VS-NEWVIEW$(v)_p$ and $g = v.id = s'.act.id_p$. Action VS-NEWVIEW$(v)_p$ sets $s'.cur$ to v, so $s'.cur.id = g$.

□

Invariant 5.2.4 (DVS-IMPL)
In any reachable state:

1. $v_0 \in \mathcal{TotReg}$.

2. $g_0 \leq v.id$ *for all* $v \in created$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Part 1 is true because in then initial state every processor $p \in P_0$ has $reg[g_0] = \texttt{true}$. Part 2 is true because the only view in *created* is v_0.

For the inductive step assume the invariant is true in s. We need to prove that it is true in s' for any possible step (s, π, s').

Consider Part 1 first. No view is ever removed from $\mathcal{TotReg}$. Hence no step can make the assertion false. Consider Part 2 now. Fix v and assume that $v \in s'.created$. We distinguish two cases.

1. $v \in s.created$.

   Then the assertion follows from the inductive hypothesis.

2. $v \notin s.created$.

   It must be π=VS-CREATEVIEW$(v)_p$. By the precondition of this action we have that $v.id > w.id$ for all $w \in s.created$. By the inductive hypothesis $g_0 \leq w.id$ for all $w \in s.created$. Since $s'.created = s.created \cup \{v\}$, it follows that $g_0 \leq w.id$ for all $w \in s'.created$.

$\square$

Invariant 5.2.5 (DVS-IMPL)
In any reachable state, if $rcvd\text{-}rgst[q, v.id]_p \neq \perp$ *then* $cur_p \neq \perp$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Fix p, q and v. In the initial state we have that $rcvd\text{-}rgst[q, v.id]_p = \perp$. Hence the invariant is vacuously true.

For the inductive step assume the invariant is true in s. We need to prove that it is true in s' for any possible step (s, π, s'). Fix p, q, v. We prove the invariant considering each possible action π. Assume that $s'.rcvd\text{-}rgst[q, v.id]_p \neq \perp$.

1. $\pi =$ VS-NEWVIEW$(v)_p$.

   Since $s'.cur_p = v$ we have that $s'.cur_p \neq \perp$ (VS cannot deliver $\perp$, it is not a view).

2. $\pi =$ VS-GPRCV$(\langle \text{"}registered\text{"} \rangle)_{p,q}$.

   By the precondition of π (see VS) we have that $s.current\text{-}viewid[p] \neq \perp$. By Invariant 5.2.1 we have $s.cur.id_p = s.current\text{-}viewid[p] \neq \perp$. Hence $s'.cur.id_p = s.cur.id_p \neq \perp$.

3. Other actions.

   Variables $rcvd\text{-}rgst[q, v.id]_p$ and cur_p are not modified. Hence the assertion cannot be made false.

□

Invariant 5.2.6 (DVS-IMPL)
In any reachable state, if $cur.id_p = \perp$ then $act_p = v_0$ and $amb_p = \emptyset$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Fix p. In the initial state we have that $act_p = v_0$ and $amb_p = \emptyset$.

For the inductive step assume the invariant is true in s. We need to prove that it is true in s' for any possible step (s, π, s'). Fix p. We prove the invariant considering each possible action π. Assume that $s'.cur_p = \perp$. Since no actions sets cur_p to $\perp$ it must be $s.cur_p = \perp$.

1. $\pi =$ VS-GPRCV$(\langle``info", v, V\rangle)_{p,q}$.

   This cannot happen. Indeed by precondition of π (see VS) we have that $s.current\text{-}viewid[p] \neq \perp$. By Invariant 5.2.1 we have $s.cur.id_p = s.\text{VS}.current\text{-}viewid[p]$ Hence $s'.cur.id_p = s.cur.id_p \neq \perp$. But we know that $s'.cur.id = \perp$.

2. $\pi =$ DVS-NEWVIEW$(v)v$.

   Cannot happen. Indeed the precondition of π says that $v = s.cur_p$. Since $s.cur.id = \perp$, we have $v = \perp$. Thus the precondition $v.id > client\text{-}cur.id_p$ cannot be satisfied ($\perp$ cannot be strictly greater than any view identifier).

3. $\pi =$ DVS-GARBAGE-COLLECT(v).

   Cannot happen. Indeed by Invariant 5.2.5 we have that $s.cur_p \neq \perp$. But we know that $s.cur_p = \perp$.

4. Other actions.

   Variables cur_p, act_p and amb_p are not modified. Hence the assertion cannot be made false.

□

The following invariant states that if an *"info"* message is in transit for view v or has been received by some process q in view v then there exists a process p that has sent the *"info"* in view v and such that its current view is either v or a later one.

Invariant 5.2.7 (DVS-IMPL)
In any reachable state, let C be the following condition:

$\langle$"*info*"$, x, X\rangle \in$ *msgs-to-vs*$[g]_p$ *or* $\langle$"*info*"$, x, X\rangle \in$ *pending*$[p, g]$ *or* $\langle\langle$"*info*"$, x, X\rangle, p\rangle \in$ *queue*$[g]$ *or* *info-rcvd*$[p, g]_q = \langle x, X\rangle$.

If C is true then *info-sent*$[g]_p = \langle x, X\rangle$ *and* $cur.id_p \geq g$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Fix p, q, g, x and X. In the initial state *msgs-to-vs*$[g]_p = \lambda$, *pending*$[p, g] = \lambda$, *queue*$[g] = \lambda$ and *info-rcvd*$[p, g]_q = \perp$. Hence, in the initial state, C is false and the invariant is vacuously true.

For the inductive step assume that the invariant is true in a reachable state s. We need to prove that it is true in state s' for any possible step (s, π, s') of the execution. Fix p, q, g, x, and X and assume that C is true in s'.

1. $\pi =$ VS-NEWVIEW$(v)_p$.

   By the code of π, $s'.cur_p = v$. Assume $v.id \neq g$. Then the code of π shows that none of *msgs-to-vs*$[g]_p$, *pending*$[p, g]$, *queue*$[g]$ or *info-rcvd*$[p, g]_q$ is changed during this step. Thus C is true also in s. By the inductive hypothesis we have $s.$*info-sent*$[g]_p = \langle x, X\rangle$ and $cur.id_p \geq g$. Since we are considering the case $v.id \neq g$, we have that *info-sent*$[g]_p$ is not changed by π. Moreover the precondition of π (see VS) shows that $s'.$*current-viewid*$[p] > s.$*current-viewid*$[p]$. By Invariant 5.2.1, $cur.id_p =$ *current-viewid*$[p]$, so $s'.cur.id_p > s.cur.id_p$. This completes showing the conclusion for the situation $w.id \neq g$.

   Assume now $v.id = g$. The code shows $s'.cur.id_p = g$ as required. It remains to show that $\langle x, X\rangle \in$ *info-sent*$[g]_p$.
   Action π does not alter the values of *pending*$[p, g]$, *queue*$[g]$ and *info-rcvd*$[p, g]_q$ and appends $\langle$"*info*"$, s.act_p, s.amb_p\rangle$ to *msgs-to-vs*$[g]_p$. We claim that it must be $x = s.act_p$ and $X = s.amb_p$. Indeed if it is not so, then condition C is true also in state s (for the given p, q, g, x, X) and by the inductive hypothesis we have $s.cur.id_p \geq g = w.id$. By Invariant 5.2.1, $s.$*current-viewid*$[p] \geq w.id$. But this contradicts the precondition of π (see VS).
   Thus $x = s.act_p$ and $X = s.amb_p$. Then the code of π shows that $\langle x, X\rangle \in$ *info-sent*$[g]_p$, as required.

2. $\pi =$ VS-GPRCV$(\langle$"*info*"$, v, V\rangle)_{p,q}$.

   If $g \neq cur.id_q$ then since C is true in s' it is true also in s (for the given p, q, g, x, X). Thus the inductive hypothesis is true. Since the code does not change *info-sent*$[g]_p$ and $cur.id_p$, the invariant follows from the inductive hypothesis.

   Hence assume that $g = cur.id_q$. First consider the case $x = v$ and $X = V$. In this case, by the precondition of π (see VS) we have that $\langle\langle$"*info*"$, x, X\rangle, p\rangle \in$ *queue*$[g]$. Then the invariant follows from the inductive hypothesis.

Consider now the case $x \neq v$ or $X \neq V$. In this case, by the code, we have that $s'.\mathit{info\text{-}rcvd}[p,g]_q \neq \langle x, X\rangle$. Since C is true in s', it must be that $\langle\text{“}\mathit{info}\text{''}, x, X\rangle \in \mathit{msgs\text{-}to\text{-}vs}[g]_p$ or $\langle\text{“}\mathit{info}\text{''}, x, X\rangle \in \mathit{pending}[p,g]$ or $\langle\langle\text{“}\mathit{info}\text{''}, x, X\rangle, p\rangle \in \mathit{queue}[g]$ is true in s'. Variables $\mathit{msgs\text{-}to\text{-}vs}[g]_p$, $\mathit{pending}[p,g]$ and $\mathit{queue}[g]$ are not changed by π. Hence C is true in s. The invariant follows from the inductive hypothesis.

3. $\pi =$ VS-GPSND$(\langle\text{“}\mathit{info}\text{''}, v, V\rangle)_p$.

   If $g \neq \mathit{client\text{-}cur.id}_p$ then since C is true in s' it is true also in s (for the given p, q, g, x, X). Thus the inductive hypothesis is true. Since the code does not change $\mathit{info\text{-}sent}[g]_p$ and $\mathit{cur.id}_p$, the invariant follows from the inductive hypothesis.

   Hence assume that $g = \mathit{client\text{-}cur.id}_p$. First consider the case $x = v$ and $X = V$. In this case, by the precondition of π (see DVS-IMPL) we have that $\langle\langle\text{“}\mathit{info}\text{''}, x, X\rangle, p\rangle \in \mathit{msgs\text{-}to\text{-}vs}[g]$. Then the invariant follows from the inductive hypothesis.

   Consider now the case $x \neq v$ or $X \neq V$. Since C is true in s' we have that C is true in s too. Indeed no $\langle\text{“}\mathit{info}\text{''}, x, X\rangle$ message is deleted and $\mathit{info\text{-}rcvd}[p,g]_q$ is not changed. The invariant follows from the inductive hypothesis.

4. $\pi =$ VS-ORDER$(\langle\text{“}\mathit{info}\text{''}, v, V\rangle, p, g)$.

   First consider the case $x = v$ and $X = V$. In this case, by the precondition of π we have that $\langle\langle\text{“}\mathit{info}\text{''}, x, X\rangle, p\rangle \in \mathit{pending}[g]$. Then the invariant follows from the inductive hypothesis.

   Consider now the case $x \neq v$ or $X \neq V$. Since C is true in s' we have that C is true in s too. Indeed no $\langle\text{“}\mathit{info}\text{''}, x, X\rangle$ message is deleted and $\mathit{info\text{-}rcvd}[p,g]_q$ is not changed. The invariant follows from the inductive hypothesis.

5. Other actions.

   Condition C never changes from false to true and variables $\mathit{info\text{-}sent}[g]_p$ and $\mathit{cur.id}_p$ are not modified. Hence the assertion cannot be made false.

$\square$

The following invariant states that if a *“registered”* message for view v has been sent by process p then variable $\mathit{reg}[v.id]_p$ is set to true (that is, the view has been registered by the client at p).

Invariant 5.2.8 (DVS-IMPL)
In any reachable state, let C be the following condition:

> $\langle\text{“}\mathit{registered}\text{''}\rangle \in \mathit{msgs\text{-}to\text{-}vs}[g]_p$ *or* $\langle\text{“}\mathit{registered}\text{''}\rangle \in \mathit{pending}[p,g]$ *or* $\langle\text{“}\mathit{registered}\text{''}, p\rangle \in \mathit{queue}[g]$ *or* $\mathit{rcvd\text{-}rgst}[p,g]_q = \texttt{true}$.

If C is true then $\mathit{reg}[g]_p = \texttt{true}$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Fix p, g, q. In the initial state we have that $msgs\text{-}to\text{-}vs[g]_p = \lambda$, $pending[p, g] = \lambda$, $queue[g] = \lambda$ and $rcvd\text{-}rgst[p, g]_q = \texttt{false}$. Hence C is false in the initial state and the invariant is vacuously true.

For the inductive step assume the invariant is true in s. We need to prove that it is true in s' for any possible step (s, π, s'). Fix p, g, q and assume that C is true in s'.

1. $\pi =$ DVS-REGISTER$_p$.

   If $s.client\text{-}cur.id_p \neq g$ then C is true also in s and the invariant follows from the inductive hypothesis. Hence assume $s.client\text{-}cur.id_p = g$. By the code of π we have that we have $reg[g]_p = \texttt{true}$.

2. $\pi =$ VS-GPSND$(\langle \text{“}registered''\rangle)_p$.

   If $s.current\text{-}viewid[p] \neq g$ then C is true also in s and the invariant follows from the inductive hypothesis. Hence assume $g = s.current\text{-}viewid[p]$. By Invariant 5.2.1 we have that $s.cur.id_p = s.current\text{-}viewid[p]$. Hence $s.cur.id_p = g$. By the precondition of π (see DVS-IMPL) we have that $\langle \text{“}registered''\rangle \in s.msgs\text{-}to\text{-}vs[g]_p$. Hence C is true in s and the invariant follows from the inductive hypothesis.

3. $\pi =$VS-ORDER$(\langle \text{“}registered'', p', g'\rangle)$.

   If $p' \neq p$ or $g' \neq g$ then C is true also in s and the invariant follows from the inductive hypothesis. Hence assume $p' = p$ and $g' = g$. By the precondition of π we have that $\langle \text{“}registered''\rangle \in s.pending[p, g]$. Hence C is true also in s and the invariant follows from the inductive hypothesis.

4. Other actions.

   Condition C never changes from false to true and variable $reg[g]_p$ is not modified. Hence the assertion cannot be made false.

$\square$

The following invariant states some facts about views in $\mathcal{T}ot\mathcal{R}eg$.

Invariant 5.2.9 (DVS-IMPL)
In any reachable state:

1. $act_p \in \mathcal{T}ot\mathcal{R}eg$.

2. *If* $info\text{-}sent[g]_p = \langle x, X\rangle$ *then* $x \in \mathcal{T}ot\mathcal{R}eg$.

3. $use_p \cap \mathcal{T}ot\mathcal{R}eg \neq \emptyset$.

Proof: First notice that Part 3 follows easily from Part 1 and the fact that, by definition, $act_p \in use_p$. Hence we only need to prove Parts 1 and 2.

By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. For Part 1, fix p. In the initial state $act_p = v_0$ and v_0 is totally registered by definition. For Part 2, fix p, g. In the initial state $\mathit{info\text{-}sent}[g]_p = \perp$. Hence the invariant is vacuously true.

For the inductive step assume the invariant is true in s. We need to prove that it is true in s' for any possible step (s, π, s'). Fix p, g, x and X. We prove the invariant by considering each possible action.

1. $\pi =$ VS-NEWVIEW$(v)_p$.

   Part 1 is still true in s' because act_p is not modified (as well as $\mathcal{T}ot\mathcal{R}eg$).

   Consider Part 2 now. Assume that $s'.\mathit{info\text{-}sent}[g]_p = \langle x, X\rangle$. If $v.id \neq g$ then $s.\mathit{info\text{-}sent}[g]_p = \langle x, X\rangle$ then by the inductive hypothesis we have that $x \in s.\mathcal{T}ot\mathcal{R}eg$. Since no view is ever removed from $\mathcal{T}ot\mathcal{R}eg$ we have that $x \in s'.\mathcal{T}ot\mathcal{R}eg$, as needed. Hence we can further assume that $v.id = g$. Since $s'.\mathit{info\text{-}sent}[g]_p = \langle x, X\rangle$ and action π sets $\mathit{info\text{-}sent}[g]_p = \langle act_p, amb_p\rangle$ it must be that $s.act_p = x$ and $s.amb_p = X$.

   By the inductive hypothesis, Part 1, we have that $s.act_p \in s.\mathcal{T}ot\mathcal{R}eg$. But $x = s.act_p$ and no view is removed from $\mathcal{T}ot\mathcal{R}eg$. Hence $x \in s'.\mathcal{T}ot\mathcal{R}eg$. Thus Part 2 is still true in s'.

2. $\pi =$ VS-GPRCV$(\langle \text{“}\mathit{info}'', v, V\rangle)_{p,q}$.

   Consider Part 1 first. If $s'.act_p = s.act_p$ then Part 1 follows by the inductive hypothesis. Hence assume that $s'.act_p \neq s.act_p$. By the code we have that $s'.act_p = v$. Thus we have to prove that $v \in \mathcal{T}ot\mathcal{R}eg$. By the precondition of π (in VS) we have $\langle\langle \text{“}\mathit{info}'', v, V\rangle, q\rangle \in s.queue[cur.id_p]$. Then Invariant 5.2.7 implies that $s.\mathit{info\text{-}sent}[cur.id_p]_q = \langle v, V\rangle$. By the inductive hypothesis, Part 2, we have that $v \in s.\mathcal{T}ot\mathcal{R}eg$, as needed.

   Part 2 is preserved because $\mathit{info\text{-}sent}[g]_p$ is not modified.

3. $\pi =$ DVS-GARBAGE-COLLECT$(v)_p$.

   Consider Part 1 first. If $s'.act_p = s.act_p$ then Part 1 follows by the inductive hypothesis. Hence assume that $s'.act_p \neq s.act_p$. By the code we have that $s'.act_p = v$. Hence we have to prove that $v \in \mathcal{T}ot\mathcal{R}eg$. By the precondition of π we have that $\mathit{rcvd\text{-}rgst}[q, v.id] = \texttt{true}$ for all $q \in v.set$. Then Invariant 5.2.8 implies that $v \in \mathcal{T}ot\mathcal{R}eg$.

   Part 2 is preserved because $\mathit{info\text{-}sent}[g]_p$ is not modified.

4. Other actions.

Variables act_p, $info\text{-}sent[g]_p$ (as well as $\mathcal{TotReg}$) are not modified. Hence the assertions cannot be made false.

□

The following invariant states that if process q is in a view which has been attempted by process p (which may or may not be q itself) then the current view of q is either v or a later one.

Invariant 5.2.10 (DVS-IMPL)
In any reachable state, if $v \in attempted_p$ and $q \in v.set$ then $cur.id_q \geq v.id$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Fix p, v and suppose that $v \in attempted_p$ and $q \in v.set$. If $p \notin P_0$ then $attempted_p = \emptyset$, a contradiction. On the other hand, if $p \in P_0$ then since $v \in attempted_p$, it must be that $v = v_0$. Moreover since $q \in v.set$ we have that $q \in P_0$. Hence $cur_q = v_0$, so $cur.id_q \geq v.id$, as needed.

For the inductive step assume the invariant is true in state s. We need to prove that it is true in s' for any possible step (s, π, s'). Fix p and v and assume that $v \in s'.attempted_p$ and $q \in v.set$. We distinguish two cases.

1. $v \in s.attempted_p$.

   By the inductive hypothesis we have that $s.cur.id_q \geq v.id$. By the monotonicity of $cur.id$ we have that $s'.cur.id_q \geq s.cur.id_q$.

2. $v \notin s.attempted_p$.

   It must be $\pi = \text{DVS-NEWVIEW}(v)_p$. We consider two possible cases: $q = p$ and $q \neq p$.

   Assume that $q = p$. Then Invariant 5.2.2 implies that $s'.client\text{-}cur_p \geq v.id$. Since $s'.cur.id_p = s'.client\text{-}cur_p$, we have that $s'.cur.id_p \geq v.id$, as needed.

   Assume that $q \neq p$. Then the precondition of π says that $s.info\text{-}rcvd[q, v.id] \neq \bot$. By Invariant 5.2.7 (used with p and q interchanged) we have that $cur.id_q \geq v.id$, as needed.

□

The following invariant states properties of views in the *use* set.

Invariant 5.2.11 (DVS-IMPL)
In any reachable state:

1. *If $cur_p \neq \bot$ and $w \in use_p$, then $w.id \leq cur.id_p$.*

2. *If $cur_p \neq \bot$ and $client\text{-}cur_p \neq cur_p$ and $w \in use_p$, then $w.id < cur.id_p$.*

3. *If* $\mathit{info\text{-}sent}[g]_p = \langle x, X \rangle$ *and* $w \in \{x\} \cup X$ *then* $w.id < g$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Consider Part 1 first. In the initial state we have that use_p is either empty or contains only v_0. In the former case Part 1 is vacuously true. In the latter case we have that $w = v_0$ and the invariant follows from the fact that g_0 is the minimum element of $\mathcal{G}$. Parts 2 and 3 are vacuously true. Indeed in the initial state $\mathit{client\text{-}cur}_p = cur_p$ and $\mathit{info\text{-}sent}[g]_p = \perp$.

For the inductive step assume the invariant is true in state s. We need to prove that it is true in s' for any possible step (s, π, s'). Fix p, g, x, X and w.

We prove that the invariant is still true in s' by considering each possible action π.

1. $\pi = \text{VS-NEWVIEW}(v)_p$

   First consider Part 1. Assume that $s'.cur_p \neq \perp$ and $w \in s'.use_p$. Then $w \in s.use_p$. If $s.cur_p = \perp$, then, by Invariant 5.2.6, $w = v_0$. Since $v_0.id$ is the minimum element of $\mathcal{G}$, we have that $w.id < s'.cur.id_p$. So assume that $s.cur_p \neq \perp$. In this case, by the inductive hypothesis, Part 1, we have that $w.id \leq s.cur.id_p$, which implies $w.id < s'.cur.id_p$.

   Hence Part 1 is still true in s'. Since we actually proved that $w.id < s'.cur.id_p$ also Part 2 is still true in s'.

   Now consider Part 3. Assume that $s'.\mathit{info\text{-}sent}[g]_p = \langle x, X \rangle$ and $w \in \{x\} \cup X$. If $g \neq v.id$ then we have that $s.\mathit{info\text{-}sent}[g]_p = \langle x, X \rangle$. By the inductive hypothesis, Part 3, we have $w.id < g$, as needed. Hence assume $g = v.id$. By the code of π, we have that $s.use_p = \{x\} \cup X$. Now if $s.cur_p = \perp$, then by Invariant 5.2.6, $w = v_0$. Since $v_0.id$ is the minimum element of $\mathcal{G}$, we have that $w.id < v.id = g$, as needed. So assume further that $s.cur_p \neq \perp$. In this case, the inductive hypothesis, Part 1, implies that $w.id \leq s.cur.id_p$, which implies $w.id < s'.cur.id_p = v.id = g$, as needed.

2. $\pi = \text{DVS-NEWVIEW}(v)_p$

   Consider Part 1 first. The only possible new element added to use_p is v. Since $v = s'.cur.id$, Part 1 still holds in s'. Part 2 is vacuously true, because $s'.\mathit{client\text{-}cur}_p = s'.cur_p$. Part 3 is preserved because $\mathit{info\text{-}sent}[g]_p$ is not modified.

3. $\pi = \text{DVS-GARBAGE-COLLECT}(v)_p$

   Consider Part 1. Assume that $s'.cur_p \neq \perp$ and that $w \in s'.use_p$. By the code $s'.cur_p = s.cur_p$. If $w \in s.use_p$ then by the inductive hypothesis Part 1 is true in s and thus it is still true in s'. Hence assume that $w \notin s.use_p$. By the code, this cannot happen because no view is added to use_p.

   Part 2 can be proved in a similar way. Part 3 is preserved because $\mathit{info\text{-}sent}[g]_p$ is not modified.

4. $\pi = $ VS-GPRCV$(\langle \text{“}info\text{''}, x, X\rangle)_{q,p}$

   The proof is exactly as in the previous case.

5. Other actions.

   Variables use_p, cur_p, $client\text{-}cur_p$ and $info\text{-}sent[g]_p$ are not modified. Hence none of the assertions can be made false.

$\square$

The following three invariants, say that certain views appear in *use* sets, or in *“info”* messages, unless they have been garbage-collected.

Invariant 5.2.12 (DVS-IMPL)
In any reachable state, if $w \in attempted_p$ then either $w \in use_p$ or $w.id < act.id_p$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. Fix p, w and suppose that $w \in attempted_p$. If $p \notin P_0$ then $attempted_p = \emptyset$, a contradiction. On the other hand, if $p \in P_0$ then since $w \in attempted_p$, it must be that $w = v_0$. But in this case also $act_p = v_0$, so $v_0 \in use_p$, as needed.

For the inductive step assume the invariant is true in state s. We need to prove that it is true in s' for any possible step (s, π, s'). So fix w and p such that $w \in s'.attempted_p$. We distinguish two possible cases.

1. $w \in s.attempted_p$.

   By the inductive hypothesis we have that either $w \in s.use_p$ or $w.id < s.act.id_p$. In the latter case, because of the monotonicity of $act.id_p$, we have $w.id < s'.act.id_p$. So assume that $w \in s.use_p$. If $w \in s'.use_p$ we are done, so assume further that $w \notin s'.use_p$. Then it must be that either $\pi = $ DVS-GARBAGE-COLLECT$(v)_p$ or $\pi = $ VS-GPRCV$(\langle \text{“}info\text{''}, x, X\rangle)_{r,p}$ for some r. In either case, the code implies that $s'.act_p > w.id$.

2. $w \notin s.attempted_p$.

   It must be $\pi = $ DVS-NEWVIEW$(v)_p$. By the code, view v is inserted into $attempted_p$, but also into amb_p (and hence into use_p). Thus the invariant is still true in s'.

$\square$

Invariant 5.2.13 (DVS-IMPL)
In any reachable state, if $info\text{-}rcvd[q, g]_p = \langle x, X\rangle$ and $w \in \{x\} \cup X$, then either $w \in use_p$ or $w.id < act.id_p$.

Proof: By induction on the length of an execution. The base case consists of proving that the invariant is true in the initial state. In the initial state $\textit{info-rcvd}[q,g]_p = \perp$ for any p,q,g. Hence the statement is vacuously true.

For the inductive step assume the invariant is true in state s. We need to prove that it is true in s' for any possible step (s,π,s'). Fix p, q, g, x, X and w, and assume that $s'.\textit{info-rcvd}[q,g]_p = \langle x, X\rangle$, and $w \in \{x\} \cup X$. We consider two cases:

1. $s.\textit{info-rcvd}[q,g]_p = \langle x, X\rangle$

   By the statement applied to s, we obtain that either $w \in s.\textit{use}_p$, or $s.\textit{act.id}_p > w.\textit{id}$. In the latter case, $s'.\textit{act.id}_p > w.\textit{id}$, because of monotonicity of $\textit{act.id}_p$. So assume that $w \in s.\textit{use}_p$. If $w \in s'.\textit{use}_p$ then we are done, so assume further that $w \notin s'.\textit{use}_p$. (That is, w is garbage-collected.)

   Then it must be that either $\pi = \text{DVS-GARBAGE-COLLECT}(v)_p$ or $\pi = \text{VS-GPRCV}(\langle \text{“}\textit{info}'', x, X\rangle)_{r,p}$ for some r. In either case, the code implies that $s'.\textit{act}_p > w.\textit{id}$.

2. $s.\textit{info-rcvd}[q,g]_p \neq \langle x, X\rangle$

   Then $\pi = \text{VS-GPRCV}(\langle \text{“}\textit{info}'', x, X\rangle)_{q,p}$. If $w \in s'.\textit{use}_p$ then we are done. Hence assume that $w \notin s'.\textit{use}_p$. By the code, we have that $s'.\textit{act}_p > w.\textit{id}$ (that is, w is garbage-collected).

∎

Invariant 5.2.14 (DVS-IMPL)
In any reachable state, if $\textit{info-sent}[g]_p = \langle x, X\rangle$, $w \in \textit{attempted}_p$, *and* $w.\textit{id} < g$, *then either* $w \in \{x\} \cup X$ *or* $w.\textit{id} < x.\textit{id}$.

Proof: By induction on the length of an execution. The base case consists of proving that the invariant is true in the initial state. In the initial state, $\textit{info-sent}[g]_p = \perp$ for all g,p, so the statement is vacuously true.

For the inductive step assume the invariant is true in state s. We need to prove that it is true in s' for any possible step (s,π,s'). Fix p, g, w, x, and X, and assume that $s'.\textit{info-sent}[g]_p = \langle x, X\rangle$, $w \in s'.\textit{attempted}_p$, and $w.\textit{id} < g$. We consider four cases:

1. $s.\textit{info-sent}[g]_p = \langle x, X\rangle$ and $w \in s.\textit{attempted}_p$.

   Then the statement for s implies that either $w \in \{x\} \cup X$ or $w.\textit{id} < x.\textit{id}$. In either case the statement is true in s' also.

2. $s.\textit{info-sent}[g]_p \neq \langle x, X\rangle$ and $w \notin s.\textit{attempted}_p$.

   This cannot happen because both conditions cannot become true in a single step: the first only becomes true by means of a $\text{VS-NEWVIEW}(v)_p$, for some view v, while the second only becomes true by means of $\text{DVS-NEWVIEW}(w)_p$.

3. $s.info\text{-}sent[g]_p \neq \langle x, X\rangle$ and $w \in s.attempted_p$.

   It must be $\pi =$ VS-NEWVIEW$(v)_p$, for some v, x must be $s.act_p$, and X must be $s.amb_p$. Invariant 5.2.12 implies that either $w \in s.use_p$ or $w.id < s.act.id_p$. Now, $s.use_p = \{s.act_p\} \cup s.amb_p = \{x\} \cup X$. So we have that either $w \in \{x\} \cup X$ or $w.id < x.id$, as needed.

4. $s.info\text{-}sent[g]_p = \langle x, X\rangle$ and $w \notin s.attempted_p$.

   Then π must be DVS-NEWVIEW$(w)_p$. We claim that this cannot happen: Since $s.info\text{-}sent[g]_p = \langle x, X\rangle$, by Invariant 5.2.3 we have $s.cur.id_p \geq g$. Since $g > w.id$, we have $s.cur_p > w.id$. But the precondition of π requires that $s.cur_p = w.id$. Hence π is not enabled in state s.

□

Invariant 5.2.15 says that two attempted views having no intervening totally registered view, and having a common member, q, that has attempted the first view, must intersect in a majority of processors. This is because, under these circumstances, information must flow from q to any processor that attempts the second view.

Invariant 5.2.15 (DVS-IMPL)
In any reachable state, suppose that $v \in attempted_p$, $q \in v.set$, $w \in attempted_q$, $w.id < v.id$, and there is no $x \in TotReg$ such that $w.id < x.id < v.id$. Then $|v.set \cap w.set| > |w.set|/2$.

Proof: By induction on the length of an execution. The base case consists of proving that the invariant is true in the initial state. In the initial state, only v_0 is attempted, so the hypotheses cannot be satisfied. Thus, the statement is vacuously true.

For the inductive step assume the invariant is true in state s. We need to prove that it is true in s' for any possible step (s, π, s'). Fix v, w, p, and q, and assume that $v \in s'.attempted_p$, $q \in v.set$, $w \in s'.attempted_q$, $w.id < v.id$, and there is no $x \in s'.TotReg$ such that $w.id < x.id < v.id$. Then also there is no $x \in s.TotReg$ such that $w.id < x.id < v.id$. We consider four cases:

1. $v \in s.attempted_p$ and $w \in s.attempted_q$.

   Then the statement for s implies that $|v.set \cap w.set| > |w.set|/2$, as needed.

2. $v \notin s.attempted_p$ and $w \notin s.attempted_q$.

   This cannot happen because we cannot have both v and w becoming attempted in a single step.

3. $v \notin s.attempted_p$ and $w \in s.attempted_q$.

   Then π must be DVS-NEWVIEW$(v)_p$. Since $q \in v.set$, by the precondition of π we have that $s.info\text{-}rcvd[q, v.id]_p = \langle x, X\rangle$ for some x and X. Then Invariant 5.2.7 implies that $s.info\text{-}sent[v.id]_q = \langle x, X\rangle$. Then (since $w.id < v.id$), Invariant 5.2.14 implies that either $w \in \{x\} \cup X$ or

$w.id < x.id$. If $w.id < x.id$, then we obtain a contradiction. Indeed by Invariant 5.2.9 $x \in s.\mathcal{T}ot\mathcal{R}eg$ and by Invariant 5.2.11, Part 3 (used with $w = x$) we have $x.id < v.id$. This contradicts the hypothesis. So $w \in \{x\} \cup X$.

Now by Invariant 5.2.13 we have that either $w \in s.use_p$ or $w.id < s.act.id_p$. In the former case, by the precondition of π, we have $|v.set \cap w.set| > |w.set|/2$. In the latter case, we obtain a contradiction. Indeed by Invariant 5.2.9 we have $s.act_p \in \mathcal{T}ot\mathcal{R}eg$. Moreover by the precondition of π, $s.cur_p$ cannot be $\perp$ and $s.cur_p > s.client\text{-}cur_p$ and, by definition, $s.act_p \in s.use_p$. Hence by Invariant 5.2.11, Part 2, we have $s.act.id_p < s.cur.id_p = v.id$. Thus we would have a totally registered view act such that $w.id < act.is < c.id$. This contradicts the inductive hypothesis.

4. $v \in s.attempted_p$ and $w \notin s.attempted_q$.

   Then π must be DVS-NEWVIEW$(w)_q$. But this cannot happen. Indeed since $v \in s.attempted_p$ and $q \in v.set$, Invariant 5.2.10 implies that $s.cur.id_q \geq v.id$. Since $v.id > w.id$, we have $s.cur.id_q > w.id$. But the precondition of action π requires $s.cur.id_q = w.id$, so π is not enabled in s.

□

Invariant 5.2.16 says that any attempted view v intersects the latest preceding totally registered view w in a majority of members of w.

Invariant 5.2.16 (DVS-IMPL)

In any reachable state, suppose that $v \in \mathcal{A}tt$, and $w \in \mathcal{T}ot\mathcal{R}eg$, $w.id < v.id$, and there is no $x \in \mathcal{T}ot\mathcal{R}eg$ such that $w.id < x.id < v.id$. Then $|v.set \cap w.set| > |w.set|/2$.

Proof: By induction on the length of an execution. The base case consists of proving that the invariant is true in the initial state. In the initial state, only v_0 is attempted, so the hypotheses cannot be satisfied. Thus, the statement is vacuously true.

For the inductive step assume the invariant is true in state s. We need to prove that it is true in s' for any possible step (s, π, s'). Fix v and w, and assume that $v \in s'.\mathcal{A}tt$, $w \in s'.\mathcal{T}ot\mathcal{R}eg$, $w.id < v.id$, and there is no $x \in s'.\mathcal{T}ot\mathcal{R}eg$ such that $w.id < x.id < v.id$. We consider four cases:

1. $v \in s.\mathcal{A}tt$ and $w \in s.\mathcal{T}ot\mathcal{R}eg$.

   Then, from the inductive hypothesis we have $|v.set \cap w.set| > |w.set|/2$.

2. $v \notin s.\mathcal{A}tt$ and $w \notin s.\mathcal{T}ot\mathcal{R}eg$.

   This cannot happen because we cannot have both v becoming attempted and w becoming totally registered in a single step.

3. $v \notin s.\mathcal{A}tt$ and $w \in s.\mathcal{T}ot\mathcal{R}eg$.

   Then π must be DVS-NEWVIEW$(v)_p$ for some p. The precondition of π implies that, for any view $y \in s.use_p$, $|v.set \cap y.set| > |y.set|/2$. Hence to prove the claim it is enough to prove that $w \in s.use_p$. We proceed by contradiction assuming that $w \notin s.use_p$.

   By Invariant 5.2.9, Part 3, $s.use_p \cap s.\mathcal{T}ot\mathcal{R}eg \neq \emptyset$. Let m be the view in $s.use_p \cap s.\mathcal{T}ot\mathcal{R}eg$ having the biggest identifier. We know that $m \neq w$ because $w \notin s.use_p$. Also, $m \neq v$, because $m \in s.\mathcal{T}ot\mathcal{R}eg$ and $v \notin s.\mathcal{T}ot\mathcal{R}eg$. It follows that $m.id \neq v.id$.

   We claim that $m.id < w.id$. We have already shown that $m.id \neq w.id$. Suppose for the sake of contradiction that $m.id > w.id$. From the precondition of action π we have that $s.cur = v$ and hence $s.cur \neq \perp$. Also from the precondition of π we have that $s.client\text{-}cur_p < s.cur_p$. Since $m \in s.use_p$, Invariant 5.2.11, Part 2, implies that $m.id < s.cur.id_p$ and since $s.cur = v$ we have we have $m.id < v.id$. So $w.id < m.id < v.id$. Since $m \in s'.\mathcal{T}ot\mathcal{R}eg$, this contradicts the hypothesis of the inductive step. Therefore, $m.id < w.id$.

   Let n be the view in $s.\mathcal{T}ot\mathcal{R}eg$ that has the smallest id strictly greater than that of m. Remember that $w \in s'.\mathcal{T}ot\mathcal{R}eg$ and since $\pi =$DVS-NEWVIEW$(v)_p$ we have that $w \in s.\mathcal{T}ot\mathcal{R}eg$; thus n exists and it holds $m.id < n.id \leq w.id < v.id$. Since $m \in s.use_p$, the precondition of π implies that $|v.set \cap m.set| > |m.set|/2$. By the statement applied to state s, $|n.set \cap m.set| > |m.set|/2$. Hence there exists a processor $q \in v.set \cap n.set$. By the precondition of π, $s.info\text{-}rcvd[q, v.id]_p = \langle x, X \rangle$ for some x, X. Then Invariant 5.2.7 implies that $s.info\text{-}sent[v.id]_q = \langle x, X \rangle$. Then Invariant 5.2.11, Part 3 (used with $w = x$), implies that $x.id < v.id$. Since $n \in s.\mathcal{T}ot\mathcal{R}eg$, we have that $n \in s.attempted_q$. Then Invariant 5.2.14 (used with $w = n$) implies that either $n \in \{x\} \cup X$ or $n.id < x.id$. In either case, $\{x\} \cup X$ contains a view $y \in s.\mathcal{T}ot\mathcal{R}eg$ (either n or x) such that $n.id \leq y.id < v.id$. Then Invariant 5.2.13 implies that either $y \in s.use_p$ or $y.id < s.act.id_p$. By Invariant 5.2.9, Part 1, $s.act_p \in s.\mathcal{T}ot\mathcal{R}eg$ and by definition, $s.act_p \in s.use_p$. So in either case, the hypothesis that m is the totally registered view with the largest id belonging to $s.use_p$ is contradicted.

4. $v \in s.\mathcal{A}tt$ and $w \notin s.\mathcal{T}ot\mathcal{R}eg$.

   Then π must be DVS-REGISTER$_p$ for some p. Let m be the view in $s.\mathcal{T}ot\mathcal{R}eg$ with the largest id that is strictly less than $w.id$. By the statement for s, we know that $|w.set \cap m.set| > |m.set|/2$ and $|v.set \cap m.set| > |m.set|/2$. Hence there is a processor $q \in w.set \cap v.set$.

   Since $v \in s.\mathcal{A}tt$, there exists a processor r such that $v \in s.attempted_r$. Thus also $v \in s'.attempted_r$. Since $w \in s'.\mathcal{T}ot\mathcal{R}eg$, we have that $w \in s'.attempted_q$. By assumption, there is no view $x \in s'.\mathcal{T}ot\mathcal{R}eg$ such that $w.id < x.id < v.id$. By Invariant 5.2.15 applied to state s' (with $p = r$), we have that $|v.set \cap w.set| > |w.set|/2$, as needed.

$\square$

The final invariant, a corollary to Invariant 5.2.16, is instrumental in proving that DVS-IMPL implements DVS.

Invariant 5.2.17 (DVS-IMPL)
In any reachable state, if $v, w \in \mathcal{A}tt$, $w.id < v.id$, and there is no $x \in \mathcal{T}ot\mathcal{R}eg$ with $w.id < x.id < v.id$, then $v.set \cap w.set \neq \emptyset$.

Proof: Suppose that v and w are as given. We consider two cases.

1. $w \in \mathcal{T}ot\mathcal{R}eg$.

   Since there is no $x \in \mathcal{T}ot\mathcal{R}eg$, Invariant 5.2.16 implies that $|v.set \cap w.set| > |w.set|/2$, which implies that $v.set \cap w.set \neq \{\}$, as needed.

2. $w \notin \mathcal{T}ot\mathcal{R}eg$.

   Then let $Y = \{y | y \in \mathcal{T}ot\mathcal{R}eg, y.id < w.id\}$. We first show that Y is nonempty: Invariant 5.2.4 implies that $v_0 \in \mathcal{T}ot\mathcal{R}eg$ and that $v_0.id \leq w.id$. If $v_0.id = w.id$, then by Invariant 4.1.1, we have $w = v_0$. But then $w \in \mathcal{T}ot\mathcal{R}eg$, a contradiction to the definition of this case. So we must have $v_0.id < w.id$, which implies that $v_0 \in Y$, so Y is nonempty.

   Now fix z to be the view in Y with the largest id. We have that there is no $x \in \mathcal{T}ot\mathcal{R}eg$ with $z.id < x.id < v.id$. Then Invariant 5.2.16 implies that $|w.set \cap z.set| > |z.set|/2$ and $|v.set \cap z.set| > |z.set|/2$. Together, these two facts imply that $v.set \cap w.set \neq \{\}$, as needed.

$\square$

5.2.3 Proof that DVS-IMPL implements DVS

We prove that DVS-IMPL implements DVS by defining a function $\mathcal{F}_{dvs}$ that maps states of DVS-IMPL to states of DVS and proving that this function is a *abstraction function*. Section 5.2.3 contains the definition of the function $\mathcal{F}_{dvs}$ and some auxiliary lemmas while Section 5.2.3 contains the proof that $\mathcal{F}_{dvs}$ is an abstraction function.

The abstraction function for DVS-IMPL.

DVS-IMPL uses VS to send client messages and messages generated by the implementation (*"info"* and *"registered"* messages). The abstraction function discards the non-client messages. Thus, if q is a finite sequence of client and non-client messages, we define $purge(q)$ to be the queue obtained by deleting any *"info"* or *"registered"* messages from q, and $purgesize(q)$ to be the number of *"info"* and *"registered"* messages in q. Figure 5-4 defines the abstraction function $\mathcal{F}_{dvs}$.

Next we give some simple consequences of the definition of $\mathcal{F}_{dvs}$. They deal with the messages delivered by DVS-IMPL. They state that these messages are exactly the ones that DVS would deliver to the client.

Let s be a state of DVS-IMPL. The state $t = \mathcal{F}_{dvs}(s)$ of DVS is the following.

- $t.created = \cup_{p \in \mathcal{P}} s.attempted_p$
- for each $p \in \mathcal{P}$, $t.current\text{-}viewid[p] = s.client\text{-}cur.id_p$
- for each $g \in \mathcal{G}$, $t.attempted[g] = \{p | g = v.id, v \in s.attempted_p\}$
- for each $g \in \mathcal{G}$, $t.registered[g] = \{p | s.reg[g]_p\}$
- for each $p \in \mathcal{P}$, $g \in \mathcal{G}$, $t.pending[p,g] = purge(s.pending[p,g]) \circ purge(s.msgs\text{-}to\text{-}vs[g]_p)$
- for each $g \in \mathcal{G}$, $t.queue[g] = purge(s.queue[g])$
- for each $p \in \mathcal{P}$, $g \in \mathcal{G}$,
  $t.next[p,g] = s.next[p,g] - purgesize(s.queue[g](1..next[p,g]-1)) - |s.msgs\text{-}from\text{-}vs[g]_p|$
- for each $p \in \mathcal{P}$, $g \in \mathcal{G}$, $t.next\text{-}safe[p,g] =$
  $s.next\text{-}safe[p,g] - purgesize(s.queue[g](1..next\text{-}safe[p,g]-1)) - |s.safe\text{-}from\text{-}vs[g]_p|$

Figure 5-4: The abstraction function $\mathcal{F}_{dvs}$.

Invariant 5.2.18 (DVS-IMPL)

In any reachable state s, if $s.msgs\text{-}from\text{-}vs[g]_p = \langle\langle m_1, q_1\rangle, \langle m_2, q_2\rangle, ..., \langle m_k, q_k\rangle\rangle$, then we have that $\mathcal{F}_{dvs}(s).queue[g](next[p,g]..next[p,g]+k-1) = \langle\langle m_1, q_1\rangle, \langle m_2, q_2\rangle, ..., \langle m_k, q_k\rangle\rangle$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. In the initial state no message is in $msgs\text{-}from\text{-}vs[g]_p$. Hence the invariant is vacuously true.

For the inductive step, assume that the invariant is true in state s. We need to prove that it is true in state s' for any possible step (s, π, s'). Fix p, g and $m_1, q_1, m_2, q_2, ..., m_k, q_k$ and assume that $s'.msgs\text{-}from\text{-}vs[g]_p = \langle\langle m_1, q_1\rangle, \langle m_2, q_2\rangle, ..., \langle m_k, q_k\rangle\rangle$. We distinguish the following cases.

1. $s.msgs\text{-}from\text{-}vs[g]_p = \langle\langle m_1, q_1\rangle, ..., \langle m_{k-1}, q_{k-1}\rangle\rangle$.

   It must be $\pi =$VS-GPRCV$(m_k)_{q_k,p}$. By the inductive hypothesis we have that $\mathcal{F}_{dvs}(s).queue[g](next[p,g]..next[p,g]+k-2) = \langle\langle m_1, q_1\rangle, ..., \langle m_{k-1}, q_{k-1}\rangle\rangle$.

   By the code in VS we have that $next[p,g]$ is increased by one and by the code in DVS we have that the size of $msgs\text{-}from\text{-}vs[g]_p$ also increases by one. Hence by the definition of $\mathcal{F}_{dvs}$, we have that $\mathcal{F}_{dvs}(s').next[p,g] = \mathcal{F}_{dvs}(s).next[p,g]$. Moreover $\mathcal{F}_{dvs}(s').queue[g] = \mathcal{F}_{dvs}(s).queue[g]$ and by the precondition of π we have that $\mathcal{F}_{dvs}(s).queue[g](s.next[p,g]+k-1) = \langle m_k, q_k\rangle$. Thus the invariant is still true in s'.

2. $s.msgs\text{-}from\text{-}vs[g]_p = \langle\langle m, q\rangle, \langle m_1, q_1\rangle, \langle m_2, q_2\rangle, ..., \langle m_k, q_k\rangle\rangle$.

   Then $\pi =$DVS-GPRCV$(m)_{q,p}$. By the inductive hypothesis we have that $\mathcal{F}_{dvs}(s).queue[g](next[p,g]..next[p,g]+k) = \langle\langle m, q\rangle, \langle m_1, q_1\rangle, \langle m_2, q_2\rangle, ..., \langle m_{k-1}, q_{k-1}\rangle\rangle$.

By the code we have that $next[p,g]$ is incremented by one. Since $\mathcal{F}_{dvs}(s').queue[g] = \mathcal{F}_{dvs}(s).queue[g]$, the invariant is still true in s'.

3. $s.msgs\text{-}from\text{-}vs[g]_p = s'.msgs\text{-}from\text{-}vs[g]_p$

   By the inductive hypothesis the assertion is true in state s. For any possible action in this case $\mathcal{F}_{dvs}(s').next[p,g] = \mathcal{F}_{dvs}(s).next[p,g]$ and the portion of $\mathcal{F}_{dvs}(s).queue[g]$ involved in the statement of the invariant never changes because messages are only appended to $queue[g]$. Thus the assertion cannot be made false.

4. Other cases.

   Not possible. Indeed $msgs\text{-}from\text{-}vs[g]_p$ either stay the same or is changed by appending a message or deleting the head.

$\square$

The following invariant follows easily from the previous one. It just states that the next message delivered by DVS-IMPL to a processor p is the same one that DVS delivers.

Invariant 5.2.19 (DVS-IMPL)
In any reachable state s, if $\langle m, q\rangle$ is head of $s.msgs\text{-}from\text{-}vs[g]_p$, then $\mathcal{F}_{dvs}(s).queue[g](next[p,g]) = \langle m, q\rangle$.

Proof: Follows easily from previous one. $\square$

Similar invariants hold for the delivery of safe messages.

Invariant 5.2.20 (DVS-IMPL)
In any reachable state s, we have that if $s.safe\text{-}from\text{-}vs[g]_p = \langle\langle m_1, q_1\rangle, \langle m_2, q_2\rangle, ..., \langle m_k, q_k\rangle\rangle$, then $\mathcal{F}_{dvs}(s).queue[g](next\text{-}safe[p,g], next\text{-}safe[p,g]+k-1) = \langle\langle m_1, q_1\rangle, \langle m_2, q_2\rangle, ..., \langle m_k, q_k\rangle\rangle$.

Proof: The proof is as for msgs except that it uses the *safe-from-vs* queue instead of *msgs-from-vs* and the pointer *next-safe* instead of *next*. $\square$

Invariant 5.2.21 (DVS-IMPL)
In any reachable state s, if $\langle m, q\rangle$ is head of $s.safe\text{-}from\text{-}vs[g]_p$, then $\mathcal{F}_{dvs}(s).queue[g](next\text{-}safe[p,g]) = \langle m, q\rangle$.

Proof: Follows easily from previous one. $\square$

Notice that v is totally registered in state s of DVS-IMPL if and only if it is totally registered in the state of DVS that appears in state $\mathcal{F}_{dvs}(s)$ of DVS.

Proof that $\mathcal{F}_{dvs}$ is abstraction an function.

In order to prove that $\mathcal{F}_{dvs}$ is an abstraction function we need to prove that for any initial state s of DVS-IMPL we have that $\mathcal{F}_{dvs}(s)$ is an initial state of DVS and that for any possible step π of DVS-IMPL there exists a sequence of α of steps of DVS such that the trace of α, that is the externally observable behavior, is equal to the trace of π. Lemmas 5.2.22 and 5.2.23, prove the above.

Lemma 5.2.22 *If s is an initial state of* DVS-IMPL *then $\mathcal{F}_{dvs}(s)$ is an initial state of* DVS.

Proof: Let s_0 be the unique initial state of DVS-IMPL and t_0 the unique initial state of DVS.

We have $s_0.attempted_p = \{v_0\}$ for $p \in P_0$ and $s_0.attempted_p = \emptyset$ for $p \notin P_0$. By the definition of $\mathcal{F}_{dvs}$ and the fact that $P_0 \neq \emptyset$ (because all membership sets are defined to be nonempty), we have $\mathcal{F}_{dvs}(s_0).created = \{v_0\}$. This is as in t_0.

We have $s_0.client\text{-}cur_p = \{v_0\}$ for $p \in P_0$ and $s_0.client\text{-}cur_p = \perp$ for $p \notin P_0$. By the definition of $\mathcal{F}_{dvs}$ we have $\mathcal{F}_{dvs}(s_0).current\text{-}viewid[p] = g_0$ for $p \in P_0$ and $\mathcal{F}_{dvs}(s_0).current\text{-}viewid[p] = \perp$ for $p \notin P_0$. This is as in t_0.

We have $s_0.attempted_p = \{v_0\}$ for $p \in P_0$ and $s_0.attempted_p = \emptyset$ for $p \notin P_0$. By the definition of $\mathcal{F}_{dvs}$ we have $\mathcal{F}_{dvs}(s_0).attempted[g_0] = P_0$ and $\mathcal{F}_{dvs}(s_0).attempted[g] = \emptyset$ for $g \neq g_0$. This is as in t_0.

Let $g \in \mathcal{G}$. We have that $s_0.reg[g]_p$ is `true` if and only if $p \in P_0$ and $g = g_0$. By the definition of $\mathcal{F}_{dvs}$ we have $\mathcal{F}_{dvs}(s_0).registered[g_0] = P_0$ and $\mathcal{F}_{dvs}(s_0).registered[g] = \emptyset$ for $g \neq g_0$, as in t_0.

Let $p \in \mathcal{P}$. We have that $s_0.msgs\text{-}to\text{-}vs[g]_p = \lambda$ and $s_0.pending[p, g] = \lambda$. By the definition of $\mathcal{F}_{dvs}$ we have $\mathcal{F}_{dvs}(s_0).pending[p, g] = \lambda$, as in t_0.

Let $g \in \mathcal{G}$. We have $s_0.queue[g] = \lambda$. By the definition of $\mathcal{F}_{dvs}$ we have $\mathcal{F}_{dvs}(s_0).queue[g] = \lambda$, as in t_0.

Let $p \in \mathcal{P}, g \in \mathcal{G}$. We have $s_0.next[p, g] = 1$, $purgesize(s.\text{VS}.queue[g]) = 0$ and $s_0.msgs\text{-}from\text{-}vs[g]_p = \lambda$. By the definition of $\mathcal{F}_{dvs}$ we have $\mathcal{F}_{dvs}(s_0).next[p, g] = 1$, as in t_0. A similar argument holds for *next-safe*.

Thus $\mathcal{F}_{dvs}(s_0) = t_0$, as needed. ∎

Lemma 5.2.23 *Let s be a reachable state of* DVS-IMPL*, $\mathcal{F}_{dvs}(s)$ a reachable state of* DVS-SYS*, and (s, π, s') a step of* DVS-IMPL*. Then there is an execution fragment α of* DVS-SYS *that goes from $\mathcal{F}_{dvs}(s)$ to $\mathcal{F}_{dvs}(s')$, such that $trace(\alpha) = trace(\pi)$.*

Proof: By case analysis based on the type of the action π. (The only interesting case is where $\pi =$ DVS-NEWVIEW$(v)_p$.) Define $t = \mathcal{F}_{dvs}(s)$ and $t' = \mathcal{F}_{dvs}(s')$.

1. $\pi =$ VS-CREATEVIEW(v)

   Then $trace((s, \pi, s')) = \lambda$. Action π modifies *created*. The definition of $\mathcal{F}_{dvs}$ is not sensitive to this change. Therefore, $t = t'$, and we set $\alpha = t$.

2. $\pi =$ VS-NEWVIEW$(v)_p$

Then $trace((s, \pi, s')) = \lambda$. Action π modifies cur_p, $\mathit{info\text{-}sent}[cur.id]_p$, and $\mathit{current\text{-}viewid}[p]$, and adds an "*info*" message to $\mathit{msgs\text{-}to\text{-}vs}[cur.id]_p$. The definition of $\mathcal{F}_{dvs}$ is not sensitive to any of these changes. Therefore, $t = t'$, and we set $\alpha = t$.

3. $\pi =$ VS-GPSND$(m)_p$

Then $trace((s, \pi, s')) = \lambda$. Action π just moves a message from the queue $\mathit{msgs\text{-}to\text{-}vs}[cur.id]_p$ to the queue $\mathit{pending}[p, \mathit{current\text{-}viewid}[p]]$. The definition of $\mathcal{F}_{dvs}$ is not sensitive to this change. Therefore, $t = t'$, and we set $\alpha = t$.

4. $\pi =$ VS-ORDER(m, p, g)

Then $trace((s, \pi, s')) = \lambda$. Action π moves a message from $\mathit{pending}[p, g]$ to $\mathit{queue}[g]$. We consider two cases.

(a) $m \in \mathcal{M}_c$

Then we set $\alpha = (t,$ DVS-ORDER$(m, p, g), t')$. We claim that DVS-ORDER(m, p, g) is enabled in t: Since VS-ORDER(m, p, g) is enabled in s, it follows that m is the head of $s.\mathit{pending}[p, g]$. By the definition of $\mathcal{F}_{dvs}$, m is also the head of $t.\mathit{pending}[p, g]$. It follows that DVS-ORDER(m, p, g) is enabled in t.

By definition of $\mathcal{F}_{dvs}$, t' differs from t only in the fact that m is moved from $\mathit{pending}[p, g]$ to $\mathit{queue}[g]$. This is the effect achieved by applying DVS-ORDER(m, p, g) to t.

(b) $m \notin \mathcal{M}_c$

Then the definition of $\mathcal{F}_{dvs}$ is not sensitive to this change. Therefore, $t = t'$, and we set $\alpha = t$.

5. $\pi =$ VS-GPRCV$(\langle \text{"}\mathit{info}\text{"}, v, s\rangle)_{q,p}$

Then $trace((s, \pi, s')) = \lambda$. This action can modify $\mathit{info\text{-}rcvd}[cur.id_p, q]_p$, act_p and amb_p (see code of DVS) and causes $\mathit{next}[p, cur.id_p]$ to be incremented (see code of VS). The definition of $\mathcal{F}_{dvs}$ is not sensitive to these changes. (The only interesting case is the definition of $t.\mathit{next}[p, cur.id_p]$, where the absolute values of the first two terms on the right-hand side are both increased by 1, but they cancel each other out.) Therefore, $t = t'$, and we set $\alpha = t$.

6. $\pi =$ VS-GPRCV$(\text{"}\mathit{registered}\text{"})_p$

Then $trace((s, \pi, s') = \lambda$. This action can modify $\mathit{rcvd\text{-}rgst}[cur.id, q]_p$. It also causes the pointer $\mathit{next}[p, cur.id_p]$ to be incremented. The definition of $\mathcal{F}_{dvs}$ is not sensitive to these changes. (The only interesting case is the definition of $t.\mathit{next}[p, cur.id_p]$, where the absolute values of the first two terms on the right-hand side are both increased by 1, but they cancel each other out.) Therefore, $t = t'$, and we set $\alpha = t$.

7. $\pi = \text{VS-GPRCV}(m)_p$, $m \in \mathcal{M}_c$

Then $trace((s, \pi, s')) = \lambda$. This action copies a message from the sequence $queue[cur.id]_p$ to the sequence $msgs\text{-}from\text{-}vs[p, client\text{-}cur[p]]$, and causes $next[p, cur.id_p]$ to be incremented. The definition of $\mathcal{F}_{dvs}$ is not sensitive to these changes. (The only interesting case is the definition of $t.next[p, cur.id_p]$, where the absolute values of the first and third terms on the right-hand side are both increased by 1, but they cancel each other out.) Therefore, $t = t'$, and we set $\alpha = t$.

8. $\pi = \text{VS-SAFE}(\langle m, v, s\rangle)_{q,p}$, $m \in \{\text{“}info\text{''}, \text{“}registered\text{''}\}$

Then $trace((s, \pi, s')) = \lambda$. Action π just causes $next\text{-}safe[p, cur.id_p]$ to be incremented. The definition of $\mathcal{F}_{dvs}$ is not sensitive to this change. (The only interesting case is the definition of $t.next\text{-}safe[p, cur.id_p]$, where the absolute values of the first two terms on the right-hand side are both increased by 1, but they cancel each other out.) Therefore, $t = t'$, and we set $\alpha = t$.

9. $\pi = \text{VS-SAFE}(m)_p$, $m \in \mathcal{M}_c$

Then $trace((s, \pi, s')) = \lambda$. Action π adds a message to $safe\text{-}from\text{-}vs[cur.id]_p$ and causes the pointer $next\text{-}safe[p, cur.id_p]$ to be incremented. The definition of $\mathcal{F}_{dvs}$ is not sensitive to these changes. (The only interesting case is the definition of $t.next\text{-}safe[p, cur.id_p]$, where the absolute values of the first and third terms on the right-hand side are both increased by 1, but they cancel each other out.) Therefore, $t = t'$, and we set $\alpha = t$.

10. $\pi = \text{DVS-NEWVIEW}(v)_p$

Then $trace((s, \pi, s)) = \pi$. In DVS-IMPL, this action modifies only variables amb_p, $attempted_p$, $client\text{-}cur_p$. We have $s'.client\text{-}cur_p = v$ and $s'.attempted_p = s.attempted_p \cup \{v\}$. By definition of $\mathcal{F}_{dvs}$, we have that $t'.current\text{-}viewid[p] = s'.client\text{-}cur.id_p = v.id$, $t'.created = t.created \cup \{v\}$ and $t'.attempted[v.id] = t.attempted[v.id] \cup \{p\}$, while all other state variables in t' are as in t. We consider two cases:

(a) $v \in t.created$.

In this case, we set $\alpha = (t, \pi', t')$, where $\pi' = \text{DVS-NEWVIEW}(v)_p$. The code shows that π' brings DVS-SYS from state t to state t'. It remains to prove that π' is enabled in state t, that is, that $v \in t.created$ and $v.id > t.current\text{-}viewid[p]$. The first of these two conditions is true because of the defining condition for this case. The second condition follows from the precondition of π in DVS-IMPL: this precondition implies that $v.id > s.client\text{-}cur.id_p$, and by the definition of $\mathcal{F}_{dvs}$ we have $t.current\text{-}viewid[p] = s.client\text{-}cur.id_p$.

(b) $v \notin t.created$.

In this case we set $\alpha = (t, \pi', t'', \pi'', t')$, where $\pi' = \text{DVS-CREATEVIEW}(v)_p$, $\pi'' = \text{DVS-NEWVIEW}(v)_p$, and t'' is the unique state that arises by running the effect of π' from t.

The code shows that α brings DVS-SYS from state t to state t'. It remains to prove that π' is enabled in t and that π'' is enabled in t''.

The precondition of π' requires that (i) $\forall w \in t.created$, $v.id \neq w.id$ and (ii) $\forall w \in t.created$, either $\exists x \in s.\mathcal{T}ot\mathcal{A}tt$ satisfying $w.id < x.id < v.id$ or $v.id < x.id < w.id$, or else $v.set \cap w.set \neq \emptyset$.

To see requirement (i), suppose for the sake of contradiction that $w \in t.created$ and $w.id = v.id$. The precondition of π in DVS-IMPL implies that $v = s.cur_p$, which implies that $v \in s.created$. Since $w \in t.created$, the definition of $\mathcal{F}_{dvs}$ implies that $w \in s.attempted_q$ for some q. This implies that $w \in s.created$. But then Invariant 4.1.1 implies that $v = w$. But this contradicts that fact that $v \notin t.created$ and $w \in t.created$.

To see requirement (ii), suppose that $w \in t.created$ and there is no $x \in s.\mathcal{T}ot\mathcal{A}tt$ satisfying $w.id < x.id < v.id$ or $v.id < x.id < w.id$. Since $w \in t.created$, by definition of $\mathcal{F}_{dvs}$, $w \in s.attempted_q$ for some q. Clearly, $w \in s'.attempted_p$. Therefore, $w \in s'.\mathcal{A}tt$. By the code of π we have that $v \in s'.attempted_q$. Therefore we also have $v \in s'.\mathcal{A}tt$. Moreover, there is no $x \in s'.\mathcal{T}ot\mathcal{A}tt$ satisfying $w.id < x.id < v.id$ or $v.id < x.id < w.id$. Then Invariant 5.2.17 implies that $v.set \cap w.set \neq \emptyset$, as needed to prove that π' is enabled in t.

We now prove that π'' is enabled in state t''. The precondition of π'' requires that $v \in t''.created$ and $v.id > t''.current\text{-}viewid[p]$. The first condition is true because v is added to *created* by π'. The second condition follows from the precondition of π in DVS-IMPL: The precondition of π implies that $v.id > s.client\text{-}cur.id_p$. The definition of $\mathcal{F}_{dvs}$ implies that $t.current\text{-}viewid[p] = s.client\text{-}cur.id_p$. Moreover, $t''.current\text{-}viewid[p] = t.current\text{-}viewid[p]$. It follows that $v.id > t''.current\text{-}viewid[p]$. Thus π'' is enabled in state t''.

11. $\pi =$DVS-REGISTER$_p$

Then $trace((s, \pi, s')) = \pi$. Let g be $s.client\text{-}cur.id_p$, which equals $t.current\text{-}viewid[p]$ by the abstraction function. If $g = \perp$, then π has no effect in DVS-IMPL, so $s = s'$; thus $t = t'$, as required to show that π brings DVS from t to t'. Otherwise, $g \neq \perp$, so by the code in DVS-IMPL, this action sets $reg[g]_p$ to `true` and inserts a *"registered"* message into $msgs\text{-}to\text{-}vs[g]_p$. By definition of $\mathcal{F}_{dvs}$, t' is the same as t except that $t'.registered[g] = t.registered[g] \cup \{p\}$. We set $\alpha = (t, \text{DVS-REGISTER}_p, t')$. It is easy to check that DVS-REGISTER$_p$ brings DVS-SYS from t to t'.

12. $\pi =$DVS-GARBAGECOLLECT$(v)_p$

Then $trace((s, \pi, s')) = \lambda$. This action can modify act_p and amb_p. The definition of $\mathcal{F}_{dvs}$ is not sensitive to these changes. Therefore, $t = t'$, and we set $\alpha = t$.

13. $\pi =$DVS-GPSND$(m)_p$

Then $trace((s, \pi, s')) = \pi$. We set $\alpha = (t,$ DVS-GPSND$(m)_p, t')$. We consider two cases:

(a) $s.client\text{-}cur.id = \perp$

Then $s = s'$. In this case, the definition of $\mathcal{F}_{dvs}$ implies that also $t.current\text{-}viewid[p] = \perp$, which implies that the action also has no effect in t, which suffices.

(b) $s.client\text{-}cur.id \neq \perp$

In this case, the action appends m to $msgs\text{-}to\text{-}vs[g]_p$, where $g = client\text{-}cur.id_p$. Hence we have that $s'.msgs\text{-}to\text{-}vs[g] = s.msgs\text{-}to\text{-}vs[g] \circ m$. By the definition of $\mathcal{F}_{dvs}$ we get that $t'.pending[p, g] = t.pending[p, g] \circ m$. This is the effect of the action in t (using the fact that $t.current\text{-}viewid[p] \neq \perp$.)

14. $\pi =$ DVS-GPRCV$(m)_p$

Then $trace((s, \pi, s')) = \pi$. This action removes the head of $msgs\text{-}from\text{-}vs[g]_p$, where $g = cur.id_p$. We have that $s.msgs\text{-}from\text{-}vs[g]_p = m \circ s'.msgs\text{-}from\text{-}vs[g]_p$. Thus $t'.next[p, g] = t.next[p, g] + 1$. We set $\alpha = (t,$ DVS-GPRCV$(m)_p, t')$. It is easy to check that the step has the required effect in DVS-SYS. The fact that DVS-GPRCV$(m)_p$ is enabled in t follows from Invariant 5.2.19.

15. $\pi =$ DVS-SAFE$(m)_p$

Then $trace(\pi) = \pi$. This action removes the head of the $safe\text{-}from\text{-}vs[g]_p$, where $g = cur.id_p$. We have that $s.safe\text{-}from\text{-}vs[g]_p = m \circ s'.safe\text{-}from\text{-}vs[g]_p$. Thus $t'.next\text{-}safe[p, g] = t.next\text{-}safe[p, g] + 1$. We set $\alpha = (t,$ DVS-GPRCV$(m)_p, t')$. It is easy to check that the step has the required effect in DVS-SYS. The fact that DVS-GPRCV$(m)_p$ is enabled in t follows from Invariant 5.2.21.

$\square$

Lemmas 5.2.22 and 5.2.23 prove that $\mathcal{F}_{dvs}$ is an abstraction function from DVS-IMPL to DVS and thus the following theorem holds.

Theorem 5.2.24 *Every trace of* DVS-IMPL *is a trace of* DVS-SYS.

5.3 An application of DVS

In this section we show how to use DVS to implement a totally ordered broadcast service, called TO. In Section 5.3.1 we give the specification of the totally ordered broadcast service TO, in Section 5.3.2 we describe the implementation, which we call TO-IMPL, and in Section 5.3.3 we prove that TO-IMPL, implements TO.

5.3.1 The TO service

The TO service was originally defined in [41]. This service accepts messages from clients and delivers them to all clients according to the same total order. This kind of service is a building block for many fault-tolerant distributed applications. The specification is reproduced in Figure 5-5.

The following is an informal description of the service. Processes can broadcast messages by means of actions $\text{BCAST}(a)_p$. Such a message a is appended to a queue local to process p, $pending[p]$. The service establishes a totally order on the messages by means of action $\text{TO-ORDER}(a, p)$, which takes a message from the local queue of a process and puts it into a global *queue*. The order established by this global queue is the one used to deliver messages. The pointer $next[q]$ points to the next message in the global *queue* to be delivered to process q by means of action $\text{BRCV}(a)_{p,q}$.

TO

Signature:

Input: $\text{BCAST}(a)_p$, $a \in \mathcal{A}$, $p \in \mathcal{P}$
Internal: $\text{TO-ORDER}(a, p)$, $a \in \mathcal{A}$, $p \in \mathcal{P}$
Output: $\text{BRCV}(a)_{p,q}$, $a \in \mathcal{A}$, $p, q \in \mathcal{P}$

State:

$queue \in seqof(\mathcal{A} \times \mathcal{P})$, init λ

for each $p \in \mathcal{P}$: $pending[p] \in seqof(\mathcal{A})$, init λ
$next[p] \in \mathbf{N}^{>0}$, init 1

Transitions:

input $\text{BCAST}(a)_p$
Eff: append a to $pending[p]$

internal $\text{TO-ORDER}(a, p)$
Pre: a is head of $pending[p]$
Eff: remove head of $pending[p]$
append $\langle a, p\rangle$ to *queue*

output $\text{BRCV}(a)_{p,q}$
Pre: $queue(next[q]) = \langle a, p\rangle$
Eff: $next[q] := next[q] + 1$

Figure 5-5: The TO service.

5.3.2 The implementation of TO

We provide an implementation of TO using DVS as a building block. The implementation, which we call TO-IMPL, consists of an automaton DVS-TO-TO_p for each $p \in \mathcal{P}$, and the DVS specification.

The implementation is similar to the TO implementation provided in [41]. Both algorithms rely on primary views to establish a total order of client messages. The main difference is that the algorithm in [41] uses a static notion of primary and the new one uses a dynamic notion. The algorithm of [41] is built upon a VS service that reports non-primary as well as primary views, and uses a simple local test to determine if the view is primary. That algorithm does some non-critical background work (gossiping information) in non-primary views. In contrast, the algorithm proposed here is built upon the DVS service, which only reports primary views. Thus the new algorithm is simpler in that it does not perform the local tests and does not carry out any processing in non-primary views. On the other hand, in the new algorithm, the application programs must perform DVS-REGISTER actions to tell the DVS service when they have "established" new views. Although the new algorithm appears very similar to the one of [41], the fact that the DVS service provides weaker and more complicated guarantees than the VS service makes the new algorithm harder to prove correct.

The TO-IMPL algorithm involves *normal* and *recovery* activity. Normal activity occurs while a group view is not changing. Recovery activity begins when a new primary view is presented by DVS, and continues while the members combine information from their previous history, to provide a consistent basis for ongoing normal activity.

During normal activity, each client message received by TO-IMPL is given a system-wide unique *label*, which consists of a view identifier (the one of the view in which the message is received), a sequence number and the identifier of the process receiving the message. The association between client messages and their unique labels is recorded in a relation *content* and communicated to other processes in the same view using DVS. When a message is received, the label is given an *order*, a tentative position in the system-wide total order the service is to provide. When client messages have been reported as delivered to all the members of the view, by the "safe" notification of DVS, the label and its order may become *confirmed*. The messages associated with confirmed labels may be released to the clients in the given order.

The consistent sequence of message delivery within each view keeps this tentative order consistent at members of a given view, but it may be not consistent between nodes in different views. To avoid inconsistencies processes need state exchange at the beginning of a new view.

When a new primary view is reported by DVS, recovery activity occurs to integrate the knowledge of different members. First, each member of a new view sends a message, using DVS, that contains a summary of that node's state. The summary of a node's state contains the following information: the association of labels with client messages, stored in *content*, the order of client messages to be reported to the clients, stored in *order*, a pointer to the next client message to be confirmed, stored in *nextconfirm* and the view identifier of the primary view with the highest view identifier in which the *order* sequence has been modified (stored in *highprimary*).

Once a node has received all members' state summaries, it processes the information in one atomic step, i.e., it *establishes* the new view. Once a node establishes a view, it informs DVS of that fact with a DVS-REGISTER action. The node processes state information as follows: it defines its confirmed labels to be the longest prefix of confirmed labels known in any of the summaries; it determines the *representatives* as the members whose summary include the greatest *highprimary* value; adopts as its new *order* the *order* of a "chosen" representative (the chosen representative is arbitrary but must be the same for all processes) extended with all other labels appearing in any of the received summaries, arranged in label order.

Then recovery continues by collecting the DVS safe indications. Once the state exchange is safe, all labels used in the exchange are marked as safe and all associated messages are confirmed just as in normal processing.

DVS-TO-TO	
Signature:	
Input: BCAST$(a)_p$, $a \in \mathcal{A}$ DVS-GPRCV$(m)_{q,p}$, $q \in \mathcal{P}$, $m \in \mathcal{C} \cup \mathcal{S}$ DVS-SAFE$(m)_{q,p}$, $q \in \mathcal{P}$, $m \in \mathcal{C} \cup \mathcal{S}$ DVS-NEWVIEW$(v)_p$, $v \in \mathcal{V}$	Output: DVS-REGISTER$_p$ DVS-GPSND$(m)_p$, $m \in \mathcal{C} \cup \mathcal{S}$ BRCV$(a)_{q,p}$, $a \in \mathcal{A}$, $q \in \mathcal{P}$ Internal: CONFIRM$_p$
State:	
$current \in \mathcal{V}_\perp$, init v_0 if $p \in P_0$, $\perp$ else $status \in \{normal, send, collect\}$, init $normal$ $content \in 2^{\mathcal{C}}$, init $\emptyset$ $nextseqno \in \mathbf{N}^{>0}$, init 1 $buffer \in seqof(\mathcal{L})$, init λ $safe\text{-}labels \in 2^{\mathcal{L}}$, init $\emptyset$ $order \in seqof(\mathcal{L})$, init λ $nextconfirm \in \mathbf{N}^{>0}$, init 1	$nextreport \in \mathbf{N}^{>0}$, init 1 $highprimary \in \mathcal{G}$, init g_0 if $p \in P_0$, $\perp$ else $gotstate$, a partial function from $\mathcal{P}$ to $\mathcal{S}$, init $\emptyset$ $safe\text{-}exch \subseteq \mathcal{P}$, init $\emptyset$ $registered \subseteq \mathcal{G}$, init $\{g_0\}$ if $p \in P_0$, $\emptyset$ else $delay \in seqof(\mathcal{A})$, init λ for each $g \in \mathcal{G}$, $established[g]$, a bool, init `true` if $g = g_0, p \in P_0$, `false` else

Figure 5-6: The DVS-TO-TO$_p$ code.

For the code, shown in Figures 5-6 and 5-7, we need the following definitions. $\mathcal{L} = \mathcal{G} \times \mathbf{N}^{>0} \times \mathcal{P}$ is the set of *labels*, with selectors $l.id$, $l.seqno$ and $l.origin$. $\mathcal{A}$ is the set of messages that can be sent by the clients of the TO service. $\mathcal{C} = \mathcal{L} \times \mathcal{A}$ is the set of possible associations between labels and client messages. $\mathcal{S} = 2^{\mathcal{C}} \times seqof(\mathcal{L}) \times \mathbf{N}^{>0} \times \mathcal{G}$ is the set of *summaries*, with selectors $x.con$, $x.ord$, $x.next$ and $x.high$. Given $x \in \mathcal{S}$, $x.confirm$ is the prefix of $x.ord$ such that $|x.confirm| = min(x.next - 1, |x.ord|)$. If Y is a partial function from processor ids to summaries, then we define:
$knowncontent(Y) = \cup_{q \in dom(Y)} Y(q).con$,
$maxprimary(Y) = \max_{q \in dom(Y)}\{Y(q).high\}$,
$maxnextconfirm(Y) = \max_{q \in dom(Y)} Y(q).next$,
$reps(Y) = \{q \in dom(Y) : Y(q).high = maxprimary\}$,
$chosenrep(Y) =$ some element in $reps(Y)$,
$shortorder(Y) = Y(chosenrep(Y)).ord$, and
$fullorder(Y) = shortorder(Y)$ followed by the remaining elements of $dom(knowncontent(Y))$, in label order.

The system TO-IMPL is the composition of all the DVS-TO-TO$_p$ automata and DVS with all the external actions of DVS hidden.

The *allstate*, *allcontent* and *allconfirm* derived variables are defined for TO-IMPL as follows (this is as in [41]).

We write $allstate[p, g]$ to denote a set of summaries, defined so that $x \in allstate[p, g]$ if and only if at least one of the following hold:

1. $current.id_p = g$ and $x = \langle content_p, order_p, nextconfirm_p, highprimary_p \rangle$.

2. $x \in pending[p, g]$.

3. $\langle x, p \rangle \in queue[g]$.

DVS-TO-TO (cont'd)

Transitions:

input BCAST$(a)_p$
Eff: append a to *delay*

internal LABEL$(a)_p$
Pre: a is head of *delay*
$current \neq \perp$
Eff: let l be $\langle current.id, nextseqno, p\rangle$
$content := content \cup \{\langle l, a\rangle\}$
append l to *buffer*
$nextseqno := nextseqno + 1$
delete head of *delay*

output DVS-GPSND$(\langle l, a\rangle)_p$
Pre: $status = normal$
l is head of *buffer*
$\langle l, a\rangle \in content$
Eff: delete head of *buffer*

input DVS-GPRCV$(\langle l, a\rangle)_{q,p}$
Eff: $content := content \cup \{\langle l, a\rangle\}$
$order := order \circ l$

input DVS-SAFE$(\langle l, a\rangle)_{q,p}$
Eff: $safe\text{-}labels := safe\text{-}labels \cup \{l\}$

internal CONFIRM$_p$
Pre: $order(nextconfirm) \in safe\text{-}labels$
Eff: $nextconfirm := nextconfirm + 1$

output BRCV$(a)_{q,p}$
Pre: $nextreport < nextconfirm$
$\langle order(nextreport), a\rangle \in content$
$q = order(nextreport).origin$
Eff: $nextreport := nextreport + 1$

input DVS-NEWVIEW$(v)_p$
Eff: $current := v$
$nextseqno := 1$
$buffer := \lambda$
$gotstate := \emptyset$
$safe\text{-}exch := \emptyset$
$safe\text{-}labels := \emptyset$
$status := send$

output DVS-GPSND$(x)_p$
Pre: $status = send$
$x = \langle content, order, nextconfirm, highprimary\rangle$
Eff: $status := collect$

input DVS-GPRCV$(x)_{q,p}$
Eff: $content := content \cup x.con$
$gotstate := gotstate \oplus \langle q, x\rangle$
if $(dom(gotstate) = current.set) \wedge (status = collect)$ then
$nextconfirm := maxnextconfirm(gotstate)$
$order := fullorder(gotstate)$
$highprimary := current.id$
$status := normal$
$established[current.id] := true$

output DVS-REGISTER$_p$
Pre: $current \neq \perp$
$established[current.id]$
$current.id \notin registered$
Eff: $registered := registered \cup \{current.id\}$

input DVS-SAFE$(x)_{q,p}$
Eff: $safe\text{-}exch := safe\text{-}exch \cup \{q\}$
if $safe\text{-}exch = current.set$ then
$safe\text{-}labels := safe\text{-}labels \cup range(fullorder(gotstate))$

Figure 5-7: The DVS-TO-TO$_p$ code (cont'd).

4. For some q, $current.id_q = g$ and $x = gotstate(p)_q$.

Thus, $allstate[p, g]$ consists of all the summary information that is in the state of p if p's current view is g, plus all the summary information that has been sent out by p in state exchange messages in view g and is now remembered elsewhere among the state components of TO-IMPL. Notice that $allstate[p, g]$ consists only of summaries: an ordinary message $\langle l, a\rangle$ is never an element of $allstate[p, g]$. We write $allstate[g]$ to denote $\bigcup_{p\in P} allstate[p, g]$, and $allstate$ to denote $\bigcup_{g\in G} allstate[g]$.

We write $allcontent$ for $\bigcup_{x\in allstate} x.con \cup \{\langle l, a\rangle : \exists g, p : \langle\langle l, a\rangle, p\rangle \in range(queue[g]) \vee \langle l, a\rangle \in range(pending[p, g])\}$. This represents all the information available anywhere that links a label with a corresponding data value.

We write $allconfirm$ for $lub_{x\in allstate}(x.confirm)$.

For every $p \in P$, $g \in G$, $buildorder[p, g]$ is defined to be a sequence of labels, initially empty; this variable is maintained by following every statement of processor p that assigns to $order$ with another statement $buildorder[p, current.id_p] := order$. It follows that if p establishes a view with id g, and later leaves view g for a view with a higher view identifier, then forever afterwards, $buildorder[p, g]$ remembers the value of $order_p$ at the point where p left view g.

5.3.3 Proof that TO-IMPL implements TO

The correctness proof for TO-IMPL follows the outline of the one in [41]. The main difference is that the main invariant, which corresponds to Lemma 6.18 of [41], requires a different, more subtle proof. We first provide some auxiliary invariants.

Invariant 5.3.1 (TO-IMPL)
In any reachable state, if $p \in$ DVS.$registered[g]$ then $established[g]_p$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. In the initial state, DVS.$registered[g]$ is empty for all g except for $g = g_0$ for which DVS.$registered[g_0] = P_0$. So assume that $g = g_0$ and $p \in P$. In the initial state $established[g]_p = \texttt{true}$ if $g = g_0$ and $p \in p_0$. Hence the invariant is true.

For the inductive step, assume that the invariant is true in a reachable state s. We need to prove that it is true in s' for any possible step (s, π, s'). Fix g and p. The hypothesis changes from false to true only in π=DVS-REGISTER$_p$ and $s.current_p = g$, and the action's precondition (in DVS-TO-TO) shows the conclusion. The conclusion never changes from true to false. ∎

Invariant 5.3.2 says that any view that is known (anywhere in the system state) to be an established primary was in fact attempted by all its members.

Invariant 5.3.2 (TO-IMPL)
In any reachable state, if $x \in allstate$ then there exists $w \in created$ such that $x.high = w.id$, and for all $p \in w.set$, $p \in attempted[w.id]$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. In the initial state, the invariant follows from the definition of *allstate* (set $w = current.id_p$).
For the inductive step, assume that the invariant is true in a reachable state s. We need to prove that it is true in s' for any possible step (s, π, s'). The only step that we have to worry about is when a new summary is created. When a new summary x is created, $x.high$ is set to the identifier of the current view, and a message has been received from everyone in the membership. ∎

Invariant 5.3.3 says that if a view w is established, then no earlier view v can still be active.

Invariant 5.3.3 (TO-IMPL)
In any reachable state, if $v \in created$, $x \in allstate$ and $x.high > v.id$ then there exists $p \in v.set$ with $current.id_p > v.id$.

Proof: Fix v, x as given. Lemma 5.3.2 shows the existence of $w \in \mathit{created}$ such that $x.\mathit{high} = w.\mathit{id}$, and for all $p \in w.\mathit{set}$, $p \in \mathit{attempted}[w.\mathit{id}]$. Then Invariant 5.1.4 implies that there exists $p \in v.\mathit{set}$ with $\mathit{current\text{-}viewid}[p] > v.\mathit{id}$. But $\mathit{current\text{-}viewid}[p] = \mathit{current}.\mathit{id}_p$, which yields the result. ∎

Finally we provide the proof for the invariant corresponding to the invariant stated in Lemma 6.18 of [41]. This invariant has a more subtle proof than the one given in Lemma 6.18 of [41]. That proof uses the strong intersection property among primary view membership (in the implementation of [41] each primary view intersects each other primary view). The proof in [41] does not work in the setting of DVS because DVS guarantees a weaker intersection property (each primary view intersects only the primary views in between the preceding and the following totally registered primary views). The new proof also uses the fact about DVS that once a view is attempted at all processes in its membership set, no views with lower identifiers can become established.

Invariant 5.3.4 (TO-IMPL)
In any reachable state, suppose that $v \in \mathit{created}$, $\sigma \in \mathit{seqof}(\mathcal{L})$, and for every $p \in v.\mathit{set}$, the following is true: If $\mathit{current}.\mathit{id}_p > v.\mathit{id}$ then $\mathit{established}[v.\mathit{id}]_p$ and $\sigma \leq \mathit{buildorder}[p, v.\mathit{id}]$.
Then for every $x \in \mathit{allstate}$ with $x.\mathit{high} > v.\mathit{id}$, we have that $\sigma \leq x.\mathit{ord}$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. In the initial state, the only created view is v_0, and there is no $x \in \mathit{allstate}$ with $x.\mathit{high} > g_0$. So the statement is vacuously true.

For the inductive step, assume that the invariant is true in a reachable state s. We need to prove that it is true in s' for any possible step (s, π, s'). So fix $v \in s'.\mathit{created}$ and σ, and assume that for every $p \in v.\mathit{set}$, if $s'.\mathit{current}.\mathit{id}_p > v.\mathit{id}$ then $s'.\mathit{established}[v.\mathit{id}]_p$ and $\sigma \leq s'.\mathit{buildorder}[p, v.\mathit{id}]$.

If $v \notin s.\mathit{created}$, then π must be CREATEVIEW(v). Then $s'.\mathit{established}[v.\mathit{id}]_p = \texttt{false}$ for all p. Fix $x \in s'.\mathit{allstate}$ and suppose that $x.\mathit{high} > v.\mathit{id}$. Then Invariant 5.3.3 applied to s' implies that there exists $p \in v.\mathit{set}$ with $s'.\mathit{current}.\mathit{id}_p > v.\mathit{id}$; fix such a p. Then the hypothesis part of the invariant for s' implies that $s'.\mathit{established}[v.\mathit{id}]_p = \mathit{true}$, a contradiction. It follows that $v \in s.\mathit{created}$.

As usual, the interesting steps are those that convert the hypothesis from false to true, and those that keep the hypothesis true while converting the conclusion from true to false.

In this case, there are no steps that convert the hypothesis from false to true: If there is some $p \in v.\mathit{set}$ for which $s.\mathit{current}.\mathit{id}_p > v.\mathit{id}$ and either $s.\mathit{established}[v.\mathit{id}]_p = \texttt{false}$ or σ is not a prefix of $s.\mathit{buildorder}[p, v.\mathit{id}]$, then also $s'.\mathit{current}.\mathit{id}_p > v.\mathit{id}$ (the id never decreases) and either $s'.\mathit{established}[v.\mathit{id}]_p = \texttt{false}$ or σ is not a prefix of $s'.\mathit{buildorder}[p, v.\mathit{id}]$. (These two cases carry over, since $s.\mathit{current}.\mathit{id}_p > v.\mathit{id}$ implies that $\mathit{established}[v.\mathit{id}]_p$ and $\mathit{buildorder}[p, v.\mathit{id}]$ cannot change.)

So it remains to consider any steps that keep the hypothesis true while converting the conclusion from true to false. Thus, we assume that if $s.\mathit{current}.\mathit{id}_p > v.\mathit{id}$ then $s.\mathit{established}[v.\mathit{id}]_p$ and $\sigma \leq s.\mathit{buildorder}[p, v.\mathit{id}]$. Suppose that $x \in s'.\mathit{allstate}$ and $x.\mathit{high} > v.\mathit{id}$. If also $x \in s.\mathit{allstate}$ then we can apply the inductive hypothesis, which implies that $\sigma \leq x.ord$, as needed. So the only concern is with steps that produce a new summary.

Any step that produces the new summary x by modifying an old summary $x' \in s.allstate$, in such a way that $x'.ord \leq x.ord$ and $x'.high = x.high$, is easy to handle: For such a step, $x'.high > v.id$ and so the inductive hypothesis implies that $\sigma \leq x'.ord \leq x.ord$, as needed. So the only concern is with GPRCV_p steps that produce a new summary x by delivering the last state-exchange message of a view w to some processor p. Thus $x.high = w.id$. Let x' be the summary of $q' = chosenrep$ in $s'.gotstate$. We claim $x'.high \geq v.id$.

To prove the claim, we let v' denote the unique element with highest viewid among the elements of $s'.created$ such that $v'.id < w.id$ and $s'.registered[v'.id] = v'.set$. Let v'' denote either v' or v, whichever has the higher viewid. Invariant 5.1.3 shows that $w.set \cap v''.set \neq \emptyset$, no matter whether v'' is v or v'. Fix any element q'' in $w.set \cap v''.set$.

Recall that the condition for establishing a view shows that $domain(s'.gotstate_p) = w.set$, so by the code, either $q'' \in domain(s.gotstate_p)$ or else q'' is the sender of the message whose receipt is the step we are examining. In the former case, let x'' be the summary $s.gotstate(q'')_p$; in the latter let x'' be the summary whose receipt is the event. In either case we have $x'' \in s.allstate[q'', w.id]$.

We now show that $s.established[v''.id]_{q''}$. We consider two cases:

1. $v'' = v'$.

   Then $q'' \in v'.set$ so by definition of v', we obtain $q'' \in s.registered[v'.id]$; therefore, we have that $s.established[v'.id]_{q''}$.

2. $v'' = v$. Because $s.allstate[q'', w.id]$ is non-empty, the analogue of Part 4 of Lemma 6.7 from [41] implies that $s.current.id_{q''} \geq w.id$. We have that $x.high > v.id$ by assumption, and $x.high = w.id$ by the code; therefore, $w.id > v.id$. So also $s.current.id_{q''} > v.id$. Recall that we are in the case where the hypothesis of this lemma is true. Therefore, by this hypothesis (uses $q'' \in v.set$), we obtain that $s.established[v.id]_{q''}$

By the analogue of Lemma 6.14 from [41], (applied with q'' replacing p) we obtain $x''.high \geq v''.id$. By the definition of q' as a member that maximizes the *high* component in the summary recorded in $s'.gotstate$, we have $x'.high \geq x''.high$. Therefore $x'.high \geq v''.id \geq v.id$, completing our proof of the claim.

If $x'.high > v.id$ then we can apply the induction hypothesis to x' and we are done, since $x'.ord \leq x.ord$. So suppose $x'.high = v.id$. Note that $x' \in s.allstate[q', w.id]$. By an analogue of Lemma 6.16 from [41] there must exist[1] $q \in v.set$ so that $s.established[v.id]_q$, $x'.ord = s.buildorder[q, v.id]$, and (either $x'.high = w.id$ or $s.current.id_q > v.id$). Since $x'.high = v.id < x.high = w.id$, the

[1]Direct application of the lemma actually shows the existence of some $\hat{v}$ and $q \in \hat{v}.set$, but since $x'.high = \hat{v}.id$ and also $x'.high = v.id$, uniqueness of viewids shows we may take $\hat{v}$ to be v itself.

last property can be simplified to $s.current.id_q > v.id$. By monotonicity of *current*, we have $s'.current_q > v.id$. The hypothesis of this lemma says that this forces $\sigma \leq s'.buildorder[q, v.id]$. Since $x'.ord \leq x.ord$ by the code for this event, and $x'.ord = s.buildorder[q, v.id]$ as shown above, and $s.buildorder[q, v.id] = s'.buildorder[q, v.id]$ since q is not currently in view v, we get $\sigma \leq x.ord$, which is what we need. ∎

Let s be a state of TO-IMPL. The state $t = \mathcal{F}_{to}(s)$ of TO is the following.

1. $t.queue = applyall(\langle s.allcontent, origin \rangle, s.allconfirm)$,
   where the selector *origin* is regarded as a function from labels to processors.
2. $t.next[p] = s.next\text{-}report_p$.
3. $t.pending[p] = applyall(s.allcontent, b) \cdot s.delay_p$ where b is the sequence of labels such that
   (a) $range(b)$ is the set of labels l such that $l.origin = p$, $\langle l, a \rangle \in s.allcontent$ for some a, and
   $l \notin range(s.allconfirm)$.
   (b) b is ordered according to the label order.

Figure 5-8: The abstraction function $\mathcal{F}_{to}$.

To complete the implementation proof, we define a function from the reachable states of TO-IMPL to the states of TO and prove that it is an abstraction function. This function is defined exactly as in [41]. Figure 5-8 shows the abstraction function $\mathcal{F}_{to}$. The proof that $\mathcal{F}_{to}$ is an abstraction function is as in [41]. Since $\mathcal{F}_{to}$ is an abstraction function we have the following theorem.

Theorem 5.3.5 *Every trace of* TO-IMPL *is a trace of* TO.

5.4 Remarks

The safe indications provided by the DVS service are crucial to the application: commitments about the total order are made only when receiving safe notifications for particular messages. This is a common point for any application that needs to preserve strong data coherence. For such applications, no commitments about the shared data can be made before safe indications are delivered. However the application can still perform some useful work while waiting for a safe indication. For example, it can pre-compute the value of the shared data so that when the safe indications arrive little processing will be needed (of course such computation is wasted if the safe indication never arrives); it can do optimistic updates to the shared data assuming that the safe indications will arrive (in this case roll back is required if the safe indications do not arrive).

The total order service that we have developed in this chapter can be built also using a sequence of executions of a consensus algorithms (e.g., the MULTIPAXOS algorithm of [61][2]). The advantage

[2]The name MULTIPAXOS is actually used in [29]. The original paper by Lamport [61] uses a different name (multi-

of the approach taken here is that we use a building block, the DVS group communication service, which may also be used for other applications. For example, in the next chapter, we will use a variant of DVS to build two applications on top of it, an atomic multi-reader multi-writer shared register and a dynamic version of the PAXOS algorithm.

This work deals entirely with safety properties; it remains to consider performance and fault-tolerance properties as well. Future work also include investigation of other applications of our DVS specification, such as replicated data applications and load-balancing applications.

Another interesting exploration direction considers variations on the DVS specification, for example, one in which the state exchange at the beginning of a new view is supported by the dynamic view service. Also, one could provide variations on our specifications that are more specifically tuned to systems like Isis and Ensemble. For example, a variation could require that processes that move together from one view to the next receive exactly the same messages in the first view. This guarantee is offered by Isis and Ensemble.

In the next chapter we provide a generalization of the DVS service to configurations. This generalization will also include support for state transfer at the beginning of a new configuration.

decree parliament protocol).

Chapter 6

The DC service

In this chapter we generalize the notion of dynamic primary view to that of *dynamic primary configuration*. We present DC, a specification for a dynamic primary configuration group communication service. Section 6.1 provides the DC specification, Section 6.4 provides an implementation of DC, and finally Section 6.5 describes an application that uses DC as a building block.

6.1 Overview

The DC specification is similar to the DVS specification; the difference is that it provides the clients with configurations instead of views. Like DVS, the DC specification is dynamic and provides primary configurations. The main difficulty here is that a notion of *dynamic primary configuration* needs to be developed (the notion of dynamic primary view has been studied in several papers, e.g. [55, 89]). In this chapter we develop such a notion and we define the DC service, which provides dynamic primary configurations to its clients.

Primary configurations must satisfy certain intersection properties with previous primary configurations. The type of configurations that we consider in this chapter is the read-write-quorum configurations (see Section 3.1.2). The intersection property that we require is that that the membership set of a new primary configuration must include the members of at least one read quorum and one write quorum of the previous primary configuration. The DC specification provides to the client only configurations satisfying this property.

Change of configurations might be driven either by change in the underlying physical distributed system or by the applications running on top of the system (e.g., a new quorum system could be installed on the same membership set).

An important feature of the DC specification is that it incorporates a state-exchange at the beginning of a new primary configuration. State-exchange at the beginning of a new configuration is required by most applications. When a new configuration is issued each member of the configuration is supposed to submit its current state to the service. Once having obtained the state from all the members of the configuration, the DC service computes the most up to date state over all the members, called the *starting state*. The starting state is then delivered to each member of the configuration. This way, each member begins regular computation in the new configuration knowing the starting state. We remark that this is different from the approach used by the DVS service which lets the members of the configuration compute the starting state. Some existing group communication services also integrate state-exchange within the service, e.g., [82, 19, 86], some others do not, e.g., [33, 73, 38, 41]. The Transis system [33] can be augmented with a layer providing state-exchange [5].

The DC specification offers a broadcast/convergecast communication mechanism. This mechanism involves all the members of a quorum, and uses a *condenser function* to process the information gathered from the quorum [66]. More specifically, a client that wants to send a message to the members of its current configuration submits the message together with a condenser function to the service; then the DC service broadcasts the message to all the members of the configuration and waits for a response from a quorum (the type of the quorum, read or write, is also specified by the client); once answers are received from a quorum, the DC service applies the condenser function to these answers in order to compute a response to give back to the client that sent the message. Such a series of actions should be seen as performing an *operation* requested by the client; executing the operation requires the participation of a quorum of the processes.

We remark that this kind of communication is different from those of the VS service [41] and the DVS service. Instead, it is like the one used in [66]. We integrate it into DC because we want to develop a particular application that benefits from this particular communication service (a read/write register as is done in [66]).

6.2 The DC specification

Prior to providing the code for the DC specification, we need some notation and definitions, which we introduce in the following.

Let *OID* be a set of operation identifiers, partitioned into sets OID_p, $p \in \mathcal{P}$. We denote by $\mathcal{M}_c \subseteq \mathcal{M}$ the set of messages that clients may use for communication.

Let $\mathcal{A}$ be a set of "acknowledgment" values and let $\mathcal{R}$ be a set of "response" values. A *condenser function* is a function from $(\mathcal{P} \rightarrow \mathcal{A}_\perp)$ to $\mathcal{R}$. Let Φ be the set of all condenser functions. Let $\mathcal{S}$ be the set of all possible states of the clients (a state of $\mathcal{S}$ does not need to be the entire client's state, but it may contain only the relevant information in order for the application to work). The DC specification uses a condenser function also to compute the starting state of a new configuration; hence we assume that $\mathcal{S} \subseteq \mathcal{A}$ and also $\mathcal{S} \subseteq \mathcal{R}$. Given a function $f : \mathcal{P} \rightarrow D$ from the set of processes

$\mathcal{P}$ to some domain D and given a subset $P \subseteq \mathcal{P}$, we write $f|P$ to denote the function $f' : P \to D$, defined as $f'(p) = f(p)$ for $p \in P$.

The following data type is used to describe operations: $\mathcal{D} = \mathcal{M} \times \Phi \times \{\text{``}read\text{''}, \text{``}write\text{''}\} \times 2^{\mathcal{P}} \times (\mathcal{P} \to \mathcal{A}_{\perp}) \times Bool$ and we let $\mathcal{O} = OID \to \mathcal{D}_{\perp}$. Given an operation descriptor, selectors for the components are msg, cnd, sel, dlv, ack, and rsp.

The code of the DC specification is given in Figure 6-1.

Next we provide remarks and an informal description of this code. We start with the derived variables.

A configuration $c \in \mathcal{A}tt$ is said to be *attempted*. For an attempted configuration c there exists at least one process p that has executed action $\textsc{newconf}(c)_p$ and thus we have that $p \in attempted[c.id]$; when this holds we say that *c is attempted at p* or that *p has attempted c*. A configuration $c \in \mathcal{T}ot\mathcal{A}tt$ is said to be *totally attempted*. A totally attempted configuration is a configuration that is attempted at all members of the configuration.

A configuration $c \in \mathcal{E}st$ is said to be *established*. For an established configuration c there exists at least one process p that has executed action $\textsc{newstate}(s)_p$ and thus we have that $p \in state\text{-}dlv[c.id]$; when this holds we say that *c is established at p* or that *p has established c*. A configuration $c \in \mathcal{T}ot\mathcal{E}st$ is said to be *totally established*. A totally established configuration is a configuration that is established at all members of the configuration.

A *dead* configuration c is a configuration for which a member process p went on to newer configurations, that is, it executed action $\textsc{newconf}(c')_p$ with $c'.id > c.id$, before receiving the notification, that is the $\textsc{newconf}(c)_p$ event, for configuration c.

Now we comment on the transitions.

Action $\textsc{createconf}(c)$ creates a new configuration c. The first precondition simply requires this new configuration to have a brand new identifier. The second precondition of this action is the key to our specification. It states that when a configuration c is created it must either be already dead or for any other configuration w such that there are no intervening totally established configurations, the earlier configuration (i.e., the one with smaller identifier) has at least one read quorum and one write quorum that are subsets of the membership set of the later configuration (i.e., the one with bigger identifier).

Action $\textsc{newconf}(c)_p$ delivers a created configuration c to the client process p. The precondition of this action makes sure that configurations are delivered in order of configuration identifiers. We notice that because of this precondition, when a configuration c is dead because a process q went on to newer configurations, we have that process q can no longer execute action $\textsc{newconf}(c)_q$.

Once a configuration c has been delivered to a client process p, the client process p is supposed to submit its current state s and a condenser function ϕ, by means of action $\textsc{submit-state}(s, \phi)_p$. Once all the processes have submitted their current states, the condenser function ϕ is used to compute

DC

Signature:

Input: $\text{SUBMIT}(m, \phi, b, i)_p$, $m \in \mathcal{M}_c$, $\phi \in \Phi$,
$b \in \{$"*read*", "*write*"$\}$, $p \in \mathcal{P}$, $i \in OID_p$
$\text{ACKDLVR}(a, i)_p$, $a \in A$, $i \in OID$, $p \in \mathcal{P}$
$\text{SUBMIT-STATE}(s, \phi)_p$, $s \in \mathcal{S}$, $\phi \in \Phi$

Internal: $\text{CREATECONF}(c)$, $c \in \mathcal{C}$
Output: $\text{NEWCONF}(c)_p$, $c \in \mathcal{C}$, $p \in c.set$
$\text{NEWSTATE}(s)_p$, $s \in \mathcal{S}$
$\text{RESPOND}(a, i)_p$, $a \in A$, $i \in OID_p$, $p \in \mathcal{P}$
$\text{DELIVER}(m, i)_p$, $m \in \mathcal{M}_c$, $i \in OID$, $p \in \mathcal{P}$

State:

$created \in 2^{\mathcal{C}}$, init $\{c_0\}$
for each $p \in \mathcal{P}$:
  $cur\text{-}cid[p] \in \mathcal{G}_\perp$, init g_0 if $p \in P_0$, $\perp$ else
for each $g \in \mathcal{G}$:
  $attempted[g] \in 2^{\mathcal{P}}$, init P_0 if $g = g_0$, $\{\}$ else

for each $g \in \mathcal{G}$:
  $got\text{-}state[g] \in \mathcal{P} \to \mathcal{S}_\perp$, init everywhere $\perp$
  $condenser[g] \in \Phi_\perp$, init everywhere $\perp$
  $state\text{-}dlv[g] \in 2^{\mathcal{P}}$, init P_0 if $g = g_0$, $\{\}$ else
  $pending[g] \in \mathcal{O}$, init everywhere $\perp$

Derived variables:

$\mathcal{A}tt \in 2^{\mathcal{C}}$, defined as $\{c \in created | attempted[c.id] \neq \emptyset\}$
$\mathcal{E}st \in 2^{\mathcal{C}}$, defined as $\{c \in created | state\text{-}dlv[c.id] \neq \emptyset\}$
$\mathcal{T}ot\mathcal{A}tt \in 2^{\mathcal{C}}$, defined as $\{c \in created | c.set \subseteq attempted[c.id]\}$
$\mathcal{T}ot\mathcal{E}st \in 2^{\mathcal{C}}$, defined as $\{c \in created | c.set \subseteq state\text{-}dlv[c.id]\}$
$dead \in 2^{\mathcal{C}}$ defined as $dead = \{c \in \mathcal{C} | \exists p \in c.set : cur\text{-}cid_p > c.id$ and $p \notin attempted[c.id]\}$.

Actions:

internal $\text{CREATECONF}(c)$
  Pre: $\forall w \in created : c.id \neq w.id$
    if $c \notin dead$ then
      $\forall w \in created, w.id < c.id$:
        $w \in dead$ or
        $(\exists x \in \mathcal{T}ot\mathcal{E}st$: $w.id < x.id < c.id)$ or
        $(\exists R \in w.rqrms, \exists W \in w.wqrms$:
          $R \cup W \subseteq c.set)$
      $\forall w \in created, w.id > c.id$
        $w \in dead$ or
        $(\exists x \in \mathcal{T}ot\mathcal{E}st$: $c.id < x.id < w.id)$ or
        $(\exists R \in c.rqrms, \exists W \in c.wqrms$:
          $R \cup W \subseteq w.set)$
  Eff: $created := created \cup \{c\}$

output $\text{NEWCONF}(c)_p$, $p \in c.set$
  Pre: $c \in created$
    $c.id > cur\text{-}cid[p]$
  Eff: $cur\text{-}cid[p] := c.id$
    $attempted[c.id] := attempted[c.id] \cup \{p\}$

input $\text{SUBMIT-STATE}(s, \phi)_p$
  Eff: if $cur\text{-}cid[p] \neq \perp$ and
      $got\text{-}state[cur\text{-}cid[p]](p) = \perp$ then
        $got\text{-}state[cur\text{-}cid[p]](p) := s$
        $condenser[cur\text{-}cid[p]](p) := \phi$

output $\text{NEWSTATE}(s)_p$ *choose* c
  Pre: $c.id = cur\text{-}cid[p]$
    $c \in created$
    $\forall q \in c.set$, $got\text{-}state[c.id](q) \neq \perp$
    let $f = condenser[c.id](p) | c.set$
    $s = f(got\text{-}state[c.id])$
    $p \notin state\text{-}dlv[c.id]$
  Eff: $state\text{-}dlv[c.id] := state\text{-}dlv[c.id] \cup \{p\}$

input $\text{SUBMIT}(m, \phi, b, i)_p$
  Eff: if $cur\text{-}cid[p] \neq \perp$ then
      $pending[cur\text{-}cid[p]](i)$
        $:= (m, \phi, b, \emptyset, \lambda(x) : x \to \perp, \texttt{false})$

output $\text{DELIVER}(m, i)_p$ *choose* g
  Pre: $g = cur\text{-}cid[p]$
    $p \notin pending[g](i).dlv$
    $pending[g](i).msg = m$
  Eff: $pending[g](i).dlv := pending[g](i).dlv \cup \{p\}$

input $\text{ACKDLVR}(a, i)_p$
  Eff: if $cur\text{-}cid[p] \neq \perp$ and
      $pending[cur\text{-}cid[p]](i).ack(p) \neq \perp$ then
        $pending[cur\text{-}cid[p]](i).ack(p) := a$

output $\text{RESPOND}(r, i)_p$ *choose* c, Q
  Pre: $c.id = cur\text{-}cid[p]$
    $c \in created$
    $i \in OID_p$
    $pending[c.id](i).rsp = \texttt{false}$
    if $pending[c.id].sel =$ "*read*"
      then $Q \in c.rqrms$
    else $Q \in c.wqrms$
    let $f = pending[c.id](i).ack$
    $\forall q \in Q : f(q) \neq \perp$
    $r = pending[c.id](i).cnd(f|Q)$
  Eff: $pending[c.id](i).rsp := \texttt{true}$

Figure 6-1: The DC specification

the starting state of configuration c for process p. The code of this action just memorizes the state s and the condenser function ϕ for the current configuration of process p.

Action NEWSTATE$(s)_p$ computes the starting state for a configuration c. The precondition of this action requires that all processes q in the membership of configuration c have submitted their state for configuration c. The starting state s of configuration c for process p is then computed by applying the condenser function that process p has submitted to the service with action SUBMIT-STATE$(s, \phi)_p$. Variable *state-dlv*$[c.d]$ records the fact that p has received the starting state for configuration

We remark that for a dead configuration c there is at least one process that does not execute action NEWCONF$(c)_p$ and thus does not submit its state for c with action SUBMIT-STATE$(s, \phi)_p$. This implies that action NEWSTATE$(s)_q$ cannot be executed for any process q. This is why such configurations are called "dead".

The remaining actions are used to handle the requests of clients. We refer to the process of handling such a request, which involves the participation of a quorum of processes, as an "operation". To request the execution of an operation a client process p uses action SUBMIT$(m, \phi, b, i)_p$. The parameters of this actions are as follows: m is a message describing the operation that p needs to perform (e.g., read a register, write a register); ϕ is a condenser function to be used to compute a response value for p when a quorum of processes have provided acknowledgment values to p's message m; b is just a selector indicating whether to wait for acknowledgment values from a write quorum or from a read quorum; i is an operation identifier needed to distinguish operations (every requested operation has a unique operation identifier). We say "operation i" to indicate the operation requested with action SUBMIT$(m, \phi, b, i)_p$. For configuration c and operation i, the variable *pending*$[c.id](i)$ contains an operation descriptor; The code of action SUBMIT$(m, \phi, b, i)_p$ sets to a default value this operation descriptor.

We now provide an explanation for each component of an operation descriptor. Let d be an operation descriptor for operation i requested by p in configuration c. $d.msg$ is the message that describes the request of p; such a message will be delivered to all members of the configuration c. $d.cnd$ is the condenser function that will be used to compute the response for the operation once a quorum of processes has provided acknowledgment values. $d.sel$ is the selector that specifies whether to use a read or a write quorum. $d.dlv$ is the set of processes to which the message $d.msg$ has been delivered; initially this is set to an empty set by action SUBMIT$(m, \phi, b, i)_p$. $d.ack$ contains the acknowledgment values received; initially this is a vector of $\perp$ values. Finally, $d.rsp$ is a flag indicating whether or not the client p, which requested the operation, has received a response for the operation.

Action DELIVER$(m, i)_p$ delivers the message m of operation i to process p. The code of this action updates the operation descriptor d for operation i by adding process p to the set $d.dlv$.

Processes that receive the message m for an operation i are supposed to provide an acknowledgment value a with action $\text{ACKDLVR}(a,i)_p$. The code of this action records the acknowledgment value a of process p into the vector $d.ack$, where d is the operation descriptor for operation i.

Finally, action $\text{RESPOND}(r,i)$ provides a response r to process p for the operation i previously submitted by p. The precondition of this action requires that a quorum Q has provided acknowledgment values (the type of the quorum depends on the selector provided at the time of the operation submission). Then the value r is computed by applying the condenser function provided by p at the time of the submission, to the acknowledgment values of processes in Q. At this point the operation has been serviced and the *rsp* component is set to `true`.

6.3 Invariants of DC

In this section we provide invariants of DC. These invariants are used to prove the correctness of the application that we build on top of DC.

Invariant 6.3.1 *In any reachable state of* DC*, the following is true. Let* $c_1, c_2 \in created \setminus dead$*, with* $c_1.id < c_2.id$*. Then either exists* $w \in \mathcal{T}ot\mathcal{E}st, c_1.id < w.id < c_2.id$*, or else there exist R, W, quorums of* c_1 *such that* $R \cup W \subseteq c_2.set$*.*

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. In the initial state the invariant is vacuously true because there are no two configurations $c_1, c_2 \in created$ such that $c_1.id < c_2.id$.
For the inductive step assume that the invariant is true in a state s. We need to prove that the invariant is true in s' for any step (s, π, s'). The only action that we need to worry about is $\text{CREATECONF}(c)$, where $c = c_1$ or $c = c_2$, because it creates a new configuration (otherwise the invariant is true in s' by the inductive hypothesis). So assume that $\pi = \text{CREATECONF}(c)$. The invariant follows easily from the precondition of π. ∎

We remark that the intersection property stated in the above invariant may not hold for dead configurations. However, in a dead configuration is not possible to make progress because for such a configuration there is at least one process that will not participate and thus the configuration will never become established.

The need for considering dead configurations comes from the implementation of the specification that we provide. It is possible to give a stronger version of DC by requiring that the intersection property in the precondition of action CREATECONF holds also for dead configurations. We do not know if this stronger version is implementable.

Invariant 6.3.2 *In any reachable state of* DC*, the following is true. If* $p \in attempted[g]$ *then* $cur\text{-}cid[p] \geq g$*.*

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. In the initial state we have that $attempted[g]$ is P_0 if $g = g_0$ and $\emptyset$ otherwise. Moreover for each $p \in P_0$ we have that $cur\text{-}cid[p] = g_0$. Hence the invariant is true.

For the inductive step fix g and p and assume that the invariant is true in a state s. We need to prove that the invariant is true in s' for any step (s, π, s'). The actions that can make the invariant false are those that either put p into $attempted[g]$ or modify $cur\text{-}cid[p]$. Hence we need only to worry about action NEWCONF(c). So assume that π=NEWCONF(c). The invariant follows easily from the precondition of π. ∎

Invariant 6.3.3 *In any reachable state of* DC*, the following is true. If $c \in created \setminus dead$, $w \in \mathcal{T}ot\mathcal{A}tt$, and $w.id > c.id$, then there exists $R \in c.rqrms$ such that for all $p \in R$, $cur\text{-}cid[p] > c.id$.*

Proof: Consider any particular reachable state. Assume that $c \in created \setminus dead$, $w \in \mathcal{T}ot\mathcal{A}tt$, and $w.id > c.id$. Then let y be the configuration in $\mathcal{T}ot\mathcal{A}tt$ having the smallest identifier strictly greater than $c.id$. Note that $y \notin dead$, since a dead configuration cannot be totally attempted. Then there is no $x \in \mathcal{T}ot\mathcal{E}st$ with $c.id < x.id < y.id$. Then Invariant 6.3.1 implies that for some $R \in c.rqrms$, $R \subseteq y.set$. Let p be any element of R. Since $y \in \mathcal{T}ot\mathcal{A}tt$ we have that $p \in attempted[y.id]$. By Invariant 6.3.2 we have $cur\text{-}cid[p] \geq y.id$. Since $y.id > c.id$ we have $cur\text{-}cid[p] > c.id$. ∎

6.4 An implementation of DC

In this section we provide an algorithm that implements, in the sense of trace inclusion, the DC specification. The algorithm is built on top of the CS service and uses ideas from [88].

6.4.1 Overview

The implementation of DC that we provide in this section is similar to the implementation of DVS provided in Chapter 5. However, there is a key difference in the implementation of DC compared to that of DVS. This difference provides new insights for the DVS specification and implementation, as we explain below.

The DVS specification requires a global intersection property which is the following: given two primary views w and v with no intervening totally established view[1], we must have that $w.set \cup v.set \neq \emptyset$. The DVS implementation, when delivering a new view v, checks *a stronger property* locally

[1]"Established" views are called "registered" views in Chapter 5. This is due to the fact that the DVS specification requires client processes to "register" a new view when they have obtained enough information to begin regular computation; in the DC service this is handled by the service itself. However the meaning of "established" is the same as that of "registered", that is, a client process has got the information needed to proceed with regular computation. We use a different name just to emphasize the fact that in DVS clients need to "register" views while in DC configurations become "established" under the control of the service.

to the processes, which requires that $|v.set \cup w.set| \geq |w.set|/2$ for all the views w, $w.id < v.id$, known by the process performing the check.

The DC specification requires a global intersection property which is the following: given two primary configurations, both of which are not dead, with no intervening totally established configurations, then there must exist a read and a write quorum of the configuration with a smaller identifier which are included in the membership set of the configuration with bigger identifier. The DC implementation checks *the same property* locally to each process. The intuitive reason why by checking locally the same property we can prove it also globally is that we exclude dead configurations. This suggests that also for DVS we can prove the stronger intersection property (the one checked locally) or we can use a weaker local check (the intersection required globally) if we do exclude dead views.

The DC specification is built upon a static configuration service, called CS. This service is basically the VS service adapted to handle configurations (see Section 4.2).

The automaton CS-TO-DC$_p$ is given in Figures 6-2 and 6-3. The overall system DC-IMPL is defined as the composition of CS and CS-TO-DC$_p$, for $p \in \mathcal{P}$.

Automaton CS-TO-DC$_p$ uses special messages, tagged either with "*info*", used to send information about the active and ambiguous configurations, or with "*got-state*", used to send the state submitted by a process to all the members of the configuration. The former information is needed to check the intersection property that new primary configurations have to satisfy according to the DC specification. The latter information is needed in order to compute the starting state for a new configuration. Thus, we use $\mathcal{M} = \mathcal{M}_c \cup \{(\text{"}info\text{"} \times \mathcal{V} \times 2^{\mathcal{V}})\} \cup \{\text{"}got\text{-}state\text{"}\}$, where $\mathcal{M}_c$ is the set of all client messages and $\mathcal{M}$ is the universe of all messages.

The major problem is that the DC specification requires a global intersection property (i.e., a property that can be checked only by someone that knows the entire system state), while each single process has a local knowledge of the system. So, in order to guarantee that a new configuration satisfies the requirement of DC, each single process needs information from other processes members of the configuration.

Informally, the filtering of configurations works as follows. Each process keeps track of the latest totally established configuration, called the "active" configuration, recorded into variable *act*, and a set of "ambiguous" configurations, recorded into variable *amb*, which are those configurations that were notified after the active configuration but did not become established yet. We define $use = act \cup amb$. When CS provides a new configuration to process p by means of action CS-NEWCONF$(c)_p$, process p sends out an "*info*" message containing its current act_p and amb_p values to all other processes in the new configuration, using the CS service, and waits to receive the corresponding "*info*" messages for configuration c from all the other members of c. After receiving this information (and updating its own act_p and amb_p accordingly), process p checks whether c has the required intersection property with each view in the use_p set. If so, configuration c is given in output to the client at p by means of action $\text{NEWCONF}(c)_p$.

When a new primary configuration c has been given in output to process p by means of action $\text{NEWCONF}(c)_p$, the client at p submits its current state together with a condenser function to be used to compute the starting state when all other members have submitted their state (such a condenser function depends on the application). Clearly the state of p is needed by other processes in the configuration while p needs the state of the other processes. Hence when a $\text{SUBMIT-STATE}(s, \phi)_p$ is executed at p, the state s submitted by process p is sent out with a *got-state* message to all other members of the configuration, using the CS service. Upon receiving the state of all other processes, CS-TO-DC_p uses the condenser function ϕ provided by the client at p in order to compute the starting state to be output, by means of action $\text{NEWSTATE}(s)_p$, to the client at p.

The communication mechanism of DC is quite different from that offered by CS: DC offers a broadcast/convergecast communication mechanism, while CS offers point-to-point communication channels. However it is not difficult to implement the former by using the communication service of the latter. The relevant code is in Figure 6-3. When a message m is submitted by means of action $\text{SUBMIT}(m, \phi, b, i)_p$, together with a condenser function ϕ, a quorum-type $b \in \{\text{“}read\text{”}, \text{“}write\text{”}\}$, and an operation identifier i, message m, tagged with i, is sent to all the members of the configuration using the CS service, and an operation descriptor for i is initialized. When another process q receives the message m of operation i, it delivers it to its client by means of action $\text{DELIVER}(m, i)_q$. At this point the client at q is expected to supply an acknowledgment value a for operation i by means of action $\text{ACKDLVR}(a, i)_q$. This value is sent back to p using the CS service. When p receives this value it updates the descriptor of operation i with the value obtained from q. If there are acknowledgment values from a quorum of the type specified by b, then the condenser function ϕ is applied to the acknowledgment values of this quorum in order to compute a response to the message m submitted by p. Such a response is given to p by means of action $\text{RESPOND}(m, i)_p$.

There are five derived variables for DVS-IMPL. Four of them are analogous to those of DVS, indicating the attempted, totally attempted, established, and totally established views, respectively. A fifth one, use_p, keeps track of the set of configurations used to check the intersection property before attempting a new configuration.

6.4.2 Invariants of DC-IMPL

This section contains invariants of DC-IMPL needed for the proof that DC-IMPL implements DC in Section 6.4.3. The proofs of these invariants are similar to those of the corresponding invariants of the DVS implementation (see Section 5.2.2). This is because the implementation of DC is similar to the implementation of DVS, thus many basic invariants are the same. For these basic invariant we provide an operational proof (i.e., a proof that does not rely exclusively on the state previous to the one for which the invariant states a property) for each of the invariants.

CS-TO-DC

Signature:

Input: $\textsc{cs-newconf}(c)_p$, $c \in \mathcal{C}$, $p \in c.set$
$\textsc{cs-gprcv}(m)_{q,p}$, $m \in \mathcal{M}$, $q \in \mathcal{P}$
$\textsc{cs-safe}(m)_{q,p}$, $m \in \mathcal{M}$, $q \in \mathcal{P}$
$\textsc{submit-state}(s, \phi)_p$, $s \in \mathcal{S}$, $\phi \in \Phi$
$\textsc{submit}(m, \phi, b, i)_p$, $m \in \mathcal{M}_c$, $\phi \in \Phi$, $b \in \{\text{"}read\text{"}, \text{"}write\text{"}\}$, $p \in \mathcal{P}$, $i \in OID_p$
$\textsc{ackdlvr}(a, i)_p$, $a \in A$, $i \in OID$, $p \in \mathcal{P}$

Internal: $\textsc{garbage-collect}(c)_p$, $c \in \mathcal{C}$

Output: $\textsc{cs-gpsnd}(m)_p$, $m \in \mathcal{M}$
$\textsc{dc-newconf}(c)_p$, $c \in \mathcal{V}$, $p \in c.set$
$\textsc{dc-newstate}(s)_p$, $s \in \mathcal{S}$
$\textsc{deliver}(m, i)_p$, $m \in \mathcal{M}_c$, $i \in OID$, $p \in \mathcal{P}$
$\textsc{respond}(a, i)_p$, $a \in A$, $i \in OID_p$, $p \in \mathcal{P}$

State:

$cur \in \mathcal{C}_\perp$, init c_0 if $p \in P_0$, $\perp$ else
$client\text{-}cur \in \mathcal{C}_\perp$, init c_0 if $p \in P_0$, $\perp$ else
$act \in \mathcal{C}$, init c_0
$amb \in 2^{\mathcal{C}}$, init $\emptyset$
$attempted \in 2^{\mathcal{C}}$, init $\{c_0\}$ if $p \in P_0$, $\emptyset$ else
for each $g \in \mathcal{G}$, $q \in \mathcal{P}$
  $info\text{-}rcvd[q, g] \in (\mathcal{C} \times 2^{\mathcal{C}})_\perp$, init $\perp$
  $rcvd\text{-}estb[q, g] \in (\mathcal{C} \times 2^{\mathcal{C}})_\perp$, init $\perp$

for each $g \in \mathcal{G}$
  $to\text{-}cs[g] \in seqof(\mathcal{M})$, init λ
  $info\text{-}sent[g] \in (\mathcal{C} \times 2^{\mathcal{C}})_\perp$, init $\perp$
  $dlv\text{-}queue[g] \in seqof(\mathcal{M})$, init λ
  $cond[g] \in \Phi_\perp$, init $\perp$
  $pend[g] \in \mathcal{O}$, init $\perp$
  $msg\text{-}dlvd[g] = OID \to \{\texttt{true}, \texttt{false}\}$
  $state\text{-}got[g] = \mathcal{P} \to \mathcal{S}_\perp$, init $\perp$
  $estb[g]$ a bool, init $\texttt{true}$ if $p \in P_0$ and $g = g_0$, $\texttt{false}$ else

Derived variables:

$\mathcal{A}tt \in 2^{\mathcal{C}}$, defined as $\mathcal{A}tt = \{c \in created \mid (\exists p \in c.set) c \in attempted_p\}$;
$\mathcal{E}st \in 2^{\mathcal{C}}$, defined as $\mathcal{E}st = \{c \in created \mid (\exists p \in c.set) estb[c.id]_p = \texttt{true}\}$;
$\mathcal{T}ot\mathcal{A}tt \in 2^{\mathcal{C}}$, defined as $\mathcal{T}ot\mathcal{A}tt = \{c \in created \mid (\forall p \in c.set) c \in attempted_p\}$;
$\mathcal{T}ot\mathcal{E}st \in 2^{\mathcal{C}}$, defined as $\mathcal{T}ot\mathcal{E}st = \{c \in created \mid (\forall p \in c.set) estb[c.id]_p = \texttt{true}\}$.
$use \in 2^{\mathcal{C}}$, defined as $use = \{act\} \cup amb$

Transitions:

input $\textsc{cs-newconf}(c)_p$
Eff: $cur := c$
  append $\langle \text{"}info\text{"}, act, amb\rangle$ to $to\text{-}cs[cur.id]$
  $info\text{-}sent[cur.id] := \langle act, amb\rangle$

input $\textsc{cs-gprcv}(\langle \text{"}info\text{"}, c, C\rangle)_{q,p}$
Eff: $info\text{-}rcvd[q, cur.id] := \langle c, C\rangle$
  if $c.id > act.id$ then $act := c$
  $amb := \{w \in amb \cup C \mid w.id > act.id\}$

input $\textsc{cs-safe}(\langle \text{"}info\text{"}, c, C\rangle)_{q,p}$
Eff: none

output $\textsc{newconf}(c)_p$
Pre: $c = cur$
  $c.id > client\text{-}cur.id$
  $\forall q \in c.set, q \neq p : info\text{-}rcvd[q, c.id] \neq \perp$
  $\forall w \in use : \exists R \in w.rqrms, R \in c.set$
  $\forall w \in use : \exists W \in w.wqrms, W \in c.set$
Eff: $amb := amb \cup \{c\}$
  $attempted := attempted \cup \{c\}$
  $client\text{-}cur := c$

input $\textsc{submit-state}(s, \phi)_p$
Eff: $g = client\text{-}cur.id$
  if $g \neq \perp$ then
    $state\text{-}got[g](p) := s$
    $cond[g] := \phi$
    append $\langle \text{"}state\text{-}got\text{"}, s\rangle$ to $to\text{-}cs[g]$

input $\textsc{cs-gprcv}(\langle \text{"}state\text{-}got\text{"}, s\rangle)_{q,p}$
Eff: $state\text{-}got[cur.id](q) := s$

input $\textsc{cs-safe}(\langle \text{"}state\text{-}got\text{"}, s\rangle)_{q,p}$
Eff: none

output $\textsc{newstate}(s)_p$
Pre: $g = cur.id$
  $g \neq \perp$
  $\forall q \in c.set$, $state\text{-}got[g](q) \neq \perp$
  $s = cond[g](state\text{-}got[g]|cur.set)$
  $estb[g] = \texttt{false}$
Eff: $estb[g] := \texttt{true}$
  append "*established*" to $to\text{-}cs[g]$

input $\textsc{cs-gprcv}(\text{"}established\text{"})_{q,p}$
Eff: $rcvd\text{-}estb[q, cur.id] := \texttt{true}$

input $\textsc{cs-safe}(\text{"}established\text{"})_{q,p}$
Eff: none

internal $\textsc{garbage-collect}(c)_p$
Pre: $\forall q \in c.set, rcvd\text{-}estb[q, c.id] = \texttt{true}$
  $c.id > act.id$
Eff: $act := cur$
  $amb := \{w \in amb \mid w.id > act.id\}$

output $\textsc{cs-gpsnd}(m)_p$
Pre: m is head of $to\text{-}cs[cur.id]$
Eff: remove head of $to\text{-}cs[cur.id]$

Figure 6-2: $\textsc{cs-to-dc}_p$

CS-TO-DC (transitions cont'd)

input SUBMIT$(m, \phi, b, i)_p$
Eff: $g = client\text{-}cur.id$
if $g \neq \bot$ then
$pend[g](i) := (m, \phi, b, \emptyset, \lambda(x) : x \rightarrow \bot, \texttt{false})$
append $\langle m, i \rangle$ to $to\text{-}cs[g]$

input CS-GPRCV$(\langle m, i \rangle)_{q,p}$
Eff: append $\langle m, i \rangle$ to $dlv\text{-}queue[cur.id]$

input CS-SAFE$(\langle m, i \rangle)_{q,p}$
Eff: none

output DELIVER$(m, i)_p$
Pre: $\langle m, i \rangle = \text{head}(dlv\text{-}queue[cur.id])$
$msg\text{-}dlvd[cur.id](i) = \texttt{false}$
Eff: $dlv\text{-}queue[cur.id] = \text{tail}(dlv\text{-}queue[cur.id])$
$msg\text{-}dlvd[cur.id](i) := \texttt{true}$

input ACKDLVR$(a, i)_p$
Eff: append $\langle a, i \rangle$ to $to\text{-}cs[client\text{-}cur.id]$

input CS-GPRCV$(\langle a, i \rangle)_{q,p}$
Eff: if $i \in OID_p$ then
$pend[i].ack(q) := a$

input CS-SAFE$(\langle a, i \rangle)_{q,p}$
Eff: none

output RESPOND$(r, i)_p$ *choose* Q
Pre: $g = cur.id$
$g \neq \bot$
$i \in OID_p$
$pend[g](i).rsp = \texttt{false}$
if $pend[g].sel =$ "*read*"
then $Q \in cur.rqrms$
else $Q \in cur.wqrms$
let $f = pend[g](i).ack$
$\forall q \in Q : f(q) \neq \bot$
$r = pend[g](i).cnd(f|Q)$
Eff: $pend[g](i).rsp := \texttt{true}$

Figure 6-3: CS-TO-DC$_p$ (transitions cont'd)

Invariant 6.4.1 (DC-IMPL)

In any reachable state if $\langle$"*info*"$, x, X\rangle \in to\text{-}cs[g]_p$ *or* $\langle$"*info*"$, x, X\rangle \in pending[p, g]$ *or* $\langle\langle$"*info*"$, x, X\rangle, p\rangle \in queue[g]$ *or* $info\text{-}rcvd[p, g]_q = \langle x, X\rangle$*, then* $info\text{-}sent[g]_p = \langle x, X\rangle$ *and* $cur.id_p \geq g$*.*

Proof Sketch: This invariant is true because whenever process p puts the message $\langle$"*info*"$, x, X\rangle$ into $to\text{-}cs[g]_p$ in action CS-NEWCONF(c), where $c.id = g$, it sets $info\text{-}sent[g]_p := \langle x, X\rangle$. Moreover at that moment it also sets $cur := c$. From that moment on, because configuration identifiers provided by CS only increase, we have that $cur.id_p \geq g$. Clearly this continues to be true when the "*info*" message goes through $pending[p, g]$, $queue[g]$ and finally gets to q and is recorded into $info\text{-}rcvd[p, g]_q$.
□

Invariant 6.4.2 (DC-IMPL)

In any reachable state, if $info\text{-}sent[g]_p = \langle x, X\rangle$*,* $w \in attempted_p$*, and* $w.id < g$*, then either* $w \in \{x\} \cup X$ *or* $w.id < x.id$*.*

Proof Sketch: If process p sent an "*info*" message for configuration g and has also attempted a previous configuration w, then either process p has already garbage-collected configuration w (if a configuration with identifier bigger than $w.id$ has been totally established) which implies that $w.id < x.id$ or w is still in the *use* set of p, which implies that $w \in \{x\} \cup X$. □

Invariant 6.4.3 (DC-IMPL)

In any reachable state:

1. $act_p \in \mathcal{T}ot\mathcal{E}st$.

2. If $\textit{info-sent}[g]_p = \langle x, X \rangle$ then $x \in \mathcal{T}ot\mathcal{E}st$.

3. $use_p \cap \mathcal{T}ot\mathcal{E}st \neq \emptyset$.

Proof Sketch: Variable act_p is initially a totally established configuration and is updated always to a totally established configuration (see action GARBAGE-COLLECT). Hence Part 1 follows. Part 2 follows from the fact that if $\textit{info-sent}[g]_p = \langle x, X \rangle$ then value of x is the value of the variable act_p at the time when $\textit{info-sent}[g]_p$ is written, and thus by Part 1 is totally established. Part 3 follows from Part 1 and the definition of use_p. ∎

Invariant 6.4.4 (DC-IMPL)
In any reachable state:

1. If $cur_p \neq \perp$ and $w \in use_p$, then $w.id \leq cur.id_p$.

2. If $cur_p \neq \perp$ and $\textit{client-cur}_p \neq cur_p$ and $w \in use_p$, then $w.id < cur.id_p$.

3. If $\textit{info-sent}[g]_p = \langle x, X \rangle$ and $w \in \{x\} \cup X$ then $w.id < g$.

Proof Sketch: The only action that adds a new configuration c to the use_p is action NEWCONF$(c)_p$. The precondition of this action requires that $cur_p = c$ which implies $cur.id = c.id$. The conclusion follows from the property of CS that configurations identifier are released in increasing order. This proves Part 1.
Part 2 follows by observing that when $cur_p \neq \perp$ and $\textit{client-cur}_p \neq cur_p$ a new configuration has been provided by CS but it has not been attempted yet. This implies that the current configuration cur_p cannot be in the use_p set. Combining this with the conclusion of Part 1, we have that for any $w \in use_p$, $w.id < cur.id_p$.
Finally Part 3 follows form the fact that process p sends information for a new configuration which has not been attempted yet. Once again, implies that the current configuration cur_p cannot be in the use_p set and we conclude as for Part 2. ∎

Invariant 6.4.5 (DC-IMPL)
In any reachable state, if $\textit{info-rcvd}[q, g]_p = \langle x, X \rangle$ *and* $w \in \{x\} \cup X$, *then either* $w \in use_p$ *or* $w.id < act.id_p$.

Proof Sketch: Since $\textit{info-rcvd}[q, g]_p = \langle x, X \rangle$ we have that process p has received $\langle x, X \rangle$ from process q. When receiving this information (action CS-GPRCV$(\langle \text{“}\textit{info}\text{”}, x, X \rangle)$) process p updates its use_p and $act.id_p$ sets. The conclusion follows from the code of CS-GPRCV$(\langle \text{“}\textit{info}\text{”}, x, X \rangle)$. ∎

Invariant 6.4.6 (DC-IMPL)

In any reachable state, if $c \in attempted_p$ and $q \in c.set$ then $cur.id_q \geq c.id$.

Proof Sketch: Since $c \in attempted_p$ we have that there has been a step $\text{NEWCONF}(c)_p$. By the precondition of this action, since $q \in c.set$ we have that process p received information from q for configuration c, that is, $info\text{-}sent[c.id]_q = \langle x, X \rangle$. By Invariant 6.4.1 we have that $cur.id_q \geq c.id$.
∎

The following invariant states a simple fact about DC-IMPL. This invariant does not have a corresponding one in DVS. However, since the statement of this invariant is simple, also for this invariant we provide an operational proof.

Invariant 6.4.7 (DC-IMPL)

In any reachable state, if $\langle m, i \rangle \in dlv\text{-}queue[g]_q$ then $pend[g](i).msg_p = m$, where p is such that $i \in OID_p$.

Proof Sketch: This is true because if $\langle m, i \rangle \in dlv\text{-}queue[g]_q$ we have that process q received the message $\langle m, i \rangle$ from a process p, with $i \in OID_p$. Moreover process p sent the message $\langle m, i \rangle$ in action $\text{SUBMIT}(m, \phi, b, i)_p$ and this action sets $pend[g](i).msg_p = m$. ∎

Next we provide the main invariants that are needed in the proof of the simulation relation.

Invariant 6.4.8 states that if a configuration c has been attempted by a process p and its membership contains a process q which has attempted a configuration w previous to c and there is no totally established configuration between w and c then c contains a read and a write quorum of w. Intuitively this is true because when p attempts c it must have received information from all the members of c, thus also from q; since q attempted w and w has not been garbage collected, because there are no totally established configurations in between w and c, process q includes w in the information it sends to p for configuration c. Invariant 6.4.9 generalizes Invariant 6.4.8 by claiming that the intersection property holds for any configurations w and c such that there is no totally established configuration x with $w.id < x.id < c.id$.

Finally Invariant 6.4.10 provides an intersection property crucial to the simulation relation. This last invariant is the one where dead configurations are needed.

Invariant 6.4.8 (DC-IMPL)

In any reachable state, suppose that $c \in attempted_p$, $q \in c.set$, $w \in attempted_q$, $w.id < c.id$, and there is no $x \in \mathcal{TotEst}$ such that $w.id < x.id < c.id$. Then there exist $R \in w.rqrms$ and $W \in w.wqrms$ such that $R \cup W \subseteq c.set$.

Proof: By induction on the length of an execution. The base case consists of proving that the invariant is true in the initial state. In the initial state, only c_0 is attempted, so the hypotheses cannot be satisfied. Thus, the statement is vacuously true.

For the inductive step assume the invariant is true in state s. We need to prove that it is true in s' for any possible step (s, π, s'). Fix c, w, p, and q, and assume that $c \in s'.attempted_p$, $q \in c.set$, $w \in s'.attempted_q$, $w.id < c.id$, and there is no $x \in s'.\mathcal{T}ot\mathcal{E}st$ such that $w.id < x.id < c.id$. Then also there is no $x \in s.\mathcal{T}ot\mathcal{E}st$ such that $w.id < x.id < c.id$. We consider four cases:

1. $c \in s.attempted_p$ and $w \in s.attempted_q$.

   Then by the inductive hypothesis we have that in state s there exist $R \in w.rqrms$ and $W \in w.wqrms$ such that $R \cup W \subseteq c.set$. Clearly this remains true in s' too (it remains true forever).

2. $c \notin s.attempted_p$ and $w \notin s.attempted_q$.

   This cannot happen because we cannot have both c and w becoming attempted in a single step.

3. $c \notin s.attempted_p$ and $w \in s.attempted_q$.

   Then π must be $\text{NEWCONF}(c)_p$. Since $q \in c.set$, by the precondition of π we have that $s.info\text{-}rcvd[q, c.id]_p = \langle x, X \rangle$ for some x and X. Then Invariant 6.4.1 implies that $s.info\text{-}sent[c.id]_q = \langle x, X \rangle$. Then, since $w.id < c.id$, Invariant 6.4.2 implies that either $w \in \{x\} \cup X$ or $w.id < x.id$. We claim that it must be $w \in \{x\} \cup X$. Indeed if $w.id < x.id$, by Invariant 6.4.3 we have that $x \in s.\mathcal{T}ot\mathcal{E}st$ and by Invariant 6.4.4, Part 3 (used with $w = x$) we have $x.id < c.id$; thus we would have a totally established configuration x such that $w.id < x.id < c.id$. This contradicts the inductive hypothesis. So it must be $w \in \{x\} \cup X$.

   By Invariant 6.4.5 we have that either $w \in s.use_p$ or $w.id < s.act.id_p$. In the former case, by the precondition of π, we have the needed conclusion. In the latter case, we obtain a contradiction. Indeed by Invariant 6.4.3 we have $s.act_p \in \mathcal{T}ot\mathcal{E}st$. Moreover by the precondition of π, $s.cur_p$ cannot be $\perp$ and $s.cur_p > s.client\text{-}cur_p$ and, by definition, $s.act_p \in s.use_p$. Hence by Invariant 6.4.4, Part 2, we have $s.act.id_p < s.cur.id_p = c.id$. Thus we would have a totally established configuration act such that $w.id < act.id < c.id$. This contradicts the inductive hypothesis.

4. $c \in s.attempted_p$ and $w \notin s.attempted_q$.

   Then π must be $\text{NEWCONF}(w)_q$. But this cannot happen. Indeed since $c \in s.attempted_p$ and $q \in c.set$, Invariant 6.4.6 implies that $s.cur.id_q \geq c.id$. Since $c.id > w.id$, we have $s.cur.id_q > w.id$. But the precondition of action π requires $s.cur.id_q = w.id$, so π is not enabled in s.

∎

Invariant 6.4.9 (DC-IMPL)

In any reachable state, suppose that $c \in \mathcal{A}tt$, and $w \in \mathcal{T}ot\mathcal{E}st$, $w.id < c.id$, and there is no $x \in \mathcal{T}ot\mathcal{E}st$

such that $w.id < x.id < c.id$. Then there exist $R \in w.rqrms$ and $W \in w.wqrms$ such that $R \cup W \subseteq c.set$.

Proof: By induction on the length of an execution. The base case consists of proving that the invariant is true in the initial state. In the initial state, only c_0 is attempted, so the hypotheses cannot be satisfied. Thus, the statement is vacuously true.

For the inductive step assume the invariant is true in state s. We need to prove that it is true in s' for any possible step (s, π, s'). Fix c and w, and assume that $c \in s'.\mathcal{A}tt$, $w \in s'.\mathcal{T}ot\mathcal{E}st$, $w.id < c.id$, and there is no $x \in s'.\mathcal{T}ot\mathcal{E}st$ such that $w.id < x.id < c.id$. We consider four cases:

1. $c \in s.\mathcal{A}tt$ and $w \in s.\mathcal{T}ot\mathcal{E}st$.

   The invariant follows from the inductive hypothesis.

2. $c \notin s.\mathcal{A}tt$ and $w \notin s.\mathcal{T}ot\mathcal{E}st$.

   This cannot happen because we cannot have both c becoming attempted and w becoming totally established in a single step.

3. $c \notin s.\mathcal{A}tt$ and $w \in s.\mathcal{T}ot\mathcal{E}st$.

   Then π must be $\text{NEWCONF}(c)_p$ for some p. The precondition of π implies that, for any configuration $y \in s.use_p$, there exist $R \in y.rqrms$ and $W \in y.wqrms$ such that $R \cup W \subseteq c.set$. Hence to prove the claim it is enough to prove that $w \in s.use_p$. We proceed by contradiction assuming that $w \notin s.use_p$.

   By Invariant 6.4.3, Part 3, $s.use_p \cap s.\mathcal{T}ot\mathcal{E}st \neq \emptyset$. Let m be the configuration in $s.use_p \cap s.\mathcal{T}ot\mathcal{E}st$ having the biggest identifier. We know that $m \neq w$ because $w \notin s.use_p$. Also, $m \neq c$, because $m \in s.\mathcal{T}ot\mathcal{E}st$ and $c \notin s.\mathcal{T}ot\mathcal{E}st$. It follows that $m.id \neq w.id$ and $m.id \neq c.id$.

   We claim that $m.id < w.id$. We have already shown that $m.id \neq w.id$. Suppose for the sake of contradiction that $m.id > w.id$. From the precondition of action π we have that $s.cur_p = c$ and hence $s.cur_p \neq \perp$. Also from the precondition of π we have that $s.client\text{-}cur_p < s.cur_p$. Since $m \in s.use_p$, Invariant 6.4.4, Part 2, implies that $m.id < s.cur.id_p$ and since $s.cur = c$ we have that $m.id < c.id$. So $w.id < m.id < c.id$. Since $m \in s'.\mathcal{T}ot\mathcal{E}st$, this contradicts the hypothesis of the inductive step. Therefore, $m.id < w.id$.

   Let n be the configuration in $s.\mathcal{T}ot\mathcal{E}st$ that has the smallest identifier strictly greater than that of m. Remember that $w \in s'.\mathcal{T}ot\mathcal{E}st$ and since $\pi = \text{NEWCONF}(c)_p$ we have that $w \in s.\mathcal{T}ot\mathcal{E}st$ and thus such an n exists and satisfies $m.id < n.id \leq w.id$. Since $m \in s.use_p$, the precondition of π implies that there exist $R \in m.rqrms$ and $W \in m.wqrms$ such that $R \cup W \subset c.set$. Since by inductive hypothesis the invariant is true in state s, we have that there exist $R' \in m.rqrms$ and $W' \in m.wqrms$ such that $R' \cup W' \subset n.set$. By the properties of quorums we have that there exists one process $q \in (R \cup W) \cap (R' \cup W')$ and thus we have that $q \in n.set \cap c.set$. By the precondition of π, $s.info\text{-}rcvd[q, c.id]_p = \langle x, X \rangle$ for some x, X. Then Invariant 6.4.1 implies that $s.info\text{-}sent[c.id]_q = \langle x, X \rangle$ and Invariant 6.4.3 says that $x \in s.\mathcal{T}ot\mathcal{E}st$. Then Invariant 6.4.4, Part 3 (used with $w = x$), implies that $x.id < c.id$. Since $n \in s.\mathcal{T}ot\mathcal{E}st$, we have that $n \in s.attempted_q$. Then Invariant 6.4.2 (used with $w = n$) implies that either $n \in \{x\} \cup X$ or $n.id < x.id$. In either case, $\{x\} \cup X$ contains a configuration $y \in s.\mathcal{T}ot\mathcal{E}st$ (either n or x) such that $n.id \leq y.id < c.id$. Then Invariant 6.4.5 implies that either $y \in s.use_p$ or $y.id < s.act.id_p$. By Invariant 6.4.3, Part 1, $s.act_p \in s.\mathcal{T}ot\mathcal{E}st$ and by definition, $s.act_p \in s.use_p$. So in either case, the hypothesis that m is the totally established configuration with the largest identifier belonging to $s.use_p$ is contradicted.

4. $c \in s.\mathcal{A}tt$ and $w \notin s.\mathcal{T}ot\mathcal{E}st$.

   Then π must be NEWSTATE$(\cdot)_p$ for some p. Let m be the configuration in $s.\mathcal{T}ot\mathcal{E}st$ with the largest identifier that is strictly less than $w.id$. By the statement for s, we know that there exist $R' \in m.rqrms$ and $W' \in m.wqrms$ such that $R' \cup W' \in w.set$ and there exist $R'' \in m.rqrms$ and $W'' \in m.wqrms$ such that $R'' \cup W'' \in c.set$. Hence, by the properties of quorums, there is a process $q \in w.set \cap c.set$.

   Since $c \in s.\mathcal{A}tt$, there exists a process r such that $c \in s.attempted_r$. Thus also $c \in s'.attempted_r$. Since $w \in s'.\mathcal{T}ot\mathcal{E}st$, we have that $w \in s'.attempted_q$. By assumption, there is no configuration $x \in s'.\mathcal{T}ot\mathcal{E}st$ such that $w.id < x.id < c.id$. By Invariant 6.4.8 applied to state s' (with $p = r$), we have that there exist $R \in w.rqrms$ and $W \in m.wqrms$ such that $R \cup W \in c.set$, as needed.

$\square$

So far, the proof that DC-IMPL implements DC has been very similar to the proof that DVS-IMPL implements DVS. The following invariant is different from the corresponding one in the DVS implementation, Invariant 5.2.17. Here is where dead configurations are needed.

Invariant 6.4.10 (DC-IMPL)
In any reachable state, if $c, w \in \mathcal{A}tt$, $w.id < c.id$, configuration w is not dead and there is no $x \in \mathcal{T}ot\mathcal{E}st$ with $w.id < x.id < c.id$, then there exist $R \in w.rqrms$ and $W \in w.wqrms$ such that $R \cup W \subseteq c.set$.

Proof: By induction on the length of an execution. The base case consists of proving that the invariant is true in the initial state. In the initial state, only c_0 is attempted, so the hypotheses cannot be satisfied. Thus, the statement is vacuously true.

For the inductive step assume the invariant is true in state s. We need to prove that it is true in s' for any possible step (s, π, s'). Fix c and w, and assume that $c \in s'.\mathcal{A}tt$, $w \in s'.\mathcal{A}tt$, $w.id < c.id$, $w \notin s'.dead$, and there is no $x \in s'.\mathcal{T}ot\mathcal{E}st$ such that $w.id < x.id < c.id$. We consider four cases:

1. $w \in s.\mathcal{A}tt$ and $c \in s.\mathcal{A}tt$. Then the invariant is true by the inductive hypothesis.

2. $w \notin s.\mathcal{A}tt$ and $c \notin s.\mathcal{A}tt$. This is not possible because a single action cannot make both w and c attempted.

3. $w \notin s.\mathcal{A}tt$ and $c \in s.\mathcal{A}tt$. Then it must be that $\pi=\text{NEWCONF}(w)_{p'}$ for some process p' that attempts w. Since $c \in s.\mathcal{A}tt$ we have that $c \in s'.\mathcal{A}tt$. Hence there exists p such that $c \in s'.attempted_p$.

   Now we claim that there must exist a process $q \in c.set \cap w.set$.

   Clearly we have that $w \notin \mathcal{T}ot\mathcal{E}st$. Let $Y = \{y | y \in \mathcal{T}ot\mathcal{E}st, y.id < w.id\}$. We first show that Y is nonempty: The initial configuration is totally established, thus $c_0 \in \mathcal{T}ot\mathcal{E}st$; moreover $c_0.id \leq w.id$. If $c_0.id = w.id$, then we have $w = c_0$. But then $w \in \mathcal{T}ot\mathcal{E}st$, a contradiction to the definition of this case. So we must have $c_0.id < w.id$, which implies that $c_0 \in Y$, so Y is nonempty.

   Now fix z to be the configuration in Y with the largest id. We have that there is no $x \in \mathcal{T}ot\mathcal{E}st$ with $z.id < x.id < c.id$. Then Invariant 6.4.9 implies that there exist $R \in z.rqrms$ and $W \in z.wqrms$ such that $R \cup W \in w.set$ and also that there exist $R' \in z.rqrms$ and $W' \in z.wqrms$ such that $R \cup W \in c.set$. By the properties of quorums we have that $(R \cup W) \cap (R' \cup W') \neq \{\}$. Hence we have that there exists q such that $q \in c.set \cap w.set$.

   Now we claim that $w \in s'.attempted_q$. By contradiction assume that $w \notin s'.attempted_q$. Since $c \in s'.attempted_p$ and $q \in c.set$ we have that $s'.info\text{-}rcvd[q, c.id]_p = \langle x, X \rangle$ for some x and X. By Invariant 6.4.1 we have that $s'.cur.id_q \geq c.id > w.id$. Since we assumed that $w \notin s'.attempted_q$, by definition of dead configuration we have that w is dead. But this contradicts the hypothesis of the invariant, which states that w is not dead. Hence it must be that $w \in s'.attempted_q$.

   Hence we have that $c \in s'.attempted_p$, $q \in c.set$, $w \in s'.attempted_q$, $w.id < c.id$ and there are no $x \in s'.\mathcal{T}ot\mathcal{E}st$ such that $w.id < x.id < c.id$. By Invariant 6.4.8 we have that there exist $R \in w.rqrms$ and $W \in w.wqrms$ such that $R \cup W \subseteq c.set$.

4. $w \in s.\mathcal{A}tt$ and $c \notin s.\mathcal{A}tt$. Then it must be that $\pi=\text{NEWCONF}(c)_p$ for some process p that attempts c. We have that $s'.attempted_p$.

   The rest of the proof is as in the previous case: it claims that there exists $q \in c.set \cap w.set$, that $w \in s'.attempted_q$ and then uses Invariant 6.4.8 to get the needed conclusion.

$\square$

6.4.3 Proof that DC-IMPL implements DC

We are now ready to prove that DC-IMPL implements DC. We first provide a function mapping states of DC-IMPL to states of DC. Then we will prove that this function is an abstraction function.

The abstraction function is given in Figure 6-4.

Let s be a state of TO-IMPL. The state $t = \mathcal{F}_{dc}(s)$ of TO is the following.

1. $t.created = \cup_{p\in\mathcal{P}} s.attempted_p$
2. for each $p \in \mathcal{P}$, $t.cur\text{-}cid[p] = s.client\text{-}cur.id_p$
3. for each $g \in \mathcal{G}$, $t.attempted[g] = \{p|g = c.id, c \in s.attempted_p\}$
4. for each $g \in \mathcal{G}$, $t.state\text{-}dlv[g] = \{p|s.estb[g]_p\}$
5. for each $g \in \mathcal{G}$, $t.got\text{-}state[g](p) = s.state\text{-}got[g](p)_p$
6. for each $g \in \mathcal{G}$, $t.condenser[g](p) = s.cond[g]_p$
7. for each $g \in \mathcal{G}$ and $i \in OID_p$, $t.pending[g](i) = \perp$ if $s.pend[g](i)_p = \perp$, otherwise is defined by:
   - $msg = s.pend[g](i).msg_p$
   - $cnd = s.pend[g](i).cnd_p$
   - $sel = s.pend[g](i).sel_p$
   - $dlv = \{q|s.msg\text{-}dlvd[g](i) = \mathtt{true}\}$
   - $ack = \{q|\langle a, i\rangle \in s.to\text{-}cs_q \texttt{ or } \langle a, i\rangle \in s.\text{CS}.pending[q, g] \texttt{ or } s.pend[g](i).ack(q)_p = a\}$
   - $rsp = s.pend[g](i).rsp_p$

Figure 6-4: The abstraction function $\mathcal{F}_{dc}$.

In order to prove that $\mathcal{F}_{dc}$ is an abstraction function we need to prove that for any initial state s of DC-IMPL we have that $\mathcal{F}_{dc}(s)$ is an initial state of DC and that for any possible step π of DC-IMPL there exists a sequence of α of steps of DC such that the trace of α, that is the externally observable behavior, is equal to the trace of π. Lemmas 6.4.11 and 6.4.12, prove the above. The proof of these lemmas is similar to the corresponding ones of DVS. The key difference is in the supporting invariant used in the proof, Invariant 6.4.10, which is crucial in proving the simulation relation for the case when the implementation executes action $\pi = \text{NEWCONF}(c)_p$, for some configuration c and some process p.

Lemma 6.4.11 *If s is an initial state of* DC-IMPL *then $\mathcal{F}_{dc}(s)$ is an initial state of* DC.

Proof: Let s be the initial state of DC-IMPL. We have that $s.attempted_p$ is $\{c_0\}$ for $p_\in P_0$ and $\emptyset$ for $p \notin P_0$. Hence, by definition of $\mathcal{F}_{dc}$ we have that $t.created = \{c_0\}$ which is as in the initial state of DC.

We have that $s.client\text{-}cur.id_p$ is $\{g_0\}$ for $p_\in P_0$ and $\perp$ for $p \notin P_0$. Hence, by definition of $\mathcal{F}_{dc}$ we have that $t.cur\text{-}cid$ is $\{g_0\}$ for $p_\in P_0$ and $\perp$ for $p \notin P_0$. This is as in the initial state of DC.

We have that $s.estb[g]_p$ is `true` for $p \in P_0, g = g_0$ and `false` otherwise. Hence we have that $t.state\text{-}dlv[g]$ is P_0 if $g = g_0$ and $\emptyset$ if $g \neq g_0$. This is as in the initial state of DC.

We have that $s.cond[g]_p = \bot$ for all g and p. Hence $t.condenser[g](p) = \bot$ for all g and p, which is as in the initial state of DC.

We have that $pend[g](i) = \bot$ for all g and i. Hence $t.pending[g](i) = \bot$ everywhere, which is as in the initial state of DC.

Hence if s is an initial state of DC-IMPL, we have that $\mathcal{F}_{dc}(s)$ is an initial state of DC. ∎

Lemma 6.4.12 *Let s be a reachable state of* DC-IMPL*, $\mathcal{F}_{dc}(s)$ a reachable state of* DC*, and (s, π, s') a step of* DC-IMPL*. Then there is an execution fragment α of* DC *that goes from $\mathcal{F}_{dc}(s)$ to $\mathcal{F}_{dc}(s')$, such that $trace(\alpha) = trace(\pi)$.*

Proof: By case analysis based on the type of the action π. (The only interesting case is where $\pi =$ NEWCONF$(v)_p$.) Define $t = \mathcal{F}_{dc}(s)$ and $t' = \mathcal{F}_{dc}(s')$.

1. $\pi =$ CS-CREATECONF(c)

   Then $trace((s, \pi, s')) = \lambda$. Action π modifies *created*. The definition of $\mathcal{F}_{dc}$ is not sensitive to this change. Therefore, $t = t'$, and we set $\alpha = t$.

2. $\pi =$ CS-NEWCONF$(c)_p$

   Then $trace((s, \pi, s')) = \lambda$. Action π modifies *current-confid*$[p]$, cur_p and *info-sent*$[cur.id]_p$, and adds an "*info*" message to *to-cs*$[cur.id]_p$. The definition of $\mathcal{F}_{dc}$ is not sensitive to any of these changes. Therefore, $t = t'$, and we set $\alpha = t$.

3. $\pi =$ CS-GPSND$(m)_p$

   Then $trace((s, \pi, s')) = \lambda$. Action π just moves a message from the queue *to-cs*$[cur.id]_p$ to the queue CS.*pending*$[p, current\text{-}confid[p]]$. The definition of $\mathcal{F}_{dc}$ is not sensitive to this change. Therefore, $t = t'$, and we set $\alpha = t$.

4. $\pi =$ CS-ORDER(m, p, g)

   Then $trace((s, \pi, s')) = \lambda$. Action π moves a message from CS.*pending*$[p, g]$ to CS.*queue*$[g]$. The definition of $\mathcal{F}_{dc}$ is not sensitive to this change. Therefore, $t = t'$, and we set $\alpha = t$.

5. $\pi =$ CS-GPRCV$(\langle \text{"}info\text{"}, c, C\rangle)_{q,p}$

   Then $trace((s, \pi, s')) = \lambda$. Action π increments the *next* pointer in CS, and may modify *act* and *amb* in CS-TO-DC. The definition of $\mathcal{F}_{dc}$ is not sensitive to this change. Therefore, $t = t'$, and we set $\alpha = t$.

6. $\pi =$ CS-SAFE$(\langle \text{"}info\text{"}, c, C\rangle)_{q,p}$

Then $trace((s, \pi, s')) = \lambda$. Action π increments the *next-safe* pointer in CS. The definition of $\mathcal{F}_{dc}$ is not sensitive to this change. Therefore, $t = t'$, and we set $\alpha = t$.

7. $\pi = \text{NEWCONF}(c)_p$

   Then $trace((s, \pi, s)) = \pi$. In DC-IMPL, this action modifies only variables amb_p, $attempted_p$, $client\text{-}cur_p$. We have $s'.client\text{-}cur_p = c$ and $s'.attempted_p = s.attempted_p \cup \{c\}$. By definition of $\mathcal{F}_{dc}$, we have that $t'.cur\text{-}cid[p] = s'.client\text{-}cur.id_p$, $t'.created = \cup_{p \in \mathcal{P}} s'.attempted_p$ and $t'.attempted[c.id] = \{p | c \in s'.attempted_p\}$. Hence we have that $t'.cur\text{-}cid[p] = c.id$, $t'.created = t.created \cup \{c\}$ and $t'.attempted[c.id] = t.attempted[c.id] \cup \{p\}$, while all other state variables in t' are as in t.

   We consider two cases:

   (a) $c \in t.created$.

   In this case, we set $\alpha = (t, \pi', t')$, where $\pi' = \text{NEWCONF}(c)_p$. The code shows that π' brings DC from state t to state t'. It remains to prove that π' is enabled in state t, that is, that $c \in t.created$ and $c.id > t.cur\text{-}cid[p]$. The first of these two conditions is true because of the defining condition for this case. The second condition follows from the precondition of π in DC-IMPL: this precondition implies that $c.id > s.client\text{-}cur.id_p$, and by the definition of $\mathcal{F}_{dc}$ we have $t.cur\text{-}cid[p] = s.client\text{-}cur.id_p$.

   (b) $c \notin t.created$. In this case we set $\alpha = (t, \pi', t'', \pi'', t')$, where $\pi' = \text{CREATECONF}(c)_p$, $\pi'' = \text{NEWCONF}(c)_p$, and t'' is the unique state that arises by running the effect of π' from t. The code shows that α brings DC from state t to state t'. It remains to prove that π' is enabled in t and that π'' is enabled in t''.

   - Action π'. We start by proving that π' is enabled in t.

     The precondition of π' requires that (i) $\forall w \in t.created$, $c.id \neq w.id$ and (ii) if c is not dead, the following two conditions, $C1$ and $C2$, are true.

     $C1$: $\forall w \in t.created$, $w.id < c.id$, either w is dead or $\exists x \in t.\mathcal{T}ot\mathcal{E}st$ satisfying $w.id < x.id < c.id$ or there exist $R \in w.rqrms$ and $W \in w.wqrms$ such that $R \cup W \subseteq c.set$;

     $C2$: $\forall w \in t.created$, $c.id < w.id$, either w is dead or $\exists x \in t.\mathcal{T}ot\mathcal{E}st$ satisfying $c.id < x.id < w.id$ or there exist $R \in c.rqrms$ and $W \in c.wqrms$ such that $R \cup W \subseteq w.set$.

     – requirement (i). To see requirement (i), suppose for the sake of contradiction that there exists $w \in t.created$ such that $w.id = c.id$. The precondition of π in DC-IMPL implies that $c = s.cur_p$, which implies that $c \in s.\text{CS}.created$. Since $w \in t.created$, the definition of $\mathcal{F}_{dc}$ implies that $w \in s.attempted_q$ for some q. This implies that $w \in s.\text{CS}.created$. Hence both c and w are created, that is, belong to $t.created$ and since $w.id = c.id$ we have that $c = w$. But this is impossible since $c \notin t.created$ and $w \in t.created$.

– requirement (ii). If c is dead, then requirement (ii) is trivially satisfied. Hence assume that c is not dead. We have to show that both $C1$ and $C2$ are true.

Let us start with $C1$. Assume that there exists $w \in t.created$ such that $w.id < c.id$, that w is not dead and that there is no $x \in t.\mathcal{TotEst}$ satisfying $w.id < x.id < c.id$ (otherwise $C1$ is true and we are done). Since $w \in t.created$, by definition of $\mathcal{F}_{dc}$, $w \in s.attempted_q$ for some q. Clearly, $w \in s'.attempted_q$. Therefore, $w \in s'.\mathcal{Att}$. By the code of π we have that $c \in s'.attempted_p$. Therefore we also have $c \in s'.\mathcal{Att}$. Moreover, there is no $x \in s'.\mathcal{TotEst}$ satisfying $w.id < x.id < c.id$ (this is true in t and thus, by definition of $\mathcal{F}_{dc}$, is true in s and, because π' does not establish any configuration, it stays true in s'). By Invariant 6.4.10 we have that there exist $R \in w.rqrms$ and $W \in w.wqrms$ such that $R \cup W \subseteq c.set$, as needed to prove $C1$.

We look now at $C2$. Assume that there exists $w \in t.created$ such that $c.id < w.id$, and that there is no $x \in t.\mathcal{TotEst}$ satisfying $c.id < x.id < w.id$. We already know that c is not dead. Since $w \in t.created$, by definition of $\mathcal{F}_{dc}$, $w \in s.attempted_q$ for some q. Clearly, $w \in s'.attempted_q$. Therefore, $w \in s'.\mathcal{Att}$. By the code of π we have that $c \in s'.attempted_p$. Therefore we also have $c \in s'.\mathcal{Att}$. Moreover, there is no $x \in s'.\mathcal{TotEst}$ satisfying $c.id < x.id < w.id$. By Invariant 6.4.10 we have that exist $R \in c.rqrms$ and $W \in c.wqrms$ such that $R \cup W \subseteq w.set$, as needed to prove $C2$.

This proves that π' is enabled in t.

- Action π''. Now we prove that π'' is enabled in state t''.

  The precondition of π'' requires that $c \in t''.created$ and $c.id > t''.cur\text{-}cid[p]$. The first condition is true because c is added to *created* by π'. The second condition follows from the precondition of π in DC-IMPL: The precondition of π implies that $c.id > s.client\text{-}cur.id_p$. The definition of $\mathcal{F}_{dc}$ implies that $t.cur\text{-}cid[p] = s.client\text{-}cur.id_p$. Moreover, $t''.cur\text{-}cid[p] = t.cur\text{-}cid[p]$. It follows that $c.id > t''.cur\text{-}cid[p]$. Thus π'' is enabled in state t''.

8. $\pi = \text{SUBMIT-STATE}(o, \phi)_p$

   Then $trace((s, \pi, s)) = \pi$. This action sets $state\text{-}got[g](p)_p := o$, $cond[g]_p := \phi$ and appends $\langle \text{“}state\text{-}got\text{”}, o\rangle$ to $to\text{-}cs[g]_p$, where $g = client\text{-}cur_p$. By the definition of $\mathcal{F}_{dc}$ we have that $t'.got\text{-}state[g](p) = o$ and $t'.condenser[g](p) = \phi$, while all other state variables are as in t. We set $\alpha = (t, \pi, t')$. The code shows that π actually brings DC from t to t'. Moreover π is an input action, so it is always enabled.

9. $\pi = \text{CS-GPRCV}(\langle \text{“}state\text{-}got\text{”}, o\rangle)_{q,p}$

Then $trace((s,\pi,s')) = \lambda$. Action π sets $state\text{-}got[g](q)_p := o$. The definition of $\mathcal{F}_{dc}$ is not sensitive to this change. Therefore, $t = t'$, and we set $\alpha = t$.

10. $\pi =$ CS-SAFE$(\langle\text{“}state\text{-}got\text{”}, o\rangle)_{q,p}$

    Then $trace((s,\pi,s')) = \lambda$. Action π increments the *next-safe* pointer in CS. The definition of $\mathcal{F}_{dc}$ is not sensitive to this change. Therefore, $t = t'$, and we set $\alpha = t$.

11. $\pi =$ NEWSTATE$(o)_p$

    Then $trace((s,\pi,s)) = \pi$. This action sets $estb[g] :=$ `true` and appends the message "*established*" to $to\text{-}cs[g]$, where $g = cur.id_p$. By definition of $\mathcal{F}_{dc}$ we have that $t'.state\text{-}dlv[g] = t.state\text{-}dlv[g] \cup \{p\}$ and this is the only difference between t' and t. We set $\alpha = (t,\pi,t')$. The code shows that π actually brings DC from t to t'. It remains to prove that π is enabled in t. The precondition of π in DC-IMPL are the same as those in DC. Thus π is enabled in DC because it is enabled in DC-IMPL.

12. $\pi =$ CS-GPRCV("*established*")$_p$

    Then $trace((s,\pi,s')) = \lambda$. Action π sets $rcvd\text{-}estb[q, cur.id_p]_p :=$ `true`. The definition of $\mathcal{F}_{dc}$ is not sensitive to this change. Therefore, $t = t'$, and we set $\alpha = t$.

13. $\pi =$ CS-SAFE("*established*")$_p$

    Then $trace((s,\pi,s')) = \lambda$. Action π increments the *next-safe* pointer in CS. The definition of $\mathcal{F}_{dc}$ is not sensitive to this change. Therefore, $t = t'$, and we set $\alpha = t$.

14. $\pi =$ GARBAGE-COLLECT$(c)_p$

    Then $trace((s,\pi,s')) = \lambda$. This action can modify act_p and amb_p. The definition of $\mathcal{F}_{dc}$ is not sensitive to these changes. Therefore, $t = t'$, and we set $\alpha = t$.

15. $\pi =$ SUBMIT$(m,\phi,b,i)_p$

    Then $trace((s,\pi,s)) = \pi$. This action sets $pend[g] := \langle m,\phi,b,\emptyset,\perp,\texttt{false}\rangle$ and appends $\langle m,i\rangle$ to $to\text{-}cs[g]$, where $g = client\text{-}cur.id_p \neq \perp$. The definition of $\mathcal{F}_{dc}$ shows that this changes the queue of pending operations *pending* so that $t'.pending[g](i) = \langle m,\phi,b,\emptyset,\perp,\texttt{false}\rangle$ and this is the only difference between t and t'. We set $\alpha = (t,\pi,t')$. The code shows that π actually brings DC from t to t'. Moreover π is an input action, so it is always enabled.

16. $\pi =$ CS-GPRCV$(m,i)_p$

    Then $trace((s,\pi,s')) = \lambda$. Action π appends $\langle m,i\rangle$ to $dlv\text{-}queue[cur.id_p]_p$. The definition of $\mathcal{F}_{dc}$ is not sensitive to this change. Therefore, $t = t'$, and we set $\alpha = t$.

17. $\pi =$ CS-SAFE$(m,i)_p$

Then $trace((s, \pi, s')) = \lambda$. Action π increments the *next-safe* pointer in CS. The definition of $\mathcal{F}_{dc}$ is not sensitive to this change. Therefore, $t = t'$, and we set $\alpha = t$.

18. $\pi = \text{DELIVER}(m, i)_p$

    Then $trace((s, \pi, s)) = \pi$. This action deletes the head of $dlv\text{-}queue[g]_p$ and sets $msg\text{-}dlvd[g](i) :=$ `true`, where $g = cur.id_p$. By the definition of $\mathcal{F}_{dc}$ we have that the only thing that changes from t to t' is $pending[g](i).dlv$, that is $t'.pending[g](i).dlv = t.pending[g](i).dlv \cup \{p\}$. We set $\alpha = (t, \pi, t')$. The code shows that π actually brings DC from t to t'. It remains to prove that π is enabled in t. From the precondition of π we have that $s.msg\text{-}dlvd[g](i)_p =$ `false` and thus we have that $p \notin t.pending[g](i).dlv$. From the precondition of π we have that $\langle m, i \rangle$ is the head of $dlv\text{-}queue[g]_p$. By Invariant 6.4.7 we have that $t.pending[g](i).msg_p = m$. Hence π is enabled in state t of DC.

19. $\pi = \text{ACK-DLVR}(a, i)_p$

    Then $trace((s, \pi, s)) = \pi$. This action appends $\langle a, i \rangle$ to $to\text{-}cs[g]_p$ where $g = client\text{-}cur.id_p$. By the definition of $\mathcal{F}_{dc}$ we have that the only thing that changes from t to t' is $pending[g](i).ack$, that is $t'.pending[g](i).ack = t.pending[g](i).ack \cup \{p\}$. We set $\alpha = (t, \pi, t')$. The code shows that π actually brings DC from t to t'. Moreover π is enabled in t because it is an input action.

20. $\pi = \text{CS-GPRCV}(a, i)_p$

    Then $trace((s, \pi, s')) = \lambda$. Action π sets $\langle m, i \rangle$ to $dlv\text{-}queue[cur.id_p]_p$. The definition of $\mathcal{F}_{dc}$ is not sensitive to this change. Therefore, $t = t'$, and we set $\alpha = t$.

21. $\pi = \text{CS-SAFE}(a, i)_p$

    Then $trace((s, \pi, s')) = \lambda$. Action π increments the *next-safe* pointer in CS. The definition of $\mathcal{F}_{dc}$ is not sensitive to this change. Therefore, $t = t'$, and we set $\alpha = t$.

22. $\pi = \text{RESPOND}(r, i)_p$

    Then $trace((s, \pi, s)) = \pi$. This action sets $pend[g](i).rsp :=$ `true`. By the definition of $\mathcal{F}_{dc}$ we have that $t'.pending[g](i).rsp =$ `true`. We set $\alpha = (t, \pi, t')$. The code shows that π actually brings DC from t to t'. It remains to prove that π is enabled in t. The precondition of π in DC-IMPL is as the precondition of π in DC. Hence π is enabled in DC because it is enabled in DC-IMPL.

∎

Lemmas 6.4.11 and 6.4.12 prove that $\mathcal{F}_{dc}$ is an abstraction function from DC-IMPL to DC and thus the following theorem holds.

Theorem 6.4.13 *Every trace of* DC-IMPL *is a trace of* DC*.*

6.5 An application of DC

In this section we show how to use DC to implement an atomic multi-writer multi-reader shared register. The algorithm is an extension of the single-writer multi-reader atomic register of Attiya, Bar-Noy and Dolev [12]. A similar extension was provided in [66].

6.5.1 Overview

In this section we provide a description of the algorithm and the code. We start with the description of the algorithm.

Each process keeps a copy of the shared register, in variable *val* paired with a tag, in variable *tag*. Tags are used to establish the time when values are written: a value paired with a bigger tag has been written after a value paired with a smaller tag. Tags consists of pairs $\langle j, p \rangle$ where j is a sequence number (a non negative integer) and p is a process identifier. Tags are ordered according to their sequence numbers with processes identifiers breaking ties. Given a tag $\langle j, p \rangle$ the notation $t.seq$ denotes the sequence number j.

The algorithm has two modes of operation: a *normal mode* and a *reconfiguration mode*. The latter is used to establish a new configuration. It is entered when a new configuration is announced (action NEWCONF) and is left when the configuration becomes established (action NEWSTATE). The former is the mode where read and write operations are performed and it is entered when a configuration is established and is left when a new configuration is announced. During the reconfiguration mode pending operations are delayed until the normal mode is restored. Variable *conf-status* is used to keep track of the mode (values *exch-ready*, *exch-wait* are for the reconfiguration mode).

Clients of the service can request read and write operations by means of actions READ_p and $\text{WRITE}(x)_p$. We assume that each client does not invoke a new operation request before receiving the response for the previous request. Both type of requests (read and write) are handled in a similar way: there is a *query* phase and a subsequent *propagate* phase. During the query phase the server receiving the request "queries" a read-quorum in order to get the value of the shared register and the corresponding tag for each of the members of the read-quorum. From these it selects the value x corresponding to the max tag t. This concludes the query phase. In the propagation phase the server sends a new value and a new tag (which are (x, t) for the case of a READ_p operation and $(y, \langle t.seq + 1, p \rangle)$ for a $\text{WRITE}(y)_p$ operation) to the members of a write quorum. These processes update their own copy of the register if the tag received is greater than their current tag; then they send back an acknowledgment to the server p. When p gets the acknowledgment message from the members of a write quorum, the propagate phase is completed. At this point the server can respond to the client that issued the operation with either the value read, in the case of a read operation, or with just a confirmation, in the case of a write operation.

We remark that when a configuration change happens during the execution of a requested operation, the completion of the operation is delayed until the normal mode is restored. However if the query phase has already been completed it is not necessary to repeat it in the new configuration.

We denote by $\mathcal{T} = \{\langle j, p\rangle | j \in \mathbf{N}, p \in \mathcal{P}\}$ the set of tags. This set is ordered according to the first component, with the process identifiers breaking ties. We denote by $\mathcal{X}$ the set of values that the shared register can assume. We assume that there is a default value $x_0 \in \mathcal{X}$. Initially all members of c_0 have the shared copy of the register set to x_0.

All other data types used in the code are as defined for DC.

Figures 6-5 and 6-6 provide the code of ABD-CODE$_p$; in this code code, we use the following condenser functions:

- ϕ_{maxtag} which computes the value and the tag of the max tag register copy. Formally this function takes a collection of tuples $Z = \{(\text{"}query\text{-}ack\text{"}, v, t, h)\}$, with $t \in \mathcal{T}$ and returns one such quadruple which has a maximum tag t among the elements of Z together with the set Q of processes that submitted the tuples; formally it returns a tuple $(\text{"}query\text{-}ack\text{"}, v, t, h, Q)$.

- ϕ_{ack} which returns an acknowledgment. Formally this function takes a collection of pairs $Z = \{(\text{"}prop\text{"}, t)\}$, all of which have the same value $t \in \mathcal{T}$, and returns $(\text{"}prop\text{-}ack\text{"}, t, Q)$, where Q is the set of processes that submitted the pairs.

- ϕ_{state} which computes the up to date state for a new configuration. Formally it takes a collection of triples $Z = \{(x, t, h)\}$ where x is a value, t a tag and h a configuration identifier, considers the subset Z' of the triples of z with maximum h and returns the first two components of a triple of Z' which has the maximum t in Z'. Such a triple can be picked by default, choosing e.g., the one that came from the process with the smallest identifier.

ABD-CODE (signature and state)

Signature:

Input:
- READ$_p$, $p \in \mathcal{P}$
- WRITE$(x)_p$, $x \in \mathcal{X}$, $p \in \mathcal{P}$
- DELIVER$(m, i)_p$, $m \in \mathcal{M}$, $i \in OID$, $p \in \mathcal{P}$
- RESPOND$(a, i)_p$, $a \in \mathcal{A}$, $i \in OID_p$, $p \in \mathcal{P}$
- NEWCONF$(c)_p$, $c \in \mathcal{C}$, $p \in c.set$
- NEWSTATE$(s)_p$, $s \in \mathcal{S}$,

Output:
- SUBMIT$(m, \phi, b, i)_p$, $m \in \mathcal{M}$, $\phi \in \Phi$, $b \in \{\text{"}r\text{"}, \text{"}w\text{"}\}$, $p \in \mathcal{P}$, $i \in OID_p$
- ACKDLVR$(a, i)_p$, $a \in \mathcal{A}$, $i \in OID$, $p \in \mathcal{P}$
- READ-CONFIRM$(x)_p$, $x \in X$, $p \in \mathcal{P}$
- WRITE-CONFIRM$_p$, $p \in \mathcal{P}$
- SUBMIT-STATE$(s, \phi)_p$, $s \in \mathcal{S}$, $\phi \in \Phi$,

State:

$current \in \mathcal{C}_\perp$ init c_o if $p \in P_0$ else $\perp$
$high \in \mathcal{G}_\perp$, init g_0 if $p \in P_0$ else $\perp$
$val \in \mathcal{X}$, initially x_0
$tag \in \mathcal{T}$, initially $\langle 0, p\rangle$
$prop\text{-}val \in \mathcal{X}$, initially x_0
$prop\text{-}tag \in \mathcal{T}$, initially $\langle 0, p\rangle$

$conf\text{-}status \in \{normal, exch\text{-}ready, exch\text{-}wait\}$, init $normal$
$status \in \{query\text{-}ready, query\text{-}wait, prop\text{-}ready, prop\text{-}wait, prop\text{-}done\}$, init $query\text{-}ready$
$request \in \{\text{"}read\text{"}, (\text{"}write\text{"}, x), \perp\}$, init $\perp$
$ack\text{-}q$, $seqof(A \times OID)$, init λ

Figure 6-5: The ABD interface and state.

We define the system ABD-SYS as the composition of DC and ABD-CODE$_p$ for each $p \in \mathcal{P}$.

ABD-CODE (transitions)

Actions:

input READ$_p$
Eff: $request$:= "$read$"

input WRITE$(x)_p$
Eff: $request := (\text{"}write\text{"}, x)$

output SUBMIT$(\text{"}query\text{"}, \phi_{maxtag}, \text{"}r\text{"}, i)_p$
Pre: $current \neq \bot$
$i \notin used\text{-}ids$
$conf\text{-}status = normal$
$status = query\text{-}ready$
$request \neq \bot$
Eff: $used\text{-}ids := used\text{-}ids \cup \{i\}$
$status := query\text{-}wait$

input RESPOND$((\text{"}query\text{-}ack\text{"}, x, t, h, Q), i)_p$
Eff: if $request$ = "$read$" then
$prop\text{-}tag := t$
$prop\text{-}val := x$
if $request = (\text{"}write\text{"}, y)$ then
$prop\text{-}tag := (t.seq + 1, p)$
$prop\text{-}val := y$
$status := prop\text{-}ready$

output SUBMIT$((\text{"}prop\text{"}, x, t), \phi_{ack}, \text{"}w\text{"}, i)_p$
Pre: $current \neq \bot$
$i \notin used\text{-}ids$
$conf\text{-}status = normal$
$status = prop\text{-}ready$
$x = prop\text{-}val$
$t = prop\text{-}tag$
Eff: $status := prop\text{-}wait$

input RESPOND$((\text{"}prop\text{-}ack\text{"}, t, Q), i)_p$
Eff: $status := prop\text{-}done$

output READ-CONFIRM$(x)_p$
Pre: $conf\text{-}status = normal$
$status = prop\text{-}done$
$request$ = "$read$"
$x = prop\text{-}val$
Eff: $request := \bot$
$status := query\text{-}ready$

output WRITE-CONFIRM$_p$ choose x
Pre: $conf\text{-}status = normal$
$status = prop\text{-}done$
$request = (\text{"}write\text{"}, x)$
Eff: $request := \bot$
$status := query\text{-}ready$

input DELIVER$(\text{"}query\text{"}, i)_p$
Eff: append $((\text{"}query\text{-}ack\text{"}, val, tag, high), i)$ to $ack\text{-}q$

input DELIVER$((\text{"}prop\text{"}, x, t), i)_p$
Eff: if $t > tag$ then
$val := x$
$tag := t$
append $((\text{"}prop\text{-}ack\text{"}, t), i)$ to $ack\text{-}q$

output ACKDLVR$(a, i)_p$
Pre: $head(ack\text{-}q) = (a, i)$
Eff: $ack\text{-}q := tail(ack\text{-}q)$

input NEWCONF$(c)_p$
Eff: $current := c$
$conf\text{-}status := exch\text{-}ready$
if $status = query\text{-}wait$ then
$status := query\text{-}ready$
if $status = prop\text{-}wait$ then
$status := prop\text{-}ready$
$ack\text{-}q := \lambda$

output SUBMIT-STATE$((x, t, h), \phi_{state})_p$
Pre: $conf\text{-}status = exch\text{-}ready$
$x = val$
$t = tag$
$h = high$
Eff: $conf\text{-}status := exch\text{-}wait$

input NEWSTATE$(x, t)_p$
Eff: $conf\text{-}status := normal$
if $t > tag$ then
$val := x$
$tag := t$
$high := current$

Figure 6-6: The ABD-CODE transitions.

6.5.2 Proof that ABD-SYS is an atomic register

In this section we prove that ABD-SYS implements an atomic read/write shared register. This proof uses an approach similar to that used in Chapter 5 and in [41] to prove the correctness of applications built on top of DVS and VS, respectively.

We need the following history variables.

For a process p and a configuration identifier g variable $buildtag[p, g] \in \mathcal{T}_\perp$ is defined as follows:

- If $current.id_p = g$ then $buildtag[p, g] = tag_p$;
- If $current.id_p > g$ then $buildtag[p, g]$ is the value of tag_p at the moment when process p leaves configuration g;
- If $current.id_p < g$ then $buildtag[p, g] = \perp$.

Informally this $buildtag[p, g]$ records the value of the latest tag, if any, used in configuration g by process p.

The value of $buildtag[p, g]$ can be easily computed by following the statement that modifies tag_p in actions DELIVER and NEWSTATE, with another statement $buildtag[p, current.id_p] := tag_p$. It should be clear that (i) when p is in configuration g we have that $buildtag[p, g] = tag_p$ and (ii) after p leaves configuration g forever afterwards, $buildtag[p, g]$ contains the value of tag_p at the point when p left configuration g.

We also need history variables to record the beginning and the end of query and propagate phases of each operation, the configurations where the phases are executed, the quorum of precesses involved, as well as the tags returned by the query phases and the ones written by propagate phases. We define the following history variables:

- $query\text{-}begin[i] \in \{\texttt{true}, \texttt{false}\}$, initially `false` for all $i \in OID$. This variable is set to `true` when action SUBMIT(“*query*”, ϕ_{maxtag}, “r”, $i)_p$ is executed by some process p. Informally, when $query\text{-}begin[i] = \texttt{true}$ the query phase of operation i has started.
- $query\text{-}end[i] \in \{\texttt{true}, \texttt{false}\}$, initially `false` for all $i \in OID$. This variable is set to `true` when action RESPOND((“*query-ack*”, $x, t, h, Q), i)_p$ is executed by some process p and for any value of x, t, h and Q. Informally, when $query\text{-}end[i] = \texttt{true}$ the query phase of operation i has been completed.
- $query\text{-}quorum[i] \in 2^{\mathcal{P}}_\perp$, initially $\perp$ for all $i \in OID$. This variable is set to Q when action RESPOND((“*query-ack*”, $x, t, h, Q), i)_p$ is executed by some process p and for any value of x, t and h. Informally, $query\text{-}quorum[i]$ records the read quorum used by the query phase of operation i.
- $query\text{-}conf[i] \in \mathcal{C}_\perp$, initially $\perp$ for all $i \in OID$. This variable is set to c when action RESPOND((“*query-ack*”, $x, t, h, Q), i)_p$ is executed by some process p when $current_p = c$, and for

any value of x, t, h and Q. Informally, variable *query-conf*$[i]$ records the configuration in which the query phase of operation i is performed.

- *query-tag*$[i] \in \mathcal{T}_\perp$, initially $\perp$ for all $i \in OID$. This variable is set to t when action $\text{RESPOND}((\text{"}query\text{-}ack\text{"}, x, h, t, Q), i)_p$ is executed by some process p and for any value of x, h and Q. Informally, variable *query-tag*$[i]$ records the tag returned by the query phase of operation i.

- *prop-begin*$[i] \in \{\texttt{true}, \texttt{false}\}$, initially `false` for all $i \in OID$. This variable is set to `true` when action $\text{SUBMIT}((\text{"}prop\text{"}, x, t), \phi_{ack}, \text{"}w\text{"}, i)_p$ is executed by some process p and for any value of x and t. Informally, when *prop-begin*$[i] = \texttt{true}$ the propagation phase of operation i has started.

- *prop-end*$[i] \in \{\texttt{true}, \texttt{false}\}$, initially `false` for all $i \in OID$. This variable is set to `true` when action $\text{RESPOND}((\text{"}prop\text{-}ack\text{"}, t, Q), i)_p$ is executed by some process p and for any value of t and Q. Informally, when *prop-end*$[i] = \texttt{true}$ the propagation phase of operation i has been completed.

- *prop-quorum*$[i] \in 2^{\mathcal{P}}{}_\perp$, initially $\perp$ for all $i \in OID$. This variable is set to Q when action $\text{RESPOND}((\text{"}prop\text{-}ack\text{"}, t, Q), i)_p$ is executed by some process p and for any value of t. Informally, *prop-quorum*$[i]$ records the write quorum used by the propagate phase of operation i.

- *prop-conf*$[i] \in \mathcal{C}_\perp$, initially $\perp$ for all $i \in OID$. This variable is set to c when action $\text{RESPOND}((\text{"}prop\text{-}ack\text{"}, t, Q), i)_p$ is executed by some process p when $current_p = c$ and for any value of t and Q. Informally, variable *prop-conf*$[i]$ records the configuration in which the propagate phase of operation i is performed.

- *prop-tag*$[i] \in \mathcal{T}_\perp$, initially $\perp$ for all $i \in OID$. This variable is set to t when action $\text{RESPOND}((\text{"}prop\text{-}ack\text{"}, t, Q), i)_p$ is executed by some process p for any value of Q. Informally, variable *prop-tag*$[i]$ records the tag written during the propagate phase of operation i; this is also the tag associated with operation i.

We define the set of *summaries* as the set $\mathcal{S}um = \{\langle x, t, g\rangle | x \in \mathcal{X}, t \in \mathcal{T}, g \in \mathcal{G}\}$. Given a summary $y \in \mathcal{S}um$, $y = \langle x, t, g\rangle$ selectors for the components are $y.value = x$, $y.tag = t$ and $y.high = g$.

Informally a summary $y = \langle x, t, g\rangle$ is used to record that some process p has been into a state in which $val_p = x$, $tag_p = t$ and $high_p = g$.

We write *allstate*$[p, g]$ to denote a set of summaries defined so that $\langle x, t, h\rangle \in$ *allstate*$[p, g]$ if one of the following holds.

1. $current_p = g$ and $val_p = x$,$tag_p = t$ and $high_p = h$.

2. *got-state*$[g](p) = (x, t, h)$

3. Message ("*query-ack*", x, t, h) is somewhere in message mechanism of DC, more formally:

   - $\langle(\text{"}query\text{-}ack\text{"}, x, t, h), i\rangle \in ack\text{-}q_p$, for some operation identifier i;
   - $pending[g](i).ack(p) = (\text{"}query\text{-}ack\text{"}, x, t, h)$, for some operation identifier i.

4. $pending[g](i).msg_p = (\text{"}prop\text{"}, x, t, h)$, for some operation identifier i.

We write $allstate[g]$ to denote $\bigcup_{p \in P} allstate[p, g]$, and $allstate$ to denote $\bigcup_{g \in G} allstate[g]$.

If Y is a partial function from process identifiers to summaries, then we define: $maxprimary(Y) = \max_{q \in dom(Y)}\{Y(q).high\}$, $reps(Y) = \{q \in dom(Y) : Y(q).high = maxprimary\}$ and $chosenrep(Y)$ denotes some element q' in $reps(Y)$ that maximizes $Y(q').tag$.

Next we provide some preliminary invariants. Since these invariants state simple facts and also some of them are very similar to the ones used in [41], we provide operational proofs instead of formal assertional proofs.

Invariant 6.5.1 (ABD-SYS)
In any reachable state, we have that $current.id_p = cur\text{-}cid_p$.

Proof: Both variables are initially g_0 for $p \in P_0$ and $\perp$ for $p \notin P_0$. Both variables are set to $c.id$ when action NEWCONF$(c)_p$ is executed. ∎

Invariant 6.5.2 (ABD-SYS)
In any reachable state, if $query\text{-}end[i] = \texttt{true}$ or $prop\text{-}end[i] = \texttt{true}$, then we have that $prop\text{-}conf[i] \in created \setminus dead$.

Proof: By definition of *prop-conf*, *query-end* and *prop-end* we have that if $query\text{-}end[i] = \texttt{true}$ or $prop\text{-}end[i] = \texttt{true}$, then $prop\text{-}conf[i] \neq \perp$. Let $c = prop\text{-}conf[i]$. Clearly c must be created.
The query phase and the propagate phase are executed in normal processing, that is when $conf\text{-}status = normal$ for each process involved. Thus processes participating in the query or in the propagate phase have executed action *newstate* for configuration c. Such an action is executed only when all members of c have submitted their state to the DC service. In order to submit their state for configuration c, each member must have executed action NEWCONF(c). Hence c is not dead. ∎

Invariant 6.5.3 (ABD-SYS)
In any reachable state the following are true.

1. $high_p \in \mathcal{T}ot\mathcal{A}tt$
2. *If* $y \in allstate$ *then* $y.high \in \mathcal{T}ot\mathcal{A}tt$.

Proof Sketch: For any summary y configuration $y.high$ is a configuration which has been the $high_p$ for some process p. Hence it suffices to prove Part 1. Variable $high_p$ is set only by action $\text{NEWSTATE}(\cdot)_p$. This action is executed only if all the members of the configuration have submitted their state to the DC service. This implies that all the members of the configuration have attempted $high_p$. ∎

Invariant 6.5.4 (ABD-SYS)
In any reachable state:

1. $high_p \notin dead$
2. If $y \in allstate$ then $y.high \notin dead$.

Proof Sketch: This invariant follows easily from the previous one since a totally attempted configuration cannot be dead. ∎

Invariant 6.5.5 (ABD-SYS)
In any reachable state, the following is true: If $c \in created \setminus dead$ *and* $\exists y \in allstate$ *such that* $y.high > c.id$ *then there exists* $R \in c.rqrms$ *such that for all* $p \in R$, $current.id_p > c.id$.

Proof: Let configuration c and $y \in allstate$ be such that $c \in created \setminus dead$ and $y.high > c.id$. By Invariant 6.5.3 we have that $y.high \in \mathcal{TotAtt}$. Then By Invariant 6.3.3, we have that there exists $R \in c.rqrms$ such that for all $p \in R$, $cur\text{-}cid_p > c.id$. By Invariant 6.5.1 we have that $current_p = cur\text{-}cid_p$. It follows that there exists $R \in c.rqrms$ such that for all $p \in R$, $current_p > c.id$.
∎

Invariant 6.5.6 (ABD-SYS)
In any reachable state, the following is true. Let c, w *be two configurations such that* $c.id < w.id$ *and* $t \in \mathcal{T}$. *Let* $p \in c.set \cap w.set$ *and let* $y = got\text{-}state[w.id](p)$. *If* $buildtag[p, c.id] \geq t$, $y \neq \perp$ *and* $y.high \geq c.id$, *then* $y.tag \geq t$.

Proof Sketch: Since $y = got\text{-}state[w.id](p) \neq \perp$ we have that $current.id_p \geq w.id > c.id$. Since $y.high \geq c.id$ we have that $y.tag \geq buildtag[p, c.id]$. By assumption $buildtag[p, c.id] \geq t$. Hence we have that $y.tag \geq t$. ∎

The following invariant is the analog of Invariant 6.13 of [41].

Invariant 6.5.7 (ABD-SYS)
In any reachable state, for any p, *for any summary* y *and for all* $c, w \in created$ *we have that: If* $state\text{-}dlv[p, c.id] \neq \perp$, $c.id < w.id$ *and* $y \in allstate[p, w.id]$ *then* $y.high \geq c.id$.

Proof Sketch: Assume that p, y, c and w satisfy the hypothesis. Since $\mathit{state\text{-}dlv}[p, c.id] \neq \bot$, we have that process p has executed action $\text{NEWSTATE}(\cdot)_p$ for configuration c. When executing this action it sets $\mathit{high}_p := c$. Hence any summary due to p for a later configuration has the *high* component which is at least $c.id$. This is true also for y which is a summary due to p for configuration w, since $w.id > c.id$. ∎

Invariant 6.5.8 (ABD-SYS)
In any reachable state, if $\mathit{got\text{-}state}[g](p) \neq \bot$ *then* $\mathit{current}.id_p \geq g$.

Proof Sketch: Since $\mathit{got\text{-}state}[g](p) \neq \bot$ we have that p submitted its state to DC. In order for process p to submit its state for configuration g it must be that $\mathit{current}_p = c$, where $c.id = g$. Afterwards, by monotonicity of configuration identifiers, we have that $\mathit{current}.id_p \geq g$. ∎

Next we provide a sequence of invariants which leads to the proof that ABD-SYS implements an atomic read/write shared register.

Invariant 6.5.9 (ABD-SYS)
In any reachable state, the following is true. Let $c \in \mathit{created} \setminus \mathit{dead}$, $W \in c.\mathit{wqrms}$ *and let* $t \in \mathcal{T}$. *If for every* $r \in W$ *such that* $\mathit{current}.id_r > c.id$ *it holds that* $\mathit{estb}[c.id]_r = \texttt{true}$ *and* $\mathit{buildtag}[r, c.id] \geq t$, *then we have that every summary* $y \in \mathit{allstate}$ *with* $y.\mathit{high} > c.id$ *satisfies* $y.\mathit{tag} \geq t$.

Proof: By induction on the length of the execution. In the initial state, the only created configuration is c_0, and there are no summaries y with $y.\mathit{high} > g_0$. So the invariant is vacuously true and the base case is proved.

For the inductive step assume that the invariant is true in a state s. We need to prove that the invariant is true in s' for any step (s, π, s'). To prove that the invariant is true in s' we fix $c \in s'.\mathit{created} \setminus s'.\mathit{dead}$, $W \in c.\mathit{wqrms}$, and $t \in \mathcal{T}$, and assume that for every $r \in W$, if $s'.\mathit{current}.id_r > c.id$ then $s'.\mathit{estb}[c.id]_r$ and $s'.\mathit{build\text{-}tag}[r, c.id] \geq t$. To prove the invariant we need to prove the following conclusion: for any summary $y \in s'.\mathit{allstate}$ such that $y.\mathit{high} > c.id$, we have that $y.\mathit{tag} \geq t$.

Let us first consider the case when $c \notin s.\mathit{created}$. Since $c \in s'.\mathit{created}$, action π must be $\text{CREATECONF}(c)$. We consider two subcases.

1. $\exists r \in c.\mathit{set} : s.\mathit{current}.id_r > c.id$.

   Fix such a process r. Since c has just been created, r has not attempted c in s so $c \in s.\mathit{dead}$, which implies $c \in s'.\mathit{dead}$. This contradicts the assumption that $c \in s'.\mathit{created} \setminus s'.\mathit{dead}$. So this case is not possible.

2. $\nexists r \in c.\mathit{set} : s.\mathit{current}.id_r > c.id$.

In this case, we claim that the invariant is true because there is no y in $s'.allstate$ with $y.high > c.id$. By contradiction, fix a $y \in s'.allstate$ such that $y.high > c.id$. By Invariant 6.5.4 configuration $y.high$ is not dead. Then Invariant 6.5.5 applied to s' implies that there exists $R \in c.rqrms$ such that for all $q \in R$, $s'.current.id_q > c.id$. Fix some $q \in R$. Since $s.current.id_q = s'.current.id_q$, it follows that $s.current.id_q > c.id$. But $q \in c.set$ and thus we have a contradiction of the defining condition for this case.

Hence in the case when $c \in s.created$ the invariant is true. For the rest of the proof we assume that $c \in s.created$. Since $c \notin s'.dead$ we have that $c \notin s.dead$.

As usual, the interesting steps are those that convert the hypothesis from false to true, and those that keep the hypothesis true while converting the conclusion from true to false. There are no steps that convert the hypothesis from false to true. So it remains to consider any steps that keep the hypothesis true while converting the conclusion from true to false. Thus, we assume that, for every $r \in W$, if $s.current.id_r > c.id$ then $s.estb[c.id]_r = \texttt{true}$ and $s.buildtag[r, c.id] \geq t$. The only steps that can convert the conclusion from true to false are steps that produce a new summary (because if a summary $y \in s.allstate$ has $y.high > c.id$, then by the inductive hypothesis we have that $y.tag \geq t$ and we are done.)

Any step that produces a summary y by modifying an old summary $y' \in s.allstate$, in such a way that $y'.tag \leq y.tag$ and $y'.high = y.high$, is easy to handle: For such a step, $y'.high > c.id$ and so the inductive hypothesis implies that $t \leq y'.tag \leq y.tag$, as needed. So the only concern is with a NEWSTATE action for some configuration w.

Hence we assume that $\pi =$NEWSTATE$(\langle \hat{x}, \hat{t}, \hat{h} \rangle)_p$ for some process p such that $s.current_p = w$. Action π produces the following new summary $y = \langle s'.val_p, s'.tag_p, s'.high_p \rangle$ and since, by the code of π, $s'.high = w.id$ we have $y.high = w.id$. Assume that $y.high > c.id$ (otherwise we are done). In order to prove the invariant we have to prove that $y.tag \geq t$.

Since $y.high > c.id$ and $y.high = w.id$, we have that $w.id > c.id$. We also notice that configuration w is not dead in s. Indeed by the code of π we have that $s'.high_p = s'.current_p$ and since $s'.current_p = s.current_p = w$ we have that $w = s'.high_p$. By Invariant 6.5.4 we have that $w \notin s'.dead$. Clearly $w \notin s.dead$.

Let $Y = s.got\text{-}state[w.id]$ and let $p' = chosenrep(Y)$. Let y' be the summary $y' = s.got\text{-}state[w.id](p')$. Before proving that $y.tag \geq t$ we prove two claims that are needed for the proof.

- CLAIM 1. $y'.high \geq c.id$.

  Let c' denote the configuration which has the highest identifier in the set of configurations $\{c'' | c'' \in s'.\mathcal{T}ot\mathcal{E}st, c''.id < w.id\}$.

  Remember that $w \notin dead$ and that $c.id < w.id$.

  We consider two possible cases:

1. $c'.id \geq c.id$

   Since $c' \in s'.\mathcal{T}ot\mathcal{E}st$, we have that $c' \notin s'.dead$ (and thus $c' \notin s.dead$). Also $w \notin s'.dead$. By definition of c', we have that there are no totally established configurations in between c' and w. Then Invariant 6.3.1 shows that there exists $R \in c'.rqrms$ such that $R \subseteq w.set$. Fix any $q \in R$. Since $c' \in s'.\mathcal{T}ot\mathcal{E}st$ we have that $s.state\text{-}dlv[q, c'.id] \neq \perp$. Let $y'' = s.got\text{-}state[w.id](q)$. By Invariant 6.5.7, we obtain that $y''.high \geq c'.id$. By the definition of p' as a member that maximizes the *high* component in the summary recorded in *got-state*, we have $y'.high \geq y''.high$. Therefore $y'.high \geq c'.id \geq c.id$, completing our proof of the claim for this case.

2. $c'.id < c.id$

   By assumption, $c \notin dead$. We have observed above that $w \notin dead$. By definition of c', we have that there are no totally established configurations in between c' and w and since $c'.id < c.id < w.id$ it follows that there are no totally established configurations in between c and w. Then Invariant 6.3.1 shows that there exists $R \in c.rqrms$ such that $R \subseteq w.set$. We have that $R \cap W \neq \emptyset$. Let q be any element of $R \cap W$. Since $R \subseteq w.set$, we have that $q \in w.set$. Because $s.got\text{-}state[w.id](q) \neq \perp$, Invariant 6.5.8 implies that $s.current.id_q \geq w.id$. Since $w.id > c.id$ we have that $s.current.id_q > c.id$.

   Recall that we have assumed that for every $r \in W$, if $s.current.id_r > c.id$ then $s.estb[c.id]_r = \texttt{true}$ and $s.buildtag[r, c.id] \geq t$. Therefore, since $q \in W$ and $s.current.id_q > c.id$, we have that $s.estb[c.id]_q = \texttt{true}$ and thus $s.state\text{-}dlv[q, c.id] \neq \perp$.

   Let $y'' = s.got\text{-}state[w.id](q)$; thus $y'' \in s.allstate[q, w.id]$. By Invariant 6.5.7, we obtain that $y''.high \geq c.id$. By the definition of p' as a member that maximizes the *high* component in the summaries recorded in $got\text{-}state[w.id]$, we have $y'.high \geq y''.high$. Therefore $y'.high \geq c.id$, completing our proof of the claim for this case.

Thus we have proved that $y'.high \geq c.id$.

- Claim 2. If $y'.high = c.id$, then in s' there is no totally established configuration w' such that $c.id < w'.id < w.id$.

  To see this, consider again the totally established configuration c' with the largest identifier less than $w.id$. By the definition of c' it suffices to prove that $c'.id \leq c.id$.

  Neither c' nor w are dead. Since there are no totally established configuration in between c' and w, Invariant 6.3.1 implies that $w.set$ contains a read-quorum of c', and thus an element of $c'.set$. That is, there exists $q \in c'.set \cap w.set$. Consider the summary $y'' = s.got\text{-}state[w.id](q)$. By the precondition of π we have $y'' \neq \perp$ and thus we have that $y'' \in s.allstate[q, w.id]$. By definition of c' as the totally established configuration with the largest identifier less than $w.id$, we have that $c'.id < w.id$ and that $state\text{-}dlv[q, c'.id] \neq \perp$. Then Invariant 6.5.7 shows that

$y''.high \geq c'.id$. The definition of p' as a member that maximizes the *high* component among the summaries recorded in $s.got\text{-}state[w.id]$, shows that $y'.high \geq y''.high \geq c'.id$. But the claim is conditional to the hypothesis that $y'.high = c.id$. So if $y'.high = c.id$ we have that $c.id \geq c'.id$, which gives the claim.

Hence we have proved that if $y'.high = c.id$ in s' there is no totally established configuration w' such that $c.id < w'.id < w.id$.

We are now ready to prove that $y.tag \geq t$. By Claim 1, we have that $y'.high \geq c.id$.

If $y'.high > c.id$, by the inductive hypothesis we have that $y'.tag \geq t$ and since $y.tag \geq y'.tag$, we have that $y.tag \geq t$, as needed.

So suppose $y'.high = c.id$. By Claim 2, in s' there is no totally established configuration w' such that $c.id < w'.id < w.id$.

We know that $c \notin s.dead$ and that $w \notin s.dead$. Thus by Invariant 6.3.1 we have that there exists $R \in c.rqrms$ such that $R \subseteq w.set$; therefore, since $R \cap W \neq \emptyset$, there exists $q \in W \cap w.set$. By the precondition of π, using the fact that $q \in w.set$, we have $s.got\text{-}state[w.id](q) \neq \bot$ and thus $s.current_q \geq w.id > c.id$. Thus also $s'.current_q > c.id$. We have that $q \in W$ and $s'.current_q > c.id$; for such a process we have that $s'.buildtag[q, c.id] \geq t$ and $s'.estb[c.id]_q = \texttt{true}$. Since action π does not modify these variables we have that $s.buildtag[q, c.id] \geq t$ and $s.estb[c.id]_q = \texttt{true}$. The precondition of π shows that $s.got\text{-}state[w.id](q) \neq \bot$. Let summary $y'' = s.got\text{-}state[w.id](q)$. Thus $y'' \in allstate[q, w.id]$. Since $s.estb[c.id]_q = \texttt{true}$ we have that $state\text{-}dlv[q, c.id] \neq \bot$. By Invariant 6.5.7, we have that $y''.high \geq c.id$. By Invariant 6.5.6 we have that $y''.tag \geq t$. Recall that $y'.high = c.id$ and by definition y' is a summary with maximal *high*. Since $y''.high \geq c.id$ it must be that $y''.high = c.id$, and so the summary y'' from q is among those with maximal high in $s.got\text{-}state[w.id]$. By the definition of p' as a member that maximizes the *tag* component, we have that $y'.tag \geq y''.tag$, so $y'.tag \geq t$. Since by the code $y.tag = y'.tag$, we have that $y.tag \geq t$, as needed. ∎

The next invariant states that when a completed propagate phase performed in a configuration c has propagate a tag t, all summaries whose *high* component is greater than $c.id$ have a *tag* component which is greater than or equal to t.

Invariant 6.5.10 (ABD-SYS)
In any reachable state, if $prop\text{-}end[i] = \texttt{true}$, $prop\text{-}tag[i] = t$, $c = prop\text{-}conf[i]$ and $y \in allstate$ is a summary with $y.high > c.id$, then $y.tag \geq t$.

Proof: Since $prop\text{-}end[i] = \texttt{true}$, by Invariant 6.5.2 we have that $c \in created \setminus dead$.

Let $W = prop\text{-}quorum[i]$ (this is the write quorum used in the propagate phase of operation i). Since $prop\text{-}conf[i] = c$, for all $p \in W$, we have that $estb[c.id]_p = \texttt{true}$ (because process p is involved in the propagate phase of operation i, so it must have established c). This implies also that $current.id_p \geq c.id$. Moreover since $prop\text{-}tag[i] = t$, if a processor $p \in W$ has $current.id_p > c.id$, by monotonicity of the tag, we have that $buildtag[p, c.id] \geq t$ (because process p is involved in the propagate phase of operation i and hence knows tag t; when it leaves configuration c, $buildtag[p, c.id]$ must be at least t).

By Invariant 6.5.9 we have that $y.tag \geq t$. ∎

The next lemma states a property of any execution of ABD-SYS. Namely, if a completed propagate phase propagates a tag t, any subsequent query phase that totally follows the propagate phase (that is, begins after the propagate phase has ended), gets a tag which is greater than or equal to t.

Lemma 6.5.11 (ABD-SYS)
In any execution, if the completed propagate phase of an operation i totally precedes the completed query phase of an operation j, then $query\text{-}tag[j] \geq prop\text{-}tag[i]$ in any state where both are not $\perp$.

Proof: Let s be any state where $query\text{-}tag[j]$ and $prop\text{-}tag[i]$ are not $\perp$, that is both the propagate phase of operation i and the query phase of operation j have been completed. Then we have $s.prop\text{-}end[i] = \texttt{true}$ and $s.query\text{-}end[j] = \texttt{true}$. Clearly we also have $s.query\text{-}begin[j] = \texttt{true}$. Let (s', π, s_1) be the step when $prop\text{-}end[i]$ is set to $\texttt{true}$, let (s'', π, s_2) be the step when $query\text{-}begin[j]$ is set to $\texttt{true}$ and let (s''', π, s_3) be the step when $query\text{-}end[j]$ is set to $\texttt{true}$. We must have that s_1 precedes s'', s_2 precedes s''' and s_3 precedes s in the execution.

Let $t = s.prop\text{-}tag[i]$. We need to prove that $s.query\text{-}tag[j] \geq t$.

Let $W = s.prop\text{-}quorum[i]$ (this is the write quorum used by the propagate phase of operation i) and let $R = s.query\text{-}quorum[j]$ (this is the read quorum used by the query phase of operation j).

Let $c_1 = s.prop\text{-}conf[i]$ and $c_2 = s.query\text{-}conf[j]$. We have $W \in c_1.wqrms$ and $R \in c_2.rqrms$.

Since $s.prop\text{-}end[i] = \texttt{true}$ and $s.query\text{-}end[j] = \texttt{true}$, by Invariant 6.5.2 we have that $c_1, c_2 \in created \setminus dead$ in state s.

We consider three cases:

1. $c_1.id = c_2.id$

   Since $c_1 = c_2$ we have that $R \cap W \neq \emptyset$. Let $q \in R \cap W$. Process q submits to the condenser function ϕ_{maxtag} of operation j a tag which is greater than or equal to t. By definition of ϕ_{maxtag} we have that $s.query\text{-}tag[j] \geq t$, which gives the claim.

2. $c_1.id < c_2.id$

   Let p be any process of R. By the code we have that variable $high_p$ is changed only when action NEWSTATE$_p$ is executed, and is set to $current.id_p$. Since p participates in the query phase of operation j there must be a state $\hat{s}$ in between s'' and s_3 such that that $\hat{s}'.high.id_p =$

$\hat{s}'.current.id_p = c_2.id$. By monotonicity of configuration identifiers we have that $s.high.id_p \geq \hat{s}'.high.id_p$ and since $c_1.id < c_2.id$ we have that $s.high.id_p > c_1.id$.

Now, let y_p be the summary due to the acknowledgment value sent by p for the query phase of operation j. Notice that the tag $y_p.tag$ is used by the condenser function ϕ_{maxtag} for the query phase of operation j.

By Invariant 6.5.10, applied to state s using $c = c_1$, we have that $y_p.tag \geq t$. By definition of ϕ_{maxtag} we have that $query\text{-}tag[j] \geq t$.

3. $c_1.id > c_2.id$

   We show that this cannot happen. There are two possible cases:

   (a) $\nexists x \in s''.\mathcal{TotEst}$ such that $c_2.id < x, id < c_1.id$.

   By Invariant 6.3.1 applied to state s'', there exists $W' \in c_2.wqrms$ such that $W' \subseteq c_1.set$. We have that $R \cap W' \neq \emptyset$. Let $p \in R \cap W'$. We have that $p \in c_1.set$.

   In state s_1 the propagate phase of operation i ends. It must be the case that every member of c_1 has $s_1.current.id \geq c_1.id$ and also that every member of c_1 has submitted its state for c_1 prior to the beginning of the query phase of operation j. Since $p \in c_1.set$, there must exist a state $\hat{s}$ preceding s_1 such that $\hat{s}.current_p = c_1$.

   Since $p \in R$ there must exist a state $\hat{s}'$ in between s'' and s_3 such that $\hat{s}'.current_p = c_2$. Since s_1 precedes s'', we have that $\hat{s}$ precedes $\hat{s}'$. By monotonicity of configuration identifiers we must have $\hat{s}.current.id_p \leq \hat{s}'.current.id_p$, that is $c_1.id \leq c_2.id$. This contradicts the hypothesis that $c_1.id > c_2.id$.

   (b) $\exists x \in s''.\mathcal{TotEst}$ such that $c_2.id < x.id < c_1.id$.

   Let c' be the totally established configuration with the smallest identifier intervening between c_2 and c_1 in s''. By definition of c' we have that $c'.id > c_2.id$.

   By Invariant 6.3.1 applied to state s'', there exists $W' \in c_2.wqrms$ such that $W' \subseteq c'.set$. We have that $R \cap W' \neq \emptyset$. Let $p \in R \cap W'$. Since $c' \in s''.\mathcal{TotEst}$ and $p \in c'.set$ we have that $s''.current.id_p \geq c'.id$.

   Since $p \in R$, there must exist a state $\hat{s}$ between s'' and s_3 such that $\hat{s}.current_p = c_2$.

   By monotonicity of configuration identifiers, since s'' precedes $\hat{s}$ we have that $\hat{s}.current.id_p \geq s''.current.id_p$. This implies that $c_2.id \geq c'.id > c_2.id$, which is impossible.

$\square$

In order to prove that the system implements an atomic object we use the following lemma from [65] (Lemma 13.16, page 435).

Lemma 6.5.12 *Let β be a (finite or infinite) sequence of actions of a read/write atomic object external interface. Suppose that β is well-formed, and contains no incomplete operations. Let Π be the set of all operations in β.*

Suppose that $\prec$ is an irreflexive partial ordering of all the operations in Π, satisfying the following properties:

1. *For any operation $i \in \Pi$, there are only finitely many operations j such that $j \prec i$.*
2. *If the response event for operation i precedes the invocation event for operation j in β, then it cannot be the case that $j \prec i$.*
3. *If i is a write operation in Π and j is any operation in Π, then either $i \prec j$ or $j \prec i$.*
4. *The value returned by each read operation is the value written by the last preceding write operation according to $\prec$ (or a fixed initial value, if there is no such write).*

Then β satisfies the atomicity property.

We can use the above lemma to prove the following result. By Lemma 13.10 of [65] (page 419) we can restrict our attention to executions with no incomplete operations

Theorem 6.5.13 ABD-SYS *implements an atomic read/write object.*

Proof: In order to show that the system implements an atomic object we need to provide a partial order that satisfies Lemma 6.5.12. Let use define the order $\prec$ as follows. First define the tag of an operation i as $tag(i) = prop\text{-}tag[i]$, that is, the tag written in the propagate phase. Order all writes operation in order of *tag* and place each read operation after the write operation with the same tag and before any other write operation (order of read operations in between two consecutive write operations is irrelevant). Place all reads for which there is no write operation with the same tag, before the first write operation.

Next we prove that $\prec$ satisfies Lemma 6.5.12. Let us start with Point 1. Any operation $j \prec i$ must have $tag(j) < tag(i)$. The number of write operations that precede i is bounded by the number of tags which are strictly smaller than $tag(i)$. This is a finite number. The number of reads which have a tag smaller than $tag(i)$ is bounded by the number of read operations completed before operation i is completed. This is also a finite number.

Now consider Point 2. Assume that the response event for an operation i precedes the invocation event for an operation j. Then we have that the propagate phase of operation i precedes the query phase of operation j and by Invariant 6.5.11 we have that the tag returned by the query phase of operation j is greater than or equal to the tag written by the propagate phase of operation i. Since the latter is equal to $tag(i)$ and since the former is less or equal than $tag(j)$ we have that $tag(i) \leq tag(j)$. Thus it cannot be that $j \prec i$ because this means that $tag(j) < tag(i)$.

Consider now Point 3. Assume that i is a write operation and that j is any other operation. Assume by contradiction than neither $i \prec j$ nor $j \prec i$. Then we have that $tag(i) = tag(j)$ and that j is also a write operation (a read operation with the same tag of i is such that $i \prec j$). Since $tag(i) = tag(j)$ and since the process identifier is part of the tag, it must be the case that both operations are requested by the same process. Hence it must be the case that one of the operation, say i, is completed before the other, operation j, is requested. Thus the (completed) propagate phase of operation i precedes the query phase of operation j. Hence by Invariant 6.5.11 we have that the tag returned by the query phase of operation j is greater than or equal to the tag written by the propagate phase of operation i. The latter is equal to $tag(i)$ and the former, by the code, is strictly less than $tag(j)$. Hence we have that $tag(i) < tag(j)$, which contradicts the fact that $tag(i) = tag(j)$.

Finally consider Point 4. Since each read is ordered right after the write with the same tag it is enough to show that a read operation i gets the value written by a write operation j such that $tag(j) = tag(i)$. So let i be a read operation and let j be a write operation with $tag(j) = tag(i)$. It follows by the code that the value returned by operation i is the one written by operation j (because tag and value are updated simultaneously). $\square$

6.6 Remarks

We remark that the intersection property of DC, namely that there exist a read quorum R and a write quorum W of a previous primary configuration both belonging to the next primary configuration comes from the particular application that we have developed. For other applications one might have different (maybe weaker) intersection properties. For example, one might require that the new primary configuration contains a read quorum of the previous one (and not necessarily a write quorum). In our case, we must require both a read quorum and a write quorum in the new primary. If we do not require a read quorum to be in the new configuration but only require a write quorum to be in the new configuration, since write quorums may not intersect, two non-intersecting write quorums might concurrently proceed to two primary configurations, violating the uniqueness of a primary configuration. The same situation can happen if we do not require a write quorum to be in the new configuration but only a read quorum to be in the new configuration, because two read quorums may not intersect. In this latter case it is also possible for a read quorum in an old configuration to read obsolete values; indeed processes in a read quorum can be left behind if newer configurations are established but since they form a read quorum of the configuration they are in, they will be able to read whatever (obsolete) value they have.

It is possible to optimize the state transfer at the beginning of a new configuration. The goal of the state transfer is to obtain all information from previous configuration. Clearly process that join the system have no information about previous configurations. Hence it is useless to wait for them to submit their state before computing the new up to date state.

We remark that the choice of integrating the state transfer into the service has been made because most applications have to perform such state exchange and thus it seems reasonable to do it within the service in order to free the application from the details of such a computation. We did not change the DVS service to also offer integrated state exchange because some applications may not require submission of the current state from every member of a new view or configuration. So it may be useful also to leave to the application control of the state exchange computation.

The above remark is connected to the question: does DC supersede DVS? On one hand DC is more general than DVS because it provides a group communication service that handles configurations instead of views and configurations carry more information than views. On the other hand there are some differences between DC and DVS. We already talked about the difference in the state exchange mechanism. Another difference is in the communication mechanism used by the two services: DVS uses a point-to-point communication mechanism, while DC use a broadcast/convergecast mechanism involving a quorum of processes. Because of these differences we have kept the two services as different services.

The DC service requires every process of a new configuration to submit its state. This is a strong requirement for applications that use quorums to improve availability. However it provides a strong service. It is possible to specify a weaker version of the DC service that requires only a read quorum to submit the state before computing the starting state of a new configuration. We believe that the TO algorithm we have developed in this chapter would still work with this weaker service.

The ABD algorithm does not use the prefix property of message delivery guaranteed by the CS specification. Hence one could use a weaker specification instead of CS to implement DC.

The implementation of DC performs garbage collection when a view becomes totally established (any previous view is discarded and no intersection checks are made with these discarded views). Yeger Lotem *et al.* [89] perform garbage collection when a view becomes established (the process that establishes a view discards all previous ambiguous views). Our garbage collection mechanism, though less efficient than that of [89], allows an easier proof of correctness.

Chapter 7

Dynamic Algorithms

In this chapter we apply the ideas about dynamic configurations developed in Chapter 6 to design a dynamic version of the PAXOS algorithm [61], called DPAXOS, and a dynamic primary copy data replication algorithm implementing an atomic object, called RAB.

Both algorithms are built upon an underlying group communication service; this service, called DLC, is a variation of the DC service (see Chapter 6) which uses "leader configurations" (see Chapter 3) and augments the service with point-to-point communication.

We start the chapter with the DLC service. Section 7.2 provides the DPAXOS algorithm. In Section 7.3 we sketch the RAB algorithm. Finally Section 7.4 contains concluding remarks.

7.1 The DLC specification

In this section we provide a dynamic primary configuration group communication service. This service, which we call DLC, is similar to the DC service; the differences are: (*i*) DLC handles leader configurations instead of read-write quorum configurations (see Chapter 3) and (*ii*) DLC provides point-to-point channels too.

The DLC service is similar to the DC service. The code is provided in Figure 7-1. In Section 7.1.1 we explain the differences between DLC and DC. We also provide a full description of DLC in Section 7.1.2; however the reader who comes from Chapter 6 and reads Section 7.1.1 can safely skip Section 7.1.2.

7.1.1 Differences with DC

There are basically two differences between DC and DLC. The first difference is due to the kind of configurations considered: leader configurations instead of read-write quorum configurations. As a result, the key intersection property becomes the following: for any two created and non dead

DLC

Signature:

Input: $\textsc{submit}(m,\phi,i)_p$, $m \in \mathcal{M}$, $\phi \in \Phi$, $p \in \mathcal{P}$, $i \in \mathit{OID}_p$
$\textsc{ackdlvr}(a,i)_p$, $a \in A$, $i \in \mathit{OID}$, $p \in \mathcal{P}$
$\textsc{submit-state}(s,\phi)_p$, $s \in \mathcal{S}$, $\phi \in \Phi$, $p \in \mathcal{P}$
$\textsc{p2p-recv}(m)_{q,p}$, $m \in \mathcal{M}$, $q,p \in \mathcal{P}$

Internal: $\textsc{createconf}(c)$, $c \in \mathcal{C}$

Output: $\textsc{newconf}(c)_p$, $c \in \mathcal{C}$, $p \in c.set$
$\textsc{newstate}(s)_p$, $s \in \mathcal{S}$, $p \in \mathcal{P}$
$\textsc{respond}(a,i)_p$, $a \in \mathcal{A}$, $i \in \mathit{OID}_p$, $p \in \mathcal{P}$
$\textsc{deliver}(m,i)_p$, $m \in \mathcal{M}$, $i \in \mathit{OID}$, $p \in \mathcal{P}$
$\textsc{p2p-send}(m)_{p,q}$, $m \in \mathcal{M}$, $q,p \in \mathcal{P}$,

State:

$\mathit{created} \in 2^{\mathcal{C}}$, init $\{c_0\}$
for each $p \in \mathcal{P}$:
  $\mathit{cur\text{-}cid}[p] \in \mathcal{G}_\perp$, init g_0 if $p \in P_0$, $\perp$ else
for each $p,q \in \mathcal{P}, g \in \mathcal{G}$:
  $\mathit{p2p\text{-}msgs}[p,g,q] \mathit{seqof}(\mathcal{M})$, initially $\emptyset$
  $\mathit{p2p\text{-}next}[p,g,q] \in \mathbf{N}^{>0}$, initially 1

for each $g \in \mathcal{G}$:
  $\mathit{got\text{-}state}[g] = \mathcal{P} \to \mathcal{S}_\perp$, init everywhere $\perp$
  $\mathit{condenser}[g] = \mathcal{P} \to \Phi_\perp$, init everywhere $\perp$
  $\mathit{state\text{-}dlv}[g] \in 2^{\mathcal{P}}$, init P_0 if $g = g_0$, $\{\}$ else
  $\mathit{pending}[g] \in \mathcal{O}$, init everywhere $\perp$
  $\mathit{attempted}[g] \in 2^{\mathcal{P}}$, init P_0 if $g = g_0$, $\{\}$ else

Derived variables:

$\mathcal{A}tt \in 2^{\mathcal{C}}$, defined as $\{c \in \mathit{created} | \mathit{attempted}[c.id] \neq \emptyset\}$
$\mathcal{E}st \in 2^{\mathcal{C}}$, defined as $\{c \in \mathit{created} | \mathit{state\text{-}dlv}[c.id] \neq \emptyset\}$
$\mathcal{T}ot\mathcal{A}tt \in 2^{\mathcal{C}}$, defined as $\{c \in \mathit{created} | c.set \subseteq \mathit{attempted}[c.id]\}$
$\mathcal{T}ot\mathcal{E}st \in 2^{\mathcal{C}}$, defined as $\{c \in \mathit{created} | c.set \subseteq \mathit{state\text{-}dlv}[c.id]\}$
$\mathit{dead} \in 2^{\mathcal{C}}$ defined as $\mathit{dead} = \{c \in \mathcal{C} | \exists p \in c.set : \mathit{cur\text{-}cid}_p > c.id$ and $p \notin \mathit{attempted}[c.id]\}$.

Actions:

internal $\textsc{createconf}(c)$
Pre: $\forall w \in \mathit{created} : c.id \neq w.id$
  if $c \notin \mathit{dead}$ then
    $\forall w \in \mathit{created}, w.id < c.id$:
      $w \in \mathit{dead}$ or
      $(\exists x \in \mathcal{T}ot\mathcal{E}st\text{: } w.id < x.id < c.id)$ or
      $(\exists Q \in w.qrms\text{: } Q \subseteq c.set)$
    $\forall w \in \mathit{created}, w.id > c.id$
      $w \in \mathit{dead}$ or
      $(\exists x \in \mathcal{T}ot\mathcal{E}st\text{: } c.id < x.id < w.id)$ or
      $(\exists Q \in c.qrms\text{: } Q \subseteq w.set)$
Eff: $\mathit{created} := \mathit{created} \cup \{c\}$

output $\textsc{newconf}(c)_p$, $p \in c.set$
Pre: $c \in \mathit{created}$
  $c.id > \mathit{cur\text{-}cid}[p]$
Eff: $\mathit{cur\text{-}cid}[p] := c.id$
  $\mathit{attempted}[c.id] := \mathit{attempted}[c.id] \cup \{p\}$

input $\textsc{submit-state}(s,\phi)_p$
Eff: if $\mathit{cur\text{-}cid}[p] \neq \perp$ and
    $\mathit{got\text{-}state}[\mathit{cur\text{-}cid}[p]](p) = \perp$ then
      $\mathit{got\text{-}state}[\mathit{cur\text{-}cid}[p]](p) := s$
      $\mathit{condenser}[\mathit{cur\text{-}cid}[p]](p) := \phi$

output $\textsc{newstate}(s)_p$ *choose* c
Pre: $c.id = \mathit{cur\text{-}cid}[p]$
  $c \in \mathit{created}$
  $\forall q \in c.set$, $\mathit{got\text{-}state}[c.id](q) \neq \perp$
  let $f = \mathit{condenser}[c.id](p) | c.set$
  $s = f(\mathit{got\text{-}state}[c.id])$
  $p \notin \mathit{state\text{-}dlv}[c.id]$
Eff: $\mathit{state\text{-}dlv}[c.id] := \mathit{state\text{-}dlv}[c.id] \cup \{p\}$

input $\textsc{submit}(m,\phi,i)_p$
Eff: if $\mathit{cur\text{-}cid}[p] \neq \perp$ then
    $\mathit{pending}[\mathit{cur\text{-}cid}[p]](i)$
      $:= (m, \phi, \emptyset, \lambda(x) : x \to \perp, \texttt{false})$

output $\textsc{deliver}(m,i)_p$ *choose* g
Pre: $g = \mathit{cur\text{-}cid}[p]$
  $p \notin \mathit{pending}[g](i).\mathit{dlv}$
  $\mathit{pending}[g](i).\mathit{msg} = m$
Eff: $\mathit{pending}[g](i).\mathit{dlv} := \mathit{pending}[g](i).\mathit{dlv} \cup \{p\}$

input $\textsc{ackdlvr}(a,i)_p$
Eff: if $\mathit{cur\text{-}cid}[p] \neq \perp$ and
    $\mathit{pending}[\mathit{cur\text{-}cid}[p]](i).\mathit{ack}(p) \neq \perp$ then
      $\mathit{pending}[\mathit{cur\text{-}cid}[p]](i).\mathit{ack}(p) := a$

output $\textsc{respond}(r,i)_p$ *choose* c, Q
Pre: $c.id = \mathit{cur\text{-}cid}[p]$
  $c \in \mathit{created}$
  $i \in \mathit{OID}_p$
  $\mathit{pending}[c.id](i).\mathit{rsp} = \texttt{false}$
  $Q \in c.\mathit{qrms}$
  let $f = \mathit{pending}[c.id](i).\mathit{ack}$
  $\forall q \in Q : f(q) \neq \perp$
  $r = \mathit{pending}[c.id](i).\mathit{cnd}(f|Q)$
Eff: $\mathit{pending}[c.id](i).\mathit{rsp} := \texttt{true}$

input $\textsc{p2p-send}(m)_{p,q}$
Eff: if $\mathit{cur\text{-}cid}[p] \neq \perp$ then
    append m to $\mathit{p2p\text{-}msgs}[p, \mathit{cur\text{-}cid}[p], q]$

output $\textsc{p2p-recv}(m)_{p,q}$, choose g
Pre: $g = \mathit{cur\text{-}cid}[q]$
  $\mathit{mp2p\text{-}msgs}[p,g,q](\mathit{p2p\text{-}next}[p,g,q])$
Eff: $\mathit{p2p\text{-}next}[p,g,q] := \mathit{p2p\text{-}next}[p,g,q] + 1$

Figure 7-1: The DLC specification

configurations c_1 and c_2, with $c_1.id < c_2.id$, either there exists an intervening totally established configuration or a quorum of c_1 is included in the membership set of c_2. Invariant 7.1.1 formalizes the above key property (this invariant is given in Section 7.1.3).

The second difference is that DLC offers also point-to-point communication. That is, a process p can send a message m to another process q, provided that both p and q are in the same configuration. Actions $\text{P2P-SEND}(m)_{p,q}$ and $\text{P2P-RECV}(m)_{p,q}$ of DLC implement the point-to-point communication mechanism (this portion of the code is not present in DC).

The rest of the DLC specification is the same as the DC specification.

7.1.2 Full description of DLC

In this section we provide a full description of DLC. The reader who comes from Chapter 6 and has read Section 7.1.1 can safely skip this section. The description we provide here is similar to the description of the DC service provided in Section 6.2.
Prior to providing the code for the DC specification, we need some notation and definitions, which we introduce in the following.

Let OID be a set of operation identifiers, partitioned into sets OID_p, $p \in \mathcal{P}$. We denote by $\mathcal{M}_c \subseteq \mathcal{M}$ the set of messages that clients may use for communication.

Let $\mathcal{A}$ be a set of "acknowledgment" values and let $\mathcal{R}$ be a set of "response" values. A *condenser function* is a function from $(\mathcal{P} \to \mathcal{A}_\perp)$ to $\mathcal{R}$. Let Φ be the set of all condenser functions. Let $\mathcal{S}$ be the set of all possible states of the clients (a state of $\mathcal{S}$ does not need to be the entire client's state, but it may contain only the relevant information in order for the application to work). The DC specification uses a condenser function also to compute the starting state of a new configuration; hence we assume that $\mathcal{S} \subseteq \mathcal{A}$ and also $\mathcal{S} \subseteq \mathcal{R}$. Given a function $f : \mathcal{P} \to D$ from the set of processes $\mathcal{P}$ to some domain D and given a subset $P \subseteq \mathcal{P}$, we write $f|P$ to denote the function $f' : P \to D$, defined as $f'(p) = f(p)$ for $p \in P$.

The following data type is used to describe operations: $\mathcal{D} = \mathcal{M} \times \Phi \times 2^{\mathcal{P}} \times (\mathcal{P} \to \mathcal{A}_\perp) \times Bool$ and we let $\mathcal{O} = OID \to \mathcal{D}_\perp$. Given an operation descriptor, selectors for the components are *msg*, *cnd*, *dlv*, *ack*, and *rsp*.

Next we provide remarks and an informal description of this code. We start with the derived variables.

A configuration $c \in \mathcal{A}tt$ is said to be *attempted.* For an attempted configuration c there exists at least one process p that has executed action $\text{NEWCONF}(c)_p$ and thus we have that $p \in attempted[c.id]$; when this holds we say that c *is attempted at* p or that p *has attempted* c. A configuration $c \in \mathcal{T}ot\mathcal{A}tt$ is said to be *totally attempted.* A totally attempted configuration is a configuration that is attempted at all members of the configuration.

A configuration $c \in \mathcal{E}st$ is said to be *established.* For an established configuration c there exists at least one process p that has executed action $\textsc{newstate}(s)_p$ and thus we have that $p \in$ *state-dlv*$[c.id]$; when this holds we say that c *is established at* p or that p *has established* c. A configuration $c \in \mathcal{TotEst}$ is said to be *totally established*. A totally established configuration is a configuration that is established at all members of the configuration.

A *dead* configuration c is a configuration for which a member process p went on to newer configurations, that is, it executed action $\textsc{newconf}(c')_p$ with $c'.id > c.id$, before receiving the notification, that is the $\textsc{newconf}(c)_p$ event, for configuration c.

Now we comment on the transitions.

Action $\textsc{createconf}(c)$ creates a new configuration c. The first precondition simply requires this new configuration to have a brand new identifier. The second precondition of this action is the key to our specification. It states that when a configuration c is created it must either be already dead or for any other configuration w such that there are no intervening totally established configurations, the earlier configuration (i.e., the one with smaller identifier) has at least one quorum included in the membership set of the later configuration (i.e., the one with bigger identifier).

Action $\textsc{newconf}(c)_p$ delivers a created configuration c to the client process p. The precondition of this action makes sure that configurations are delivered in order of configuration identifiers. We notice that because of this precondition, when a configuration c is dead because a process q went on to newer configurations, we have that process q can no longer execute action $\textsc{newconf}(c)_q$.

Once a configuration c has been delivered to a client process p, the client process p is supposed to submit its current state s and a condenser function ϕ, by means of action $\textsc{submit-state}(s,\phi)_p$. Once all the processes have submitted their current states, the condenser function ϕ is used to compute the starting state of configuration c for process p. The code of this action just memorizes the state s and the condenser function ϕ for the current configuration of process p.

Action $\textsc{newstate}(s)_p$ computes the starting state for a configuration c. The precondition of this action requires that all processes q in the membership of configuration c have submitted their state for configuration c. The starting state s of configuration c for process p is then computed by applying the condenser function that process p has submitted to the service with action $\textsc{submit-state}(s,\phi)_p$. Variable *state-dlv*$[c.id]$ records the fact that p has received the starting state for configuration c.

We remark that for a dead configuration c there is at least one process that does not execute action $\textsc{newconf}(c)_p$ and thus does not submit its state for c with action $\textsc{submit-state}(s,\phi)_p$. This implies that action $\textsc{newstate}(s)_q$ cannot be executed for any process q. This is why such configurations are called "dead".

The remaining actions are used to handle the requests of clients. We refer to the process of handling such a request, which involves the participation of a quorum of processes, as an "operation". To request the execution of an operation a client process p uses action $\textsc{submit}(m,\phi,i)_p$. The parameters of this actions are as follows: m is a message describing the operation that p needs to perform; ϕ is a condenser function to be used to compute a response value for p when a quorum of processes have provided acknowledgment values to p's message m; i is an operation identifier needed to distinguish operations (every requested operation has a unique operation identifier). We say "operation i" to indicate the operation requested with action $\textsc{submit}(m, \phi, i)_p$. For configuration c and operation i, the variable $\mathit{pending}[c.id](i)$ contains an operation descriptor. The code of action $\textsc{submit}(m, \phi, i)_p$ sets this operation descriptor to a default value.

We now provide an explanation for each component of an operation descriptor. Let d be an operation descriptor for operation i requested by p in configuration c. $d.msg$ is the message that describes the request of p; such a message will be delivered to all members of the configuration c. $d.cnd$ is the condenser function that will be used to compute the response for the operation once a quorum of processes has provided acknowledgment values. $d.dlv$ is the set of processes to which the message $d.msg$ has been delivered; initially this is set to an empty set by action $\textsc{submit}(m, \phi, i)_p$. $d.ack$ contains the acknowledgment values received; initially this is a vector of $\perp$ values. Finally, $d.rsp$ is a flag indicating whether or not the client p that requested the operation has received a response for the operation.

Action $\textsc{deliver}(m, i)_p$ delivers the message m of operation i to process p. The code of this action updates the operation descriptor d for operation i by adding process p to the set $d.dlv$.

Processes that receive the message m for an operation i are supposed to provide an acknowledgment value a with action $\textsc{ackdlvr}(a, i)_p$. The code of this action records the acknowledgment value a of process p into the vector $d.ack$, where d is the operation descriptor for operation i.

Action $\textsc{respond}(r, i)$ provides a response r to process p for the operation i previously submitted by p. The precondition of this action requires that a quorum Q has provided acknowledgment values. Then the value r is computed by applying the condenser function provided by p at the time of the submission, to the acknowledgment values of processes in Q. At this point the operation has been serviced and the rsp component is set to `true`.

The code that handles point-to-point communication, is provided by actions $\textsc{p2p-send}(m)_{p,q}$ and $\textsc{p2p-recv}(m)_{p,q}$. State variable $\mathit{p2p\text{-}msgs}[p, g, q]$ is used to record the messages sent. When action $\textsc{p2p-send}(m)_{p,q}$ is executed in a configuration whose identifier is g, message m is added to $\mathit{p2p\text{-}msgs}[p, g, q]$ which contains a queue of messages sent by p to q in configuration g. Action $\textsc{p2p-recv}(m)_{q,p}$ delivers message m to q.

7.1.3 Invariant of DLC

In this section we provide a key invariant of DLC. The property stated by this invariant is used to prove correct the applications that we build on top of DLC.

The second precondition of $\textsc{createconf}(c)$ is the key to our specification. It states that when a configuration c is created it must either be already dead or for any other configuration w such that there are no intervening totally established configurations, the earlier configuration (i.e., the one with smaller identifier) has one quorum whose members are included in the membership set of the later configuration (i.e., the one with bigger identifier). The above precondition enables us to prove the following key invariant:

Invariant 7.1.1 *In any reachable state of* DLC*, the following is true. Let* $c_1, c_2 \in created \setminus dead$*, with* $c_1.id < c_2.id$*. Then either there exists* $w \in \mathcal{T}ot\mathcal{E}st, c_1.id < w.id < c_2.id$*, or else there exists a quorum* $Q \in c_1.qrms$ *such that* $Q \subseteq c_2.set$*.*

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. In the initial state the invariant is vacuously true because there are no two configurations $c_1, c_2 \in created$ such that $c_1.id < c_2.id$.
For the inductive step assume that the invariant is true in a state s. We need to prove that the invariant is true in s' for any step (s, π, s'). The only action that we need to worry about is CREATECONF(c), where $c = c_1$ or $c = c_2$, because it creates a new configuration (otherwise the invariant is true in s' by the inductive hypothesis). So assume that $\pi =$CREATECONF(c). The invariant follows easily from the precondition of π. ∎

7.2 Dynamic PAXOS algorithm

In this section we present DPAXOS, an algorithm that solves the consensus problem and that adapts well to dynamic distributed systems. The DPAXOS algorithm is derived from Lamport's PAXOS algorithm because it uses the same strategy. We refer the reader to the paper by Lamport [61] for a complete and colorful presentation of the PAXOS algorithm; the papers [29, 30] provide a less colorful description of the PAXOS algorithm using the I/O automaton model. For the sake of completeness, in this section we provide a brief description of the PAXOS algorithm.

We first provide a formal definition of the consensus problem in Section 7.2.1. Then in Section 7.2.2 we recall the original PAXOS algorithm. Section 7.2.3 provides the DPAXOS algorithm and Section 7.2.4 the proof of correctness.

7.2.1 Distributed Consensus

Distributed consensus is a fundamental problem in distributed computation. Roughly speaking the problem is the one of reaching agreement among the members of a distributed system; such agreement is often necessary for distributed applications (e.g., data replication, airline reservation system, distributed transactions).

Next we give a formal definition of the consensus problem that we consider in this paper. Process p starts the computation with an input value $v_p \in \mathcal{X}$, where $\mathcal{X}$ is the set of all possible initial values.

Given a particular execution α, we denote by $\mathcal{X}_\alpha \subseteq \mathcal{X}$ the set of the initial values of processes in α.

Each process has a state variable called *decision* which is a write-once variable initially not written. Processes have to write their *decision* variable in such a way that three conditions are satisfied:

- **Agreement:** All the written *decision* variables are set to the same value.
- **Validity:** Any written *decision* variable is set to a value in $\mathcal{X}_\alpha$.

Since the termination condition is a liveness issue and we don't address liveness, we don't provide a formal definition. Informally, the termination condition requires that if in a given state s a configuration c is totally established, c is the configuration with the biggest identifier in $s.created$, no other configurations are created after s, all processes of c are alive in state s, and no failures happen after s, then all processes that are members of c eventually write the *decision* variable, provided that the execution is a fair execution.

7.2.2 The original PAXOS algorithm

The PAXOS algorithm does not use a group communication service, so there are no views or configurations; it relies on an external leader elector module which provides (unreliable) information about the current status of the underlying distributed system, i.e., tells the current membership and the leader, to each process. Termination is guaranteed only when there are no failures, and the external leader elector provides reliable information on the system, for a sufficiently long time.

The basic idea is to have processes propose values until one of them is accepted by a majority of the processes; that value is the final output value. Any process may propose a value by initiating a *round* for that value. The process initiating a round is said to be the *leader* of that round while all processes, including the leader itself, are said to be *agents* for that round.

Since different rounds may be carried out concurrently (the leader elector is not reliable hence several processes may concurrently consider themselves leaders), we need to distinguish them. Every round has a unique identifier. Next we formally define these round identifiers. A *round number* is a pair (x, i) where x is a nonnegative integer and i is a process identifier. The set of round numbers is denoted by $\mathcal{R}$. A total order on elements of $\mathcal{R}$ is defined by $(x, i) < (y, j)$ iff $x < y$ or, $x = y$ and $i < j$.

We say that round r *precedes* round r' if $r < r'$. If round r precedes round r' then we also say that r is a *previous* round, with respect to round r'. We remark that the ordering of rounds is not related to the actual time the rounds are conducted. It is possible that a round r' is started at some point in time and a previous round r, that is, one with $r < r'$, is started later on.

Every round in the algorithm is tagged with a unique round number. Every message sent by the leader or by an agent for a round (with round number) $r \in \mathcal{R}$, carries the round number r so that no confusion among messages belonging to different rounds is possible.

Informally, the steps for a round are the following.

1. To initiate a round, the leader sends a "Collect" message to all agents announcing that it wants to start a new round with round number r and at the same time asking for information about previous rounds in which agents may have been involved.

2. An agent that receives a message sent in step 1 from the leader of the round, responds with a "Last" message giving its own information about rounds previously conducted, namely the last round r' for which the agent made a commitment and the value v of that round. With this, the agent makes a kind of commitment for this particular round that may prevent it from accepting (in step 4) the value proposed in some other round. If the agent is already committed for a round with a bigger round number then it informs the leader of its commitment with an "OldRound" message.

3. Once the leader has gathered information about previous rounds from a majority of agents, it decides, according to some rules, the value to propose for its round and sends to all agents a "Begin" message announcing the value v for round r and asking them to accept it. In order for the leader to be able to choose a value for the round it is necessary that initial values be provided. If no initial value is provided, the leader must wait for an initial value before proceeding with step 3. The set of processes from which the leader gathers information is called the *info-quorum* of the round.

4. An agent that receives a message from the leader of the round sent in step 3, responds with an "Accept" message by accepting the value proposed in the current round r, unless it is committed for a later round and thus must reject the value proposed in the current round. In the latter case the agent sends an "OldRound" message to the leader indicating the round r' for which it is committed.

5. If the leader gets "Accept" messages from a majority of agents, then the leader sets its own output value to the value proposed in the round. At this point the round is successful. The set of agents that accept the value proposed by the leader is called the *accepting-quorum*.

Since a successful round implies that the leader of the round reaches a decision, after a successful round the leader still needs to do something, namely to broadcast the decision. Thus, once the leader has made a decision it broadcasts a "Success" message announcing the value for which it has decided. An agent that receives a "Success" message from the leader makes its decision choosing the value of the successful round. We use also an "Ack" message sent from the agent to the leader, so that the leader can make sure that everyone knows the outcome.

The most important issue is about the values that leaders propose for their rounds. Indeed, since the value of a successful round is the output value of some processes, we must guarantee that the values of successful rounds are all equal in order to satisfy the agreement condition of the consensus problem. This is the tricky part of the algorithm and basically all the difficulties derive from solving this problem. Consistency is guaranteed by choosing the values of new rounds exploiting the information about previous rounds from at least a majority of the agents so that, for any two rounds, there is at least one process that participated in both rounds.

In more detail, the leader of a round chooses the value for the round in the following way. In step 1, the leader asks for information and in step 2 an agent responds with the number of the latest round in which it accepted the value and with the accepted value or with round number $(0, j)$ and $\perp$ if the agent has not yet accepted a value. Once the leader gets such information from a majority of the agents (which is the info-quorum of the round), it chooses the value for its round to be equal to the value of the latest round among all those it has heard from the agents in the info-quorum or equal to its initial value if all agents in the info-quorum were not involved in any previous round. Moreover, in order to keep consistency, if an agent tells the leader of a round r that the last round in which it accepted a value is round r', $r' < r$, then implicitly the agent commits itself not to accept any value proposed in any other round r'', $r' < r'' < r$.

Given the above setting, if r' is the round from which the leader of round r gets the value for its round, then, when a value for round r has been chosen, any round r'', $r' < r'' < r$, cannot be successful; indeed at least a majority of the processes are committed for round r, which implies that at least a majority of the processes are rejecting round r''. This, along with the fact that info-quorums and accepting-quorums are majorities, implies that if a round r is successful, then any round with a bigger round number $\tilde{r} > r$ is for the same value. Indeed the information sent by processes in the info-quorum of round $\tilde{r}$ is used to choose the value for the round, but since info-quorums and accepting-quorums share at least one process, at least one of the processes in the info-quorum of round r' is also in the accepting-quorum of round r. Indeed, since the round is successful, the accepting-quorum is a majority. This implies that the value of any round $\tilde{r} > r$ must be equal to the value of round r, which, in turn, implies agreement.

Instead of majorities for info-quorums and accepting-quorums, any quorum system can be used (DPAXOS uses the quorum system of the configuration). Indeed the only property that is required is that there be a process in the intersection of any info-quorum with any accepting-quorum.

To end up with a decision value, rounds must be started until at least one is successful.

7.2.3 The DPAXOS algorithm

The DPAXOS algorithm borrows the basic ideas of the PAXOS algorithm, but it is built upon the DLC group communication service. Thus it exploits the properties guaranteed by such a service. A round of the DPAXOS algorithm is similar to that of the PAXOS algorithm with the following major differences: (i) since any time that the leader changes the DLC service provides a new configuration, we have that at most one round is conducted in each configuration (hence we do not distinguish rounds and configurations); (ii) the first part of a round, whose purpose is to find a value that the leader proposes in the round, is no longer necessary, because the group communication service provides, with the starting state of a new configuration, a value that can be safely proposed by the leader. Thus in DPAXOS the leader needs only to send "Begin" messages and collect the "Accept" messages.

Because of (i) we no longer need to worry about processes committing to reject rounds or to make sure that messages belonging to different rounds do not interfere: by the properties of the DLC group communication service we have that processes receive and send messages only in the current configuration. Since in each configuration only one round is conducted, no interferences are possible and older rounds are automatically rejected. Configuration identifiers can be used as round numbers (and viceversa). Because of (ii) a round of DPAXOS is shorter than a round of PAXOS (the first part of the round is basically done while establishing the new configuration).

As a result of the above, the code of DPAXOS is simpler and much shorter than the code of PAXOS as implemented in [29, 30].

Since in each configuration only one round is run, round numbers in DPAXOS are configuration identifiers. Hence the set of round numbers is $\mathcal{R} = \mathcal{G}$. We say that round r *precedes* round r' if $r < r'$. If round r precedes round r' then we also say that r is a *previous* round, with respect to round r'.

For the DPAXOS algorithm, the set $\mathcal{S}$ consists of pairs $\langle r, v\rangle$, where $r \in \mathcal{G}$ and $v \in \mathcal{X}$.

The code, DLC-TO-PAXOS$_p$, is shown in Figure 7-2. The overall system DPAXOS consists of the composition of DLC and automaton DLC-TO-PAXOS$_p$ for each $p \in \mathcal{P}$.

Next we provide an informal description of the code.

We start by describing the state variables. Variable $current_p$ contains the current configuration for process p; if process p runs a round, then the round number is given by $current.id_p$. Variable $rnd\text{-}val_p$ contains the value that process p proposes in the current round. Variable $decision_p$ contains the decision of process p. Variable $used\text{-}ids_p \subseteq OID_p$ is a set of identifiers used to distinguish operations that process p submits to the DLC service. Variable $last\text{-}rnd_p$ contains the last round for which process p has accepted the value. Variable $last\text{-}val$ contains the value of round $last\text{-}rnd$.

The DLC-TO-PAXOS$_p$ has a reconfiguration phase, which starts when process p is notified of a new configuration and ends when process p is given the starting state for the new configuration. When not in reconfiguration phase process p performs normal processing. Variable $status_p$ is used to switch from reconfiguration phase to normal processing. For normal processing we have $status_p = normal$.

Variable $mode_p$ is used by the leader to go through the steps of a round.

DLC-TO-PAXOS

Signature:

Input: $\text{NEWCONF}(c)_p$, $c \in \mathcal{C}$, $p \in c.set$
$\text{NEWSTATE}(s)_p$, $s \in \mathcal{S}$, $p \in \mathcal{P}$
$\text{DELIVER}(m,i)_p$, $m \in \mathcal{M}$, $i \in OID$, $p \in \mathcal{P}$
$\text{RESPOND}(a,i)_p$, $a \in \mathcal{A}$, $i \in OID_p$, $p \in \mathcal{P}$
$\text{P2P-RECV}(m)_{q,p}$, $m \in \mathcal{M}$, $q,p \in \mathcal{P}$

Internal: NEWROUND_p, $p \in \mathcal{P}$

Output: $\text{SUBMIT}(m,\phi,i)_p$, $m \in \mathcal{M}$, $\phi \in \Phi$, $p \in \mathcal{P}$, $i \in OID_p$
$\text{ACKDLVR}(a,i)_p$, $a \in A$, $i \in OID$, $p \in \mathcal{P}$
$\text{SUBMIT-STATE}(s,\phi)_p$, $s \in S$, $\phi \in \Phi$, $p \in \mathcal{P}$
$\text{P2P-SEND}(m)_{p,q}$, $m \in \mathcal{M}$, $q,p \in \mathcal{P}$

State:

$current \in \mathcal{C}_\perp$ init c_o if $p \in P_0$, else $\perp$
$rnd\text{-}val \in \mathcal{X}$, initially v_p
$decision \in \mathcal{X}$, initially $\perp$
$used\text{-}ids \subseteq OID$, initially $\emptyset$
$ack\text{-}q$, $seqof(A \times OID)$, init λ
$ack\text{-}d$, either "*ack*" or $\perp$, init $\perp$

$last\text{-}rnd \in \mathcal{G}$, initially g_0 if $p \in P_0$, else $\perp$
$last\text{-}val \in \mathcal{X}$, initially v_p
$status \in \{normal, exch\text{-}ready, exch\text{-}wait\}$, init $normal$
$mode \in \{wait, newround, collect, begincast, decided\}$,
init $newround$ for $p = c_0.ldr$, $wait$ else
$ack\text{-}success(q)$, a boolean, initially `false` for all $q \in \mathcal{P}$

Actions:

input $\text{NEWCONF}(c)_p$
Eff: $current := c$
$status := exch\text{-}ready$
$ack\text{-}q := \lambda$
if $p = c.ldr$ and $mode \neq decided$ then
$mode := newround$

output $\text{SUBMIT-STATE}(\langle r,v\rangle, \phi_{state})_p$
Pre: $status = exch\text{-}ready$
$r = last\text{-}rnd$
$v = last\text{-}val$
Eff: $status := exch\text{-}wait$

input $\text{NEWSTATE}(\langle r,v\rangle)_p$
Eff: $status := normal$
$last\text{-}rnd := current.id$
$last\text{-}val := v$
$rnd\text{-}value := v$
$Hfrom(current.id) = r$
$Hvalue(current.id) = v$

internal NEWROUND_p
Pre: $mode = newround$
$status = normal$
Eff: $mode := begincast$

output $\text{SUBMIT}(\langle\text{“}begin\text{”}, v\rangle, \phi_{begin}, i)_p$
Pre: $current \neq \perp$
$i \notin used\text{-}ids$
$status = normal$
$v = rnd\text{-}value$
$mode = begincast$
Eff: $used\text{-}ids := used\text{-}ids \cup \{i\}$
$mode := wait$

input $\text{DELIVER}(\langle\text{“}begin\text{”}, v\rangle, i)_p$
Eff: append $(\langle\text{“}accept\text{”}\rangle, i)$ to $ack\text{-}q$
$last\text{-}rnd := current.id$
$last\text{-}val := v$

output $\text{ACKDLVR}(a,i)_p$
Pre: $head(ack\text{-}q) = (a,i)$
Eff: $ack\text{-}q := tail(ack\text{-}q)$

input $\text{RESPOND}(\langle\text{“}begin\text{”}, Q\rangle, i)_p$
Eff: $decision := rnd\text{-}value$
$mode := decided$
$Haccquo(current.id) := Q$

output $\text{P2P-SEND}(v)_{p,q}$
Pre: $mode = decided$
$v = decision$
$ack\text{-}success(q) =$ `false`
Eff: none

input $\text{P2P-RECV}(v)_{q,p}$
Eff: $decision := v$
$mode := decided$
$ack\text{-}d :=$ "*ack*"

output $\text{P2P-SEND}(\text{“}ack\text{”})_{p,q}$
Pre: $ack\text{-}d =$ "*ack*"
Eff: $ack\text{-}d := \perp$

input $\text{P2P-RECV}(\text{“}ack\text{”})_{q,p}$
Eff: $ack\text{-}success(q) :=$ `true`

Figure 7-2: The DLC-TO-PAXOS code.

Variable $ack\text{-}q_p$ is a queue of acknowledgment values to be sent to the DLC service. Variable $ack\text{-}d_p$ is used to send back to the leader an acknowledgment for the decision. Variable $success\text{-}ack_p$ is used by the leader to record those processes that have sent an acknowledgment for the decision.

Next we describe the transitions. We start with the transitions for the reconfiguration phase.

Action $\text{NEWCONF}(c)_p$ notifies process p of a new configuration c. Process p enters the reconfiguration phase by changing its status to *exch-ready*. It also resets queue $ack\text{-}q$ in order to stop sending acknowledgments for older rounds. Moreover if process p is the leader of the new configuration and it has not yet reached a decision, then it prepares itself, by setting $mode_p$ to *newround*, to start a new round when the normal processing mode will be re-entered.

With action $\text{SUBMIT-STATE}(\langle r, v\rangle, \phi_{state})_p$ process p submits its current state to the DLC service. The relevant information submitted to the service consists of the last round r for which process p has accepted a value (i.e., has sent an "*accept*" message) and the value v of that round. The condenser function ϕ_{state} collects all the states submitted by processes in the current configuration and computes the value to propose in the current round. Formally it is a function that takes a set of pairs $W = \{\langle r, v\rangle | r \in \mathcal{G}, v \in \mathcal{X}\}$ and returns a pair $\langle r', v'\rangle \in W$ where r' is such that $r' \geq r$ for all $\langle r, v\rangle \in W$. At this point process p has to wait for the DLC service to deliver the starting state. Hence it sets $status_p$ to *exch-wait*.

Action $\text{NEWSTATE}(\langle r, v\rangle)_p$ delivers to process p the starting state for p's current configuration. This starting state contains the value v to propose in the round for the current configuration. Normal processing is re-entered by setting $status_p = normal$.

Next we describe the transitions for normal processing, where rounds are run.

Action NEWROUND_p starts a new round; in order to do this, the leader of the current configuration, say process p, must be ready to start a new round, that is $mode_p$ must be equal to *newround*. The effect is just to change the $mode_p$ to *begincast*. Process p is now ready to send a *begin* message for the current round.

Action $\text{SUBMIT}(\langle\text{"}begin\text{"}, v\rangle, \phi_{begin}, i)_p$ takes care of sending the *begin* message through the DLC service. The condenser function ϕ_{begin} is a function that takes a set of acknowledgment values $\{\langle\text{"}accept\text{"}, i\rangle_q | q \in Q\}$ for some set of processes $Q \subseteq \mathcal{P}$ and returns $\langle\text{"}begin\text{"}, Q\rangle$. After the execution of this action process p has $mode_p = wait$ because it needs to wait for acknowledgment values.

Action $\text{DELIVER}(\langle\text{"}begin\text{"}, v\rangle, i)_p$ is an input action from the DLC service, which delivers the *begin* message from another process. The effect of this action is to put the acknowledgment value $\langle\text{"}accept\text{"}, i\rangle$ into the queue $ack\text{-}q_p$, from which it will be sent back to the DLC service by action $\text{ACKDLVR}(a, i)_p$.

Action $\text{RESPOND}(\langle\text{"}begin\text{"}, Q\rangle, i)_p$ is an input action from the DLC service which provides the response to the "*begin*" message previously sent with action $\text{SUBMIT}(\langle\text{"}begin\text{"}, v\rangle, \phi_{begin}, i)_p$. This action tells the leader that processes in the quorum Q have accepted the current round. At this point the leader can make a decision and thus sets the $decision_p$ variable to the value $rnd\text{-}value_p$ proposed in the current round.

The remaining actions are used to spread a decision to all members of the current configuration once the leader has reached a decision. Action $\text{P2P-SEND}(v)_{p,q}$ is used by the leader p to send the decision $v = \mathit{decision}_p$ to a process q that does not know yet the decision. Action $\text{P2P-RECV}(v)_{p,q}$ is executed by process q when it receives a decision v from the leader p; process q sets its $\mathit{decision}_q$ variable and sets its mode_q to *decided*. Then, process q sends an acknowledgment back to the leader with action $\text{P2P-SEND}(\text{``}\mathit{ack}\text{''})_{q,p}$ and this acknowledgment is received by the leader with action $\text{P2P-RECV}(\text{``}\mathit{ack}\text{''})_{q,p}$.

We augment the code with the following history variables:

- $\mathit{Hvalue}(r)_p \in \mathcal{X} \cup \perp$, initially v_p for $r = g_0$ and $p = c_0.\mathit{ldr}$ and $\perp$ elsewhere. This variable records the value for round/configuration r.
- $\mathit{Hfrom}(r)_p \in \mathcal{R} \cup \perp$, initially $\perp$ for all r, p. This variable records the round/configuration from which $\mathit{Hvalue}(r)$ is taken.
- $\mathit{Haccquo}(r)_p$, a subset of $\mathcal{P}$ or $\perp$, initially $\perp$ for all r, p. This variable records the accepting-quorum of round r.

We conclude with a few remarks. Submitting the state for a new configuration corresponds to sending a "Last" message in the original PAXOS algorithm. Notice that there is no need to commit to reject older rounds, because this is automatically guaranteed by the configuration oriented communication of the DLC service. Submitting the *begin* message to the DLC service corresponds to broadcasting a "Begin" message in the original PAXOS algorithm. Sending the acknowledgment value $\langle\text{``}\mathit{accept}\text{''}, i\rangle$ for a "*begin*" operation, corresponds to sending an "Accept" message in the original PAXOS algorithm.

Finally we remark that the DLC service encapsulates in the state-exchange mechanism part of the PAXOS algorithm. This is done by means of the condenser function ϕ_{state} which computes the value of the latest configuration for which processes member of the new configuration have accepted a value. This computation is a key point in the original PAXOS algorithm.

7.2.4 Proof of correctness for DPAXOS

In this section, we prove the correctness of DPAXOS. We recall that since at most one round is run in a configuration, we use configuration identifiers as round numbers (round numbers are elements of the set $\mathcal{G}$). Also, we say that a configuration c is *successful* in a state s when $s.\mathit{Haccquo}(c.id) \neq \perp$; informally this means that the round conducted in the configuration is successful, and a decision is made by the leader.

Next we provide invariants needed to prove agreement. We start with some basic invariants.

Invariant 7.2.1 *(*DPAXOS*)*

In any reachable state the following is true. Let c be a configuration established at a process p and such that $c.id > g_0$. Then $Hvalue(c.id)_p \neq \bot$.

Proof: Since configuration c is established at p and $c.id > g_0$, we have that action $\text{NEWSTATE}(r, v)_p$ for some $r \in \mathcal{G}$ and $v \in \mathcal{X}$, with $v \neq \bot$, has been executed (this is not true for $c = c_0$). By the code of this action we have that $Hvalue(c.id)_p = v$. ∎

Invariant 7.2.2 *(*DPAXOS*)*

In any reachable state the following is true. Let c be a successful configuration and let $p = c.ldr$. Then $Haccquo(c.id)_p \neq \bot$ and c is established at $Haccquo(c.id)_p$ and at p.

Proof: In order for a configuration c to be successful, the leader $p = c.ldr$ must propose a value. Clearly it must be that c is established at p. In order for a configuration c to be successful, the leader p must execute action $\text{RESPOND}(\langle\text{``}begin\text{''}, Q\rangle, i)_p$ which sets $Haccquo(c.id)_p$ to Q. Clearly any process of Q must have established c. ∎

Invariant 7.2.3 *(*DPAXOS*)*

In any reachable state the following is true. Let c be an established configuration such that $g_0 < c.id$. Then for any two processes p, q that have established c, we have $Hvalue(c.id)_p = Hvalue(c.id)_q$.

Proof: Process q sets the $Hvalue(c.id)_q$ to v when $\text{NEWSTATE}(\langle r, v\rangle)_q$ for configuration c is executed. Process p sets the $Hvalue(c.id)_p$ to v' when $\text{NEWSTATE}(\langle r', v'\rangle)_p$ for configuration c is executed. By the code of the DLC service, every process gets the same state for configuration c, that is $r' = r$ and $v' = v$. ∎

The following invariant states that when a configuration c is successful, any other configuration up to (and including) the next totally established configuration is for the same value as c.

Invariant 7.2.4 *(*DPAXOS*)*

In any reachable state the following is true. Let c be a successful configuration. Then for any configuration c' established at a process q, with $c.id < c'.id$ and such that there are no totally established configurations in between c and c', we have that $Hvalue(c'.id)_q = Hvalue(c.id)_{c.ldr} \neq \bot$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. In the initial state the invariant is vacuously true because there is no successful configuration.

For the inductive step assume that the invariant is true in a state s. We need to prove that the invariant is true in s' for any step (s, π, s'). Consider s' and fix c and c' as required by the statement in s'. Let $p = c.ldr$. By Invariant 7.2.2, in state s configuration c is established at p and at the accepting-quorum $Haccquo(c.id)_p = Q \in c.qrms$. So we have that $Q \subseteq s.state\text{-}dlv[c.id]$.

Now we distinguish two cases: (i) configuration c' is established at q in state s, (ii) configuration c' is not established at q in state s. In the former case the invariant follows by the inductive hypothesis. We need to consider the latter case. Since c' is not established at q in s and is established at q in s' it must be the case that $\pi =$NEWSTATE$(\langle r, v\rangle)_q$ for some $r \in \mathcal{G}$ and $v \in \mathcal{X}$. Configurations c and c' are not dead in s', as well as in s, because they are established in s'; moreover, by assumption, we have that in s' there is no totally established configuration between c and c'. Hence, by Invariant 7.1.1, there exists a quorum $Q' \in c.qrms$ such that $Q' \subseteq c'.set$. By the properties of quorums, there exists a process $q' \in Q \cap Q'$. For such a process we have that $q' \in c'.set$ and $q' \in s.state\text{-}dlv[c.id]$. Let s'' be the state in which process q' executes action SUBMIT-STATE$(\langle r', v'\rangle)$ that submits the state of q' to the condenser function ϕ_{state} for c'; since $q' \in s.state\text{-}dlv[c.id]$ and in state s'' we have that $current_{q'} = c'.id$, it must be the case that $q' \in s''.state\text{-}dlv[c.id]$ (because q' will not execute any other action for configuration c once its current configuration is c'). Hence the pair $\langle r', v'\rangle$ that q' submits to the condenser function ϕ_{state} for c' is such that $r' \geq c.id$. By the definition of ϕ_{state} we have that the pair $\langle r, v\rangle$ returned by action π is such that $r \geq c.id$. Hence we have that $s'.Hfrom(c'.id)_q \geq c.id$.

Let q'' be the process from which the ϕ_{state} function takes the pair $\langle r, v\rangle$ returned with action π; thus $v = s'.Hvalue(r)_{q''}$. By the code we have that $s'.Hfrom(c'.id)_q = r$ and that $s'.Hvalue(c'.id)_q = v$. Hence $s'.Hvalue(c'.id)_q = s'.Hvalue(r)_{q''}$.

If $r = c.id$ then we consider two cases.

Case (i): $r = g_0$. Then we claim that $q'' = p$. Indeed if the configuration with the biggest identifier among those submitted to the condenser function for c' is c_0, this means all members of c', when they submit their state to the condenser function, have not established any other configurations with identifier greater than g_0. This implies that all processes except p submit $\langle \perp, \cdot\rangle$ to the condenser function ϕ_{state} and p submits $\langle g_0, v_p\rangle$. Hence p is selected by the condenser function ϕ_{state}. Thus we have $s'.Hvalue(c'.id)_q = s'.Hvalue(c.id)_p$, as needed.

Case (ii): $r > g_0$. Obviously we have that $s'.Hvalue(c'.id)_q = s'.Hvalue(c.id)_{q''}$. By Invariant 7.2.3 we have that $s'.Hvalue(c.id)_{q''} = s'.Hvalue(c.id)_p$, and the invariant holds in this case.

It remains to consider the case $r > c.id$. By the inductive hypothesis applied to c, r and q'' we have that $s'.Hvalue(r)_{q''} = s'.Hvalue(c.id)_p$. Hence we conclude that $s'.Hvalue(c'.id)_q = s'.Hvalue(c.id)_p$, as needed. ∎

The following invariant is similar to the previous one, but considers totally established configurations instead of successful ones. It states that when a configuration c is totally established, any other configuration up to (and including) the next totally established configuration is such that its *Hvalue* is the same as that of c. First we give an auxiliary invariant.

Invariant 7.2.5 *(DPAXOS)*

In any reachable state the following is true. Let c be a totally established configuration such that $g_0 < c.id$. Then for any configuration c' established at a process q, with $c.id < c'.id$ and such that there are no totally established configurations in between c and c', we have that $Hvalue(c'.id)_q = Hvalue(c.id)_{c.ldr}$.

Proof: By induction on the length of the execution. The base case consists of proving that the invariant is true in the initial state. In the initial state the invariant is vacuously true because there is no established configuration c such that $g_0 < c.id$.

For the inductive step assume that the invariant is true in a state s. We need to prove that the invariant is true in s' for any step (s, π, s'). Consider state s' and let c and c' be as required by the statement in state s'. Let $p = c.ldr$. We distinguish four possible cases.

CASE 1: $c \in s.\mathcal{T}ot\mathcal{E}st$ and c' is established at q in s. Then we can apply the inductive hypothesis.

CASE 2: $c \in s.\mathcal{T}ot\mathcal{E}st$ and c' is not established at q in s. Then it must be the case that $\pi = \text{NEWSTATE}(\langle r, v\rangle)_q$. This action sets $Hvalue(c'.id)_q = v$.

Since c' is established in s' it is also not dead. Clearly also c is not dead in s'. By Invariant 7.1.1 we have that there is a quorum Q of c such that $Q \in c'.set$. Let $q' \in Q$. For such a process we have that $q' \in s'.state\text{-}dlv[c.id]$, $q' \in c'.set$.

Let s'' be the state in which process q' executes action $\text{SUBMIT-STATE}(\langle r', v'\rangle)$ that submits the state of q' to the condenser function ϕ_{state} for c'; since $q' \in s.state\text{-}dlv[c.id]$ and in state s'' we have that $current_q = c'.id$ it must be the case that $q' \in s''.state\text{-}dlv[c.id]$ (because q' will not execute any other action for configuration c once its current configuration is c'). Hence the pair $\langle r', v'\rangle$ that q' submits to the condenser function ϕ_{state} for c' is such that $r' \geq c.id$. By the definition of ϕ_{state} we have that the pair $\langle r, v\rangle$ returned by action π is such that $r \geq c.id$. Hence we have that $s'.Hfrom(c'.id)_q \geq c.id$. Let q'' be the process from which the ϕ_{state} function takes the pair $\langle r, v\rangle$ returned with action π; thus $v = s'.Hvalue(r)_{q''}$. By the code we have that $s'.Hfrom(c'.id)_q = r$ and that $s'.Hvalue(c'.id)_q = v$. Hence $s'.Hvalue(c'.id)_q = s'.Hvalue(r)_{q''}$.

If $r = c.id$ then, we have that $s'.Hvalue(c'.id)_q = s'.Hvalue(c.id)_{q''}$. By Invariant 7.2.3 we have that $s'.Hvalue(c.id)_{q''} = s'.Hvalue(c.id)_p$ and thus the invariant holds. So consider the case $r > c.id$. By the inductive hypothesis applied to c and r and q'' we have that $s'.Hvalue(r)_{q''} = s'.Hvalue(c.id)_p$. Hence we conclude that $s'.Hvalue(c'.id)_q = s'.Hvalue(c.id)_p$.

CASE 3: $c \notin s.\mathcal{T}ot\mathcal{E}st$, c' is established at q in s. Then it must be the case that $\pi = \text{NEWSTATE}(\langle r, v\rangle)_{p'}$ for some process p' that totally establishes configuration c. Configurations c and c' are not dead in s'. By Invariant 7.1.1 we have that there is a quorum Q of c such that $Q \in c'.set$. Let $q' \in Q$. For such a process we have that $q' \in s'.state\text{-}dlv[c.id]$, $q' \in c'.set$.

The proof proceeds as in the previous case: Let s'' be the state in which process q' executes action $\text{SUBMIT-STATE}(\langle r', v'\rangle)$ which submits the state of q' to the condenser function ϕ_{state} for c'; etc. (as done in the previous case).

CASE 4: $c \notin s.\mathcal{T}ot\mathcal{E}st$, c' not established at q in s. This is not possible because a single action cannot make both c totally established and c' established. ∎

The following invariant follows easily from the previous one.

Invariant 7.2.6 *(DPAXOS)*
In any reachable state the following is true. Let c be a totally established configuration such that $g_0 < c.id$. Then for any configuration c' established at $c'.ldr$, with $c.id < c'.id$, we have that $Hvalue(c'.id)_{c'.ldr} = Hvalue(c.id)_{c.ldr}$.

Proof: Let c and c' be as required by the statement. Let $c_1, c_2, ..., c_k$ be the sequence in order of configuration identifiers of the totally established configurations properly between c and c'. By Invariant 7.2.5 we have that $Hvalue(c_1.id)_{c_1.ldr} = Hvalue(c.id)_{c.ldr}$; by the same invariant we have $Hvalue(c_2.id)_{c_2.ldr} = Hvalue(c_1.id)_{c_1.ldr}$ and so on up to $Hvalue(c'.id)_{c'.ldr} = Hvalue(c_k.id)_{c_k.ldr}$. Thus we have that $Hvalue(c'.id)_{c'.ldr} = Hvalue(c.id)_{c.ldr}$. ∎

The following invariant is crucial to proving agreement.

Invariant 7.2.7 *(DPAXOS)*
In any reachable state the following is true. Let c be a successful configuration. Then for any configuration c' established at $c'.ldr$, with $c.id < c'.id$, we have that $Hvalue(c'.id)_{c'.ldr} = Hvalue(c.id)_{c.ldr}$.

Proof: If there are no totally established configurations between c and c' then the invariant follows directly from Invariant 7.2.4. So assume that there exists at least one totally established with identifier strictly greater than $c.id$ and strictly smaller than $c'.id$. Let c^* be the totally established configuration having the smallest identifier strictly greater than $c.id$ and let q be $c^*.ldr$. Clearly we have that c^* is established at q. By definition of c^* there are no totally established configurations between c and c^*. By Invariant 7.2.4 we have that $Hvalue(c^*.id)_q = Hvalue(c.id)_{c.ldr}$.
Since $c.id \geq g_0$, we have $c^*.id > g_0$. Hence by Invariant 7.2.6 we have that $Hvalue(c'.id)_{c'.ldr} = Hvalue(c^*.id)_q$. Hence $Hvalue(c'.id)_{c'.ldr} = Hvalue(c.id)_{c.ldr}$, as needed. ∎

We are now ready to prove agreement.

Theorem 7.2.8 *In any execution of the system* DPAXOS, *agreement is satisfied.*

Proof: In order to prove agreement we need to show that all the *decision* variables are set to the same value. By the code it is immediate that *decision* variables are always set to be equal to $Hvalue(c.id)_{c.ldr}$ for some successful configuration c. Hence it is enough to prove that any two successful configurations c and c' are such that $s.Hvalue(c.id)_{c.ldr} = s.Hvalue(c'.id)_{c'.ldr}$.

Let $p = c.ldr$ and $p' = c'.ldr$ and without loss of generality assume that $c.id < c'.id$. By Invariant 7.2.7 we have that $s.Hvalue(c.id)_p = s.Hvalue(c'.id)_{p'}$. ∎

Validity is easier to prove since the value proposed in any round comes either from an initial value or from a previous round.

Invariant 7.2.9 *(DPAXOS)*
In any reachable state of an execution α, for any round r such that $Hvalue(r)_{r.ldr} \neq \perp$, we have that $Hvalue(r)_{r.ldr} \in \mathcal{X}_\alpha$.

Proof: By induction on the length of the execution α. The base case consists of proving that the invariant is true in the initial state. In the initial state $Hvalue(r)_p$ is not $\perp$ only for $r = g_0$ and $p = r.ldr$. Moreover $Hvalue(g_0)_p$ is equal to the initial value of p. Hence the assertion is true.

For the inductive step assume that the invariant is true in a reachable state s. We need to prove that the invariant is still true in s' for any possible step (s, π, s').

Clearly the only actions that can make the assertion false are those that set $Hvalue(r)_p$ for some round r and $p = r.ldr$. The only action that sets $Hvalue(r)_p$ is action $\pi =$NEWSTATE$(\langle r', v\rangle)_p$ for round r. Action π sets $Hvalue(r)_p$ to v. We need to prove that $v \in \mathcal{X}_\alpha$. This follows from the definition of ϕ_{state} and the fact that the values submitted to ϕ_{state} are the *last-val*$_q$ variables which, in turn, are either the initial value v_q of q or the value $Hvalue(r')_{r'.ldr}$ of a previous round r'; by the inductive hypothesis we have that $Hvalue(r')_{r'.ldr}$ belongs to $\mathcal{X}_\alpha$. ∎

Theorem 7.2.10 *In any execution of the system* DPAXOS*, validity is satisfied.*

Proof: Let α be an execution of DPAXOS. A variable *decision* is always set to be equal to some $Hvalue(r)_p \neq \perp$ for some r and $p = r.ldr$ or to some other *decision* variable. By Invariant 7.2.9 we have that $Hvalue(r)_p$ belongs to $\mathcal{X}_\alpha$. Hence validity is satisfied ∎

Finally we claim, informally, that termination is satisfied. We remark that we are making the assumption that any failure in the system is detected by the group communication service which changes the configuration in order to reflect the new status of the underlying distributed system.

Consider an execution of the system such that there exists a state s in which a configuration c becomes totally established. Let t the point in time at which the system enters state s. Assume that there are no failures after time t. There is nothing that can prevent the round run in configuration c from success. Thus the leader of configuration c eventually writes its own *decision* variable. Once having done that, the leader keeps sending (see code) the value of its *decision* variable to any other process member of the configuration until it receives an acknowledgment. Since there are no failures every member of the configuration c will eventually receive the message from the leader and write the *decision* variable.

7.2.5 Remarks

We remark that the point-to-point communication mechanism of DLC is used by DLC-TO-PAXOS just to spread a reached decision to all members of the current configuration. Though it is fine to use configuration synchronous point-to-point messages, there is no need to require that messages used to spread the decision be configuration synchronous. A regular point-to-point channel which delivers messages regardless of the configurations in which the sender or the receiver are works fine too. Hence for the DPAXOS algorithm we could use a weaker version of the DLC service which provides point-to-point messages without configuration synchrony. We have used this stronger version because the algorithm that we present in the next section needs configuration synchronous point-to-point messages.

The original PAXOS algorithm [61] is designed to work with majorities or with more general quorums of a static universe of processes. Using quorums is good for handling transient failures of a system. However it does not work well for permanent failures. The usefulness of building PAXOS over the DLC group communication service is that it can adapt also to permanent failures by changing the configuration of the system.

It would be useful to compare the performance of the PAXOS algorithm built on top of DLC with that of the original PAXOS algorithm. Since our work has not addressed performance issues we leave this as future work.

The same technique that we have used to build PAXOS on top of DLC could be used to build MULTIPAXOS on top of DLC. The MULTIPAXOS algorithm [61][1] is basically a sequence of instances of the PAXOS algorithm that run together and optimize the number of messages needed in the first part of the round. The optimization is achieved by sending a unique message that works for all the instances of PAXOS. By using the DLC service such an optimization would be obtained by running multiple instance of PAXOS in the same configuration; the state exchange needs to be done only once for all the instances.

7.3 A Replicated Atomic Object Algorithm

In this section we develop a data replication algorithm that implements a replicated atomic object with arbitrary operations (not necessarily just read and write, though in practice these are the most common type of operations used). The algorithm, called RAB (Replicated Atomic oBject), is built upon the DLC service and uses a primary site to handle access to the object.

We start with an informal description of the algorithm, then we provide the formal code and finally we provide key arguments for its correctness. Providing a formal proof of correctness is left

[1] The name MULTIPAXOS is actually used in [29]. The original paper by Lamport [61] uses a different name (multi-decree parliament protocol).

as future work.

7.3.1 Description of RAB

Operations are centralized at the leader of the current configuration; the leader requires the collaboration of at least a quorum of processes in order to handle requests.

Clients of the service request to perform operations on the data. Each process accepts requests from its client and places them in a local order. Then each of the received requests is sent to the leader who is responsible for building a global order for all the client's requests. For each of these requests the leader makes sure that at least a quorum of processes know the request before providing an answer to the process that originates the request. Once such an answer is provided to the originator process, a response can be given back to the client.

When a configuration change happens all the members submit their knowledge about the requests performed so far and a new common state is computed from the local knowledge of the processes. In particular, each process submits its own information about the global order of operations plus all the local requests that are still pending, that is, have been submitted to the leader but have not received a response. The global orders submitted by each member of the new configuration are used to compute the most up to date global order, while the information about pending requests is used to locate those operations that must be resubmitted to the leader.

7.3.2 The code of DLC-TO-RAB

In this section we provide the code of algorithm DLC-TO-RAB. We first define some data types. We denote by $\mathcal{X}$ the set of values that the shared data can assume and by $v_0 \in \mathcal{X}$ a predefined value. The set $\mathcal{T}$ is a set of types of operations (e.g., read, write operations).

The set $\mathcal{D}$ of "operation descriptors" is defined as $\mathcal{D} = \{\langle p, t, w, i\rangle | p \in \mathcal{P}, t \in \mathcal{T}, w \in \mathcal{X}, i \in \mathbf{N}^{>0}\}$. Operation descriptors are used to describe both the requests from the clients and the corresponding responses. For an element $y = \langle p, t, w, i\rangle$ of $\mathcal{D}$ we use the following selectors to extract the single components: $y.origin = p$, $y.type = t$, $y.param = w$ and $y.local\text{-}rank = i$. Component $y.origin$ records the client at which the request has originated. Component $y.type$ specifies the type of operation. For example, if we want a read-write register, types could be $\mathcal{T} = \{\text{"}read\text{"}, \text{"}write\text{"}\}$. Component $y.param$ provides possible parameters that need to be passed along with the request or with the corresponding response. Considering again the case of a read-write register, a write needs to pass the value to be written and the response to a read needs to pass the value read. For simplicity we assume that only one value needs to be passed and this value is an element of $\mathcal{X}$ (this is so in the case of a read-write register).

The set of messages that can be sent over the point to point channels and through the DLC service is defined as $\mathcal{M} = \mathcal{D} \cup (\{\text{"}req\text{"}, \text{"}ans\text{"}\} \times \mathcal{D})$. The set of operation identifiers is $OID = \mathbf{N}^{>0} \times \mathcal{P}$.

The set $\mathcal{S}$ is defined as $\mathcal{S} = \{\langle o, d, a, h\rangle | o \in \mathit{arrayof}(\mathcal{D}), d \in \mathit{arrayof}(\mathit{bool}), a \in \mathit{arrayof}(\mathit{bool}), h \in \mathcal{G}\}$. We remind the reader that the notation $\mathit{arrayof}(\mathcal{D})$ indicates an array whose elements are either elements of $\mathcal{D}$ or $\perp$. The set of acknowledgment values is $\mathcal{A} = \{\text{“}\mathit{ack}\text{”}\} \cup \mathcal{S}$. The set of response values is $\mathcal{R} = \{\text{“}\mathit{done}\text{”}\} \cup \{\langle o, d, a\rangle | o \in \mathit{arrayof}(\mathcal{D}), d \in \mathit{arrayof}(\mathit{bool}), a \in \mathit{arrayof}(\mathit{bool})\}$.

The DLC-TO-RAB algorithm uses two condenser functions that we define in the following:

- ϕ_{done}: This condenser function takes a set of acknowledgment values “*ack*” and returns the string “*done*”. This function is used by the leader to make sure that a quorum of processes have received information about a particular operation.

- $\phi_{rabstate}$: Let c be the configuration for which this condenser function is to be used. The condenser function takes a collection $S \subseteq \mathcal{S}$ of tuples, one for each member of c, and returns a triple $\langle o, d, a\rangle \in \mathcal{R}$, defined as follows:

  - o is defined as follows: Let R be the set of tuples of S that have the maximum *high* component. For any $i \in \mathbf{N}^{>0}$ such that there exists at least one element $x \in R$ with $x.\mathit{order}(i) \neq \perp$, fix any such element x and set $o(i) := x.\mathit{order}(i)$. For any i for which no such element exists set $o(i) := \perp$.
  - d is defined as the “or” of the *req-done* components in R.
  - a is defined as the “or” of the *req-answrd* components in R.

The code of DLC-TO-RAB is provided in Figure 7-3; we describe it next. We start with the description of the state variables.

Variable $\mathit{current}_p$ contains the current configuration of process p and variable high_p contains the latest established configuration of process p. Variable $\mathit{local\text{-}req}_p$ is the sequence of requests that the client submits at process p; variable $\mathit{local\text{-}ans}_p$ contains the answers for all of the requests. Variable next_p is a pointer used to insert new requests from the client into $\mathit{local\text{-}req}_p$. Variable order_p contains the sequence of all requests as known by process p. Variable status_p contains the status of process p; it is used when a new configuration is announced; for regular computation this variable is set to *normal*.

The remaining state variables are flags used to record that some particular actions have happened. Variable $\mathit{req\text{-}sent}(j)_p$ is set to `true` when the j^{th} request of the client at p, that is $\mathit{local\text{-}req}(j)_p$, is sent to the leader of the current configuration. Such a request, when received by the leader is placed in the global sequence of request *order* into some available position, say i. Variable $\mathit{req\text{-}sbmttd}(i)_p$ is set to `true` when the leader p has submitted the request via the DLC service with action SUBMIT$(m, \phi_{done}, \langle i, p\rangle)$. Variable $\mathit{req\text{-}acked}(i)_p$ is set to `true` when process p has sent an acknowledgment for the i^{th} request. Variable $\mathit{req\text{-}done}(i)_p$ is set to `true` when the leader p receives the response from the DLC service for the i^{th} request with action RESPOND(“*done*”$, \langle i, p\rangle)$. Variable

DLC-TO-RAB

Signature:

Input: $\text{READ}(desc, param)_p$, $desc \in \mathcal{D}, param \in \mathcal{X}, p \in \mathcal{P}$
$\text{P2P-RECV}(m)_p$, $m \in \mathcal{M}$, $p \in \mathcal{P}$
$\text{DELIVER}(m, i)_p$, $m \in \mathcal{M}$, $i \in OID$, $p \in \mathcal{P}$
$\text{RESPOND}(a, i)_p$, $a \in \mathcal{A}$, $i \in OID_p$, $p \in \mathcal{P}$
$\text{NEWCONF}(c)_p$, $c \in \mathcal{C}$, $p \in c.set$
$\text{NEWSTATE}(s)_p$, $s \in \mathcal{S}$, $p \in \mathcal{P}$

Output: $\text{CONFIRM}(param)_p$, $param \in \mathcal{X}$, $p \in \mathcal{P}$
$\text{P2P-SEND}(m)_p$, $m \in \mathcal{M}$, $p \in \mathcal{P}$
$\text{SUBMIT}(m, \phi, i)_p$, $m \in \mathcal{M}$, $\phi \in \Phi, p \in \mathcal{P}$, $i \in OID_p$
$\text{ACKDLVR}(a, i)_p$, $a \in \mathcal{A}$, $i \in OID$, $p \in \mathcal{P}$
$\text{SUBMIT-STATE}(s, \phi)_p$, $s \in \mathcal{S}$, $\phi \in \Phi$, $p \in \mathcal{P}$

State:

$current \in \mathcal{C}_\perp$ initially c_o if $p \in P_0$, else $\perp$
$high \in \mathcal{C}_\perp$ initially $\perp$
$local\text{-}req \in arrayof(\mathcal{D})$, initially $\perp$ everywhere
$local\text{-}ans \in arrayof(\mathcal{D})$, initially $\perp$ everywhere
$next \in \mathbf{N}$, initially 1
$order \in arrayof(\mathcal{D})$, initially $\perp$ everywhere
$status \in \{normal, exch\text{-}ready, exch\text{-}wait\}$, initially $normal$

$req\text{-}sent \in seqof(bool)$, initially `false` everywhere
$req\text{-}sbmttd \in seqof(bool)$, initially `false` everywhere
$req\text{-}acked \in seqof(bool)$, initially `false` everywhere
$req\text{-}done \in seqof(bool)$, initially `false` everywhere
$req\text{-}answrd \in seqof(bool)$, initially `false` everywhere
$req\text{-}cnfrmd \in seqof(bool)$, initially `false` everywhere

Derived variable:

$apply\text{-}all(i)_p$ is defined as follows:
if for all $k \leq i$, $order(k)_p \neq \perp$ then
  $apply\text{-}all(i)_p = \langle q, t, w, j\rangle$, where $q = order(i)_p.origin$, $t = order(i)_p.type$,
  w is the value obtained by applying operations $order(1, .., i)_p$ to the initial value v_0 in order,
  and $j = order(i)_p.local\text{-}rank$;
else
  $apply\text{-}all(i)_p = \perp$.

Actions:

input $\text{REQUEST}(type, param)_p$
 Eff: $local\text{-}req(next) := \langle p, type, param, next\rangle$
  $next := next + 1$

output $\text{CONFIRM}(param)_p$
 Pre: $local\text{-}ans(i) = \langle p, type, param, i\rangle$
  $\forall j \leq i$, $req\text{-}cnfrmd(j) = \texttt{true}$
  $req\text{-}cnfrmd(i) = \texttt{false}$
 Eff: $req\text{-}cnfrmd(i) := \texttt{true}$

output $\text{P2P-SEND}(\langle \text{“}req\text{”}, m\rangle)_{p,q}$ choose j
 Pre: $current \neq \perp$
  $q = current.ldr$
  $m = local\text{-}req(j)$
  $m.local\text{-}rank = j$
  $req\text{-}sent(j) = \texttt{false}$
  $status = normal$
 Eff: $req\text{-}sent(j) := \texttt{true}$

input $\text{P2P-RECV}(\langle \text{“}req\text{”}, m\rangle)_{q,p}$
 Eff: Let i be such that
   $\forall k < i$, $order(k) \neq \perp$
   $order(i) = \perp$
  $order(i) := m$

output $\text{SUBMIT}(m, \phi_{done}, \langle i, p\rangle)_p$
 Pre: $current \neq \perp$
  $p = current.ldr$
  $order(i) = m$
  $m \neq \perp$
  $req\text{-}sbmttd(i) = \texttt{false}$
  $status = normal$
 Eff: $req\text{-}sbmttd(i) := \texttt{true}$

input $\text{DELIVER}(m, \langle i, r\rangle)_p$
 Eff: $order(i) := m$
  $req\text{-}acked(i) := \texttt{false}$

output $\text{ACKDLVR}(\text{“}ack\text{”}, \langle i, r\rangle)_p$
 Pre: $r = current.ldr$
  $\forall j \leq i$, $order(j) \neq \perp$
  $req\text{-}acked(i) = \texttt{false}$
  $status = normal$
 Eff: $req\text{-}acked(i) := \texttt{true}$

input $\text{RESPOND}(\text{“}done\text{”}, \langle i, p\rangle)_p$
 Eff: $req\text{-}done(i) := \texttt{true}$

output $\text{P2P-SEND}(\langle \text{“}ans\text{”}, a\rangle)_{p,q}$ choose i
 Pre: $p = current.ldr$
  $order(i) = \langle m, j\rangle$ for some m
  $req\text{-}answrd(i) = \texttt{false}$
  $req\text{-}done(i) = \texttt{true}$
   $\forall k \leq i$, $order(k) \neq \perp$
   $\forall k < i$, $req\text{-}answrd(k) = \texttt{true}$
  $q = order(i).origin$
  $a = apply\text{-}all(i)$
  $status = normal$
 Eff: $req\text{-}answrd(i) := \texttt{true}$

input $\text{P2P-RECV}(\langle \text{“}ans\text{”}, a\rangle)_{q,p}$
 Eff: $j := a.local\text{-}rank$
  $local\text{-}ans(j) := a$

Figure 7-3: The DLC-TO-RAB code.

DLC-TO-RAB

input NEWCONF$(c)_p$
Eff: $current := c$
$status := exch\text{-}ready$
$\forall i,\ order(i) \neq \bot$ and $req\text{-}acked(i) = \texttt{false}$
  $req\text{-}acked(i) := \texttt{true}$

output SUBMIT-STATE$(\langle o, d, a, g\rangle, \phi_{rabstate})_p$
Pre: $status = exch\text{-}ready$
$o = order$
$d = req\text{-}done$
$a = req\text{-}answrd$
$g = prev.id$
Eff: $status := exch\text{-}wait$

input NEWSTATE$(\langle o, d, a\rangle)_p$
Eff: $status := normal$
$high := current$
$order := o$
$req\text{-}done := d$
$req\text{-}answrd := a$
$\forall j,\ local\text{-}req(j) \neq \bot$
  if $\nexists i$ such that
    $order(i).origin = p$ and
    $order(i).local\text{-}rank = j$ then
      $req\text{-}sent(j) := \texttt{false}$
$\forall i,\ order(i) \neq \bot$
  if $req\text{-}done = \texttt{false}$ then
    $req\text{-}submttd(i) := \texttt{false}$
  else
    $req\text{-}submttd(i) := \texttt{true}$

Figure 7-4: The DLC-TO-RAB code (cont'd).

$req\text{-}answrd(i)_p$ is set to `true` when the leader p has sent an answer for the i^{th} request to the originator process of that request. Variable $req\text{-}cnfrmd(j)_p$ is set to `true` when process p has given the client a response for the j^{th} request submitted at p.

Next we describe the transitions.

Action REQUEST$(type, param)_p$ records a new request from the client at process p in the sequence of local requests $local\text{-}req_p$. Pointer $next_p$ always points to the first available location in the sequence $local\text{-}req_p$.

Action CONFIRM$(param)_p$ provides the response to the requests of the client. Such responses are given in the same order as they are received. This is accomplished by using variable $req\text{-}cnfrmd_p$; the response to the j^{th} (local) request is given back to the client after all responses for the previous requests have been provided, and, of course, when the response is available, that is, when $local\text{-}ans(j)_p$ has been set.

Action P2P-SEND$(\langle \text{“}req\text{”}, m\rangle)_{p,q}$ is used by process p to send the request m to the leader. Such a request is received by the leader with action P2P-RECV$(\langle \text{“}req\text{”}, m\rangle)_{q,p}$. The leader inserts request m into the global order of requests $order_p$ in the next available position.

Once the leader p has placed a request in the i^{th} position of $order_p$, it executes action SUBMIT$(m, \phi_{done}, \langle i, p\rangle)_p$, where $m = order(i)_p$. The leader needs to make sure that at least a quorum of processes learn about the request.

A request m submitted by process q is delivered to process p by the DLC service by means of action DELIVER$(m, \langle i, r\rangle)_p$. Upon receiving such a request process p simply updates its own *order* by placing the request m into $order(i)_p$. We remark that the code allows for overwriting a previous value; however it is never the case that a process p overwrites an old value of $order(i)$ with something different received with action DELIVER$(m, \langle i, r\rangle)_p$. Flag $req\text{-}acked(i)_p$ is set to `false` so that action ACKDLVR will send an acknowledgment.

Action $\text{ACKDLVR}(\text{“}ack\text{”}, \langle i, r\rangle)_p$ sends back to the DLC service an acknowledgment for the i^{th} operation and sets $\textit{req-acked}(i)_p$ to `true`.

Once a quorum of processes has sent acknowledgments for a particular request, the DLC service notifies the leader with action $\text{RESPOND}(\text{“}done\text{”}, \langle i, p\rangle)_p$. The leader simply sets the flag $\textit{req-done}(i)$ to `true` to record the fact that now request i is known by a quorum of processes.

Once all the requests up to the i^{th} one are known to a quorum of the processes, the leader can send an answer to the originator of the request. This is done in action $\text{P2P-SEND}(\langle \text{“}ans\text{”}, val\rangle)_{p,q}$. The code of this action uses the derived variable $\textit{apply-all}(i)$ which applies all operations up to the i^{th} and returns a tuple $a \in \mathcal{D}$ that contains the response for operation i ($a.param$ is the value of the shared data after the i^{th} operation). We remark that, of course, a real implementation will only keep the current value; it would apply operations in order and provide an answer to operation i right after applying it and before applying operation $i + 1$.

When process p receives the answer for a request previously sent to the leader it just records the answer into $\textit{local-ans}_p$. This is done in action $\text{P2P-RECV}(\langle \text{“}ans\text{”}, a\rangle)_{q,p}$.

Finally we describe the actions used for the state exchange. When a new configuration is announced with action $\text{NEWCONF}(c)_p$ process p sets its current configuration to c and goes into reconfiguration mode by setting $status_p$ to *exch-ready*. It also sets $\textit{req-acked}_p$ to `true` for all those operations that have pending acknowledgments to be sent; since the old configuration has been left, such an acknowledgment must not be sent anymore and setting the flag $\textit{req-acked}_p$ to `true` has this effect. Indeed in the new configuration process p has not yet received any message from the leader so it is incorrect to acknowledge a message.

Then process p submits to the DLC service its $order_p$, $\textit{req-done}_p$, $\textit{req-answrd}_p$ and $high_p$, which constitute the relevant part of the state that has to be exchanged.

When all the processes have submitted their states, the DLC service is able to compute the starting state of the new configuration by using the $\phi_{rabstate}$ condenser function. Then it gives this state to process p by means of action $\text{NEWSTATE}(\langle o, d, a\rangle)_p$. When this action is executed, process p updates its $order_p$, $\textit{req-done}_p$ and $\textit{req-answrd}_p$ state components. It also adjusts the values of $\textit{req-sent}_p$ and $\textit{req-submitted}_p$ to take care of two problems that arise in establishing a new configuration, as we explain below.

The first problem is that any process p has to check whether all of its local requests are in the global order o returned by $\text{NEWSTATE}(\langle o, d, a\rangle)_p$; for any local request not included in the order o, process p has to send that local request to the leader because the leader of the new configuration does not know about such a request. This is done by setting $\textit{req-sent}(j)$ to `false` for those local operations that the leader does not know about.

The other problem regards operations that are included in the order o returned by $\text{NEWSTATE}(\langle o, d, a\rangle)_p$, but for which *req-done* is still `false`. For such operations the leader cannot be sure that a quorum of processes have them in their global order and thus cannot provide an answer for such operations. The leader needs to resubmit such operations to the DLC service in order to make sure that a quorum of processes learn about them, before giving an answer to the originator of the request.

A simple scenario that illustrates this problem is the following. Assume that quorums are just majorities, the initial configuration has membership $\{p_1, p_2, p_3, p_4, p_5, p_6\}$, the leader is p_6 and the identifier of the configuration is 1. Process p_6 receives a request op_1 from its client, puts it into its local requests list and sends it to the leader (which in this case is itself). Then process p_6, upon receiving its own request, puts op_1 into the first position of the global order. Process p_6 submits the request to the DLC service, but before any other process gets its message, a configuration change happens. A new process p_7 joins the system and configuration 2, whose membership set is $\{p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$, is created. The leader of this configuration is p_7. Every member submits its state. Only p_6 has something in its global order (namely op_1 in position 1) and thus the new global order computed by $\phi_{rabstate}$ for configuration 2 just contains op_1 in position 1. Moreover we have that *req-done* for this operation is `false` because the previous leader did not succeed in having a quorum learn about such a request. Assume that the global order for configuration 2 is delivered only to processes p_6 and p_7, but not to the other members. The leader p_7 cannot yet give back a response to p_6 for the request op_1; indeed if the leader does so it allows process p_6 to give an answer back to its client and an inconsistency may arise as we show next. Assume that configuration 3 is established. This new configuration has membership set $\{p_1, p_2, p_3, p_4, p_5\}$ and leader p_1. No one in configuration 3 knows that op_1 has even been submitted. The global order conveyed by action NEWSTATE(o, d, a) is empty. A new operation op_2 comes in, say from process p_2, the leader p_1 receives it, puts it into the first position of the global order, submits it through the DLC service and receives acknowledgments from a quorum and gives an answer back to process p_2. We have an inconsistency: process p_2 told its client that op_2 is the first operation applied to the shared object while process p_6 told its client that operation op_1 is the first operation applied to the shared object. Hence, before giving a response, the leader of a new configuration needs to submit to the DLC service operations that have not successfully gone to a quorum through the DLC service.

Thus for any i such that $order(i)_p \neq \bot$ and $req\text{-}done(i) =$ `false`, that is, for any operation in *order* for which the leader cannot be sure that a quorum of processes know about that operation, the flag $req\text{-}submitted(i)$ is set to `false` so that the operation will be submitted to the DLC service.

Another way to get around the problem mentioned above, is to delay the response for an operation i, for which the leader does not know that the operation has been spread to a quorum, until a later operation j, $j > i$, is spread to a quorum in the same configuration. When operation j is known to a quorum Q, we have that also operation i is known to a quorum because each of the processes in Q has to establish the configuration and thus knows operation i. The RAB algorithm adopts the solution of submitting operation i to the DLC service to make sure a quorum knows it.

7.3.3 Sketch of proof of correctness

In this section we present the key arguments for a correctness proof for the RAB algorithm.

The overall algorithm RAB consists of the composition of DLC and automaton DLC-TO-RAB$_p$ for each $p \in \mathcal{P}$. We claim that the RAB algorithm implements an atomic shared object. An atomic shared object is an object that can be accessed concurrently by several processes that issue invocations (requests) and receive responses for those requests in such a way that it is possible to insert serialization points that make the responses consistent with all previous (with respect to the serialization points) events. In the RAB algorithm invocation events are the actions REQUEST$_p$ and response events are the actions CONFIRM$_p$. We refer the reader to Chapter 13 of [65] for a formal definition of atomic object.

We define the history variable $\textit{build-order}(g,i)_p \in \mathcal{D}_\perp$ for each process $p \in \mathcal{P}$, each configuration identifier $g \in \mathcal{G}$ and $i \in \mathbf{N}^{>0}$. Such a variable is $\perp$ if process p has not established g, otherwise is defined as follows: if $\textit{current.id}_p > g$ then $\textit{build-order}(g,i)_p$ is equal to the value $\textit{order}(i)_p$ when p left configuration g; if $\textit{current.id}_p = g$ then $\textit{build-order}(g,i)_p = \textit{order}(i)_p$.

Next we provide key invariants that will be used to prove that RAB implements an atomic object. Remember that variable $\textit{state-dlv}[c.id]$ contains the set of processes that have established configuration c.

Invariant 7.3.1 *In any reachable state the following is true. Let c be a configuration such that $c.ldr \notin \textit{state-dlv}[c.id]$. Then for any $p, q \in \textit{state-dlv}[c.id]$ and any $i \in \mathbf{N}^{>0}$, we have that $\textit{build-order}(c.id,i)_p = \textit{build-order}(c.id,i)_q$.*

Proof: When a configuration is established at a member p, process p executes action NEWSTATE$(\langle o,d,a\rangle)_p$ and sets $\textit{order}_p := o$. The tuple $\langle o,d,a\rangle$ is the same for every member of the configuration. So initially every member has the same value of $\textit{order}$. Within the configuration a member p updates $\textit{order}(i)_p$ to a particular value m only when the leader r executes action DELIVER$(m,\langle i,r\rangle)_p$; but since the leader has not established c, such an action cannot be executed. □

Invariant 7.3.2 *In any reachable state the following is true. Let c be a configuration such that $c.ldr \in \textit{state-dlv}[c.id]$. Then for any $p \in \textit{state-dlv}[c.id]$ and any $i \in \mathbf{N}^{>0}$, we have that if $\textit{build-order}(c.id,i)_p \neq \perp$ then $\textit{build-order}(c.id,i)_p = \textit{build-order}(c.id,i)_{c.ldr}$.*

Proof: When a configuration is established at a member p, process p executes action NEWSTATE$(\langle o,d,a\rangle)_p$ and sets $\textit{order}_p := o$. The tuple $\langle o,d,a\rangle$ is the same for every p member of the configuration. So initially every member has the same value of $\textit{order}_p$. Within the configuration a member p updates $\textit{order}(i)_p$ to a particular value m only when executing action DELIVER$(m,\langle i,r\rangle)_p$; but in this case we have that the leader of the configuration is r and $\textit{order}(i)_r = m$. □

We remark that the knowledge of *order* may diverge for those processes that remain in obsolete configurations. For example if a process p updates $order(i)_p$ to m because it receives such an m from the leader of a configuration c_1, but a new configuration c_2 is established before any other process updates $order(i)$, then the only two processes to have $order(i) = m$ might be p and $c_1.ldr$. Assume that neither p or $c_1.ldr$ is a member of c_2. Then the leader of c_2 can write something different from m into $order(i)$. The above scenario is possible because m is not known to enough processes.

Given an index i, an $m \in \mathcal{D}$ and a configuration c we say that the triple (i, m, c) is *good* in a state s if there exists a quorum $Q \in c.qrms$ such that for every process $p \in Q$ we have $s.build\text{-}order(c.id, i)_p = m$.

We also say that (i, m) is good if there exists a configuration c such that (i, m, c) is good and that an index i is good if there exists m such that (i, m) is good.

The definition of a good index admits the possibility that in a given state there exist m and m' such that (i, m) and (i, m') are both good; however, as we will see later, this never happens in the algorithm. Indeed the notion of good index is intended to capture the fact that an operation m has been assigned to the i^{th} position of *order*. In the following we will prove that operations assigned to good indexes are propagated to newer configurations.

Invariant 7.3.3 *In any reachable state the following is true. Suppose that $w \in \mathcal{E}st$ and $c \in \mathcal{E}st$ such that $w.id < c.id$, and there are no totally established configurations x with $w.id < x.id < c.id$. Suppose (i, m, w) is good. Then $order(i)_p = m$ for every $p \in state\text{-}dlv[c.id]$.*

Proof: We prove the invariant by induction on the length of the execution. The base case is vacuously satisfied.

For the inductive step, let s be a reachable state and assume that the invariant is true in all states previous to s. We need to prove that the invariant is true in s.

Let w and c be configurations satisfying the assumption of the statement. Let (i, m, w) be good in s. Since (i, m, w) is good in s, there exists a quorum $Q \in w.qrms$ of processes such that for each $r \in Q$ we have $s.build\text{-}order(w.id, i)_r = m$. For the rest of the proof we fix such a Q.

We need to prove that any process $p \in s.state\text{-}dlv[c.id]$ has $s.order(i)_p = m$. Processes that establish c set their *order* variables to the value computed by the condenser function $\phi_{rabstate}$ for configuration c. Hence we need to look at the inputs that the condenser function $\phi_{rabstate}$ for configuration c receives from the members of c.

Since c is established, all members of c submit their state to the condenser function $\phi_{rabstate}$ for configuration c. Partition $c.set$ into three subsets S_1, S_2 and S_3, as follows: S_1 contains the processes that had $high < w.id$ at the moment they submitted the state to $\phi_{rabstate}$ for configuration c; S_2 contains the processes that had $high = w.id$ at the moment they submitted the state to $\phi_{rabstate}$ for configuration c; S_3 contains the processes that had $high > w.id$ at the moment they submitted the state to $\phi_{rabstate}$ for configuration c.

In the following we provide three claims that will be used to complete the proof.

CLAIM 1: $S_2 \cup S_3 \neq \emptyset$.

PROOF OF CLAIM 1: By Invariant 7.1.1, a quorum $Q' \in w.qrms$ is included in $c.set$. Let r be a process in $Q \cap Q'$. Such a process exists because Q and Q' are quorums of w. Clearly $r \in c.set$. Since process $r \in Q$ we have that $s.build\text{-}order(w.id, i)_r = m$. By definition of *build-order* we have that process r has established w in state s. Process r has to establish w before submitting its state for c, because it does not take any action for w after participating in c. Hence at the moment r submits its state for c we have that $high_r \geq w.id$. Therefore $r \in S_2 \cup S_3$. Thus $S_2 \cup S_3 \neq \emptyset$.

CLAIM 2: If $q \in S_2 \cup S_3$ then q submits either m or $\perp$ as $order(i)$ to the condenser function $\phi_{rabstate}$ for configuration c.

PROOF OF CLAIM 2: Fix any $q \in S_2 \cup S_3$. Let s' be the state in which process q submits its state for the condenser function $\phi_{rabstate}$ for configuration c. We need to prove that $s'.build\text{-}order(w'.id, i)_q = m$, where w' is the configuration that q establishes before joining c. (Configuration w' is equal to w for $q \in S_2$ and is a later one for $q \in S_3$.)

We consider two cases: $q \in S_2$ and $q \in S_3$.

CASE 1: $q \in S_2$. In this case $w' = w$ so we need to prove that $s'.build\text{-}order(w.id, i)_q$ is either m or $\perp$.

We first notice that state s' precedes state s.

We consider two cases: (i) $q \in Q$, and (ii) $q \notin Q$.

CASE 1.1: $q \in Q$. Since in s' process q already participates in c and $c.id > w.id$ we have that after s' process q does not execute any action for configuration w and thus $build\text{-}order(w.id, i)_q$ does not change after s'. Since $q \in Q$, we have that $s.build\text{-}order(w.id, i)_q = m$. Since $build\text{-}order(w.id, i)_q$ does not change after s', we must have $s'.build\text{-}order(w.id, i)_q = m$.

CASE 1.2: $q \notin Q$. We first notice that since $q \in S_2$ we have that that $q \in s'.state\text{-}dlv[w.id]$. This implies that $q \in s.state\text{-}dlv[w.id]$.

If $s.build\text{-}order(w.id, i)_q = \perp$, since process q has already left configuration w by state s', we have that $s'.build\text{-}order(w.id, i)_q = \perp$, as needed. Hence assume that $s.build\text{-}order(w.id, i)_q \neq \perp$.

By Invariant 7.1.1, a quorum $Q'' \in w.qrms$ is included in $c.set$. Since Q and Q'' are quorums of w, there exists a process $r \in Q \cap Q''$. Clearly $r \in c.set$. Since $r \in Q$ we have that $s.build\text{-}order(w.id, i)_r = m$.

Next we prove that $s.\mathit{build\text{-}order}(w.id, i)_q = m$.

If w is not established at $w.ldr$ in s, by Invariant 7.3.1 we have that $s.\mathit{build\text{-}order}(w.id, i)_q = s.\mathit{build\text{-}order}(w.id, i)_r = m$.

If w is established at $w.ldr$ in s, by Invariant 7.3.2 we have that $s.\mathit{build\text{-}order}(w.id, i)_{w.ldr} = s.\mathit{build\text{-}order}(w.id, i)_r = m$. By the same invariant, since $s.\mathit{build\text{-}order}(w.id, i)_q \neq \bot$, we have that $s.\mathit{build\text{-}order}(w.id, i)_q = s.\mathit{build\text{-}order}(w.id, i)_{w.ldr} = m$; hence also in this case $s.\mathit{build\text{-}order}(w.id, i)_q = m$.

Thus we have that $s.\mathit{build\text{-}order}(w.id, i)_q = m$.

Since in state s' process q has already left configuration w, we have that after s' process q does not change $\mathit{build\text{-}order}(w.id, i)_q$. Since $s.\mathit{build\text{-}order}(w.id, i)_q = m$ we have that $s'.\mathit{build\text{-}order}(w.id, i)_q = m$, as needed.

Case 2: $q \in S_3$. This process arrives in configuration c from a configuration w' such that $w.id < w'.id < c.id$. We need to prove that $s'.\mathit{build\text{-}order}(w'.id, i)_q = m$.

By the inductive hypothesis we have that the statement is true in s'. By applying the inductive hypothesis to state s' with $c = w'$ we have that $s'.order(i)_q = m$. By definition of *build-order* we have that $s'.\mathit{build\text{-}order}(w'.id, i)_q = m$, as needed.

Claim 3: At least one process in $S_2 \cup S_3$ submits m as $order(i)$ to the condenser function $\phi_{rabstate}$ for configuration c.

Proof of Claim 3: By Invariant 7.1.1, a quorum $Q''' \in w.qrms$ is included in $c.set$. Since Q and Q''' are quorums of w, there exists a process $r \in Q \cap Q'''$. Clearly $r \in c.set$ and also $r \in S_2 \cup S_3$. Since $r \in Q$ we have that $s.\mathit{build\text{-}order}(w.id, i)_r = m$.

If r arrives to configuration c directly from w, then since $s.\mathit{build\text{-}order}(w.id, i)_r = m$, process r submits m to the condenser function $\phi_{rabstate}$ for configuration c.

Thus consider the case when r arrives to configuration c from w', $w.id < w'.id < c.id$. Let s' be the state in which process r submits its state to the $\phi_{rabstate}$ function for configuration c. By applying the inductive hypothesis to state s' with $c = w'$ we have that $s'.order(i)_q = m$. By definition of *build-order* we have that $s'.\mathit{build\text{-}order}(w'.id, i)_q = m$. Hence, also in this case, process r submits m to the condenser function $\phi_{rabstate}$ for configuration c.

We are now ready to conclude the proof.

By Claim 1, we have that $S_2 \cup S_3$ contains at least one process. Thus, by definition of the condenser function $\phi_{rabstate}$, we have that the states of processes in S_1 are ignored by $\phi_{rabstate}$. So we only need to worry about what processes in $S_2 \cup S_3$ submit to the state condenser function for configuration c. By Claim 2, we have that processes in S_2 and S_3 submit either m or $\perp$ as the i^{th} entry of *order* to the state condenser function for configuration c. By Claim 3, at least one process in S_2 and S_3 submits m.

Hence $\phi_{rabstate}$ computes an order o for configuration c such that $o(i) = m$.

Every process p that establishes configuration c sets $order_p := o$ when executing action $\text{NEWSTATE}(\langle o, d, a \rangle)_p$ for configuration c. Thus, for such a process, we have that $order(i)_p = m$ when it establishes configuration c. Within configuration c, process p modifies $order_p$ only when receiving a message from the leader. However the leader also has $order(i)_{c.ldr} = m$. So $order_p$ does not change.

Hence we conclude that if $p \in s.state\text{-}dlv[c.id]$, then $s.order(i)_p = m$. ∎

The next invariant is similar to the previous one, but removes the requirement that there be no totally established configurations between w and c.

Invariant 7.3.4 *In any reachable state the following is true. Let $w \in \mathcal{E}st$ and $c \in \mathcal{E}st$ such that $w.id < c.id$. Let (i, m, w) be good. Then $order(i)_p = m$ for every $p \in state\text{-}dlv[c.id]$.*

Proof: Let s be any reachable state and let w and c be as required by the statement. Let (i, m, w) be good in s. Let $p \in state\text{-}dlv[c.id]$. We need to prove that $s.order(i)_p = m$

Let $x_1, x_2, ...x_k$ be the sequence, in order of configuration identifier, of totally established configurations between w and c. Since (i, m, w) is good, by Invariant 7.3.3 we have that $s.order(i)_q = m$ for any process q such that $q \in state\text{-}dlv[x_1.id]$. Configuration x_1 is totally established, hence for any $q \in x_1.set$ we have $s.order(i)_q = m$. Hence we have $s.build\text{-}order(x_1.id, i)_q = m$ for each member of x_1.

It follows that (i, m, x_1) is good. Thus by Invariant 7.3.3, used with $w = x_1$, we have that $s.order(i)_q = m$ for any process q such that $q \in state\text{-}dlv[x_2.id]$. Configuration x_2 is totally established, hence for any $q \in x_2.set$ we have $s.order(i)_q = m$. Hence we have $s.build\text{-}order(x_2.id, i)_q = m$ for each member of x_2.

It follows that (i, m, x_2) is good. We can iterate this reasoning for $x_3, ..., x_k$ and obtain that $s.order(i)_q = m$ for any process q such that $q \in state\text{-}dlv[c.id]$, as needed. ∎

Next we show that in a given state no two operations can be good for a particular index i.

Lemma 7.3.5 *In any reachable state, given an index i, there exists at most one operation m such that (i, m) is good.*

Proof: Fix any reachable state s. By contradiction assume that there exist m and m' such that (i, m) and (i, m') are good in s and such that $m \neq m'$.

By definition of good we have that there exists at least one configuration w' such that (i, m, w') is good in s. Let w_1 be the configuration with the smallest identifier among the configurations w' for which (i, m, w') is good. Of course (i, m, w_1) is good in s.

Similarly, by definition of good we have that there exists at least one configuration w' such that (i, m', w') is good in s. Let w_2 be the configuration with the smallest identifier among the configurations w' for which (i, m', w') is good. Of course (i, m', w_2) is good in s.

Without loss of generality assume that $w_1.id < w_2.id$.

We now distinguish two possible cases.

CASE 1: There exists $c \in s.\mathcal{E}st$ such that $w_2.id < c.id$. Fix $p \in s.state\text{-}dlv[c.id]$. By Invariant 7.3.4, applied with $w = w_1$, since (i, m, w_1) is good, we have that $s.order(i)_p = m$.

By the same invariant, applied with $w = w_2$, since (i, m', w_2) is good, we have that $s.order(i)_p = m'$. This is a contradiction since $m \neq m'$.

CASE 2: There is no $c \in s.\mathcal{E}st$ such that $w_2.id < c.id$. Since (i, m', w_2) is good in s we have that there exists a quorum $Q \in c.set$ such that $s.build\text{-}order(w_2.id, i) = m'$ for all members of Q. Fix $p \in Q$. We have $s.build\text{-}order(w_2.id, i)_p = m'$. Since there is no $c \in s.\mathcal{E}st$ such that $w_2.id < c.id$, we have that w_2 is the latest configuration established by p. Hence we have that $s.order(i)_p = m'$.

By Invariant 7.3.4, applied with $w = w_1$, we have that $s.order(i)_p = m$. This is a contradiction since $m \neq m'$. ▯

The following lemma generalizes the previous one by claiming that, even across an entire execution and not just in single state, we cannot have two different elements of $\mathcal{D}$ being stable at the same index.

Lemma 7.3.6 *Let α be an execution. Let s and s' be two states of α and let $m, m' \in \mathcal{D}$ be such that $m \neq m'$. Then it cannot be that (i, m) is good in s and (i, m') is good in s'.*

Proof: By definition of good we have that if (i, m) is good in a state s then (i, m) is good in any subsequent state s'. Then the lemma follows easily by Lemma 7.3.5. ▯

The following lemma states that an index i for which an answer has been computed is good.

Lemma 7.3.7 *In any reachable state we have that if $req\text{-}answrd(i)_p =$ `true` for some process p then i is good.*

Proof: Let s be a reachable state and assume that $s.req\text{-}answrd(i)_p =$ `true`.

Variable $req\text{-}answrd(i)_p$ is set to true by process p either when providing the answer for operation i with action P2P-SEND$(\langle \text{“}ans\text{”}, a\rangle)_{p,q}$ or when establishing a new configuration with action NEWSTATE$(\langle o, d, a\rangle)_p$.

In the former case process p is the leader of some configuration c in which the answer for operation i is computed. Process p computes such an answer only after informing a quorum $Q \in c.qrms$, by means of the underlying DLC service, about $order(i)_p$. Hence i is good in s.

In the latter case process p sets $req\text{-}answrd_p := a$. So $req\text{-}answrd(i)_p$ is `true` only if $a(i)$ is `true`. But $a(i)$ is true only if the leader p' of some previous configuration has computed an answer for operation i and thus has executed action P2P-SEND($\langle$"*ans*"$, a\rangle$) for i. Let s' be the state when p' computed the answer for operation i. Clearly i is good in s'. But once i is good in a state it stays good in all subsequent states. Hence i is good in s. $\square$

In order to prove that the system implements an atomic object we use the following lemma, which is a version of Lemma 13.16 of [65] (page 435) that considers general operations instead of specific read/write operations.

Lemma 7.3.8 *Let β be a (finite or infinite) sequence of actions of an atomic object external interface. Suppose that β is well-formed, and contains no incomplete operations. Let Π be the set of all operations in β. Suppose that $\prec$ is an irreflexive total ordering of the operations in Π, satisfying the following properties:*

1. *For any operation $A \in \Pi$, there are only finitely many operations B such that $B \prec A$.*
2. *If the response event for operation A precedes the invocation event for operation B in β, then $A \prec B$.*
3. *The response for any operation $A \in \Pi$ is the result of applying all the operations that precede A, including A itself, in the order $\prec$.*

Then β satisfies the atomicity property.

Finally we can give the following claim.

Claim 7.3.9 *The system* RAB *implements an atomic object.*

Proof: By Lemma 13.10 of [65] (page 419) we can restrict our attention to executions with only complete operations. Fix such an execution α. Remember that Π is the set of operations of α.

In order to show that the system implements an atomic object we need to provide a total order $\prec$ on Π that satisfies Lemma 7.3.8. Let us define the order $\prec$ as follows.

Let A be an operation in Π. By definition of Π we have that A gets completed. In order for A to complete there must be a leader that stores A in some position i and computes an answer for A setting *req-answrd*(i) to $\texttt{true}$. So there exists a state s of α such that $s.\mathit{req\text{-}answrd}(i)_p = \texttt{true}$ for some p. By Lemma 7.3.7 we have that (i, A) is good in s. Denote by $\mathit{tag}(A)$ the index i. This is well defined because of Lemma 7.3.6. Note that no two operations A and B can get the same tag i, because we would have that both (i, A) and (i, B) are good in some state contradicting Lemma 7.3.6.

Order all operations in Π in order of *tag*.

Next we prove that $\prec$ satisfies the hypothesis of Lemma 7.3.8.

Let us start with Point 1. Fix $A \in \Pi$. Any operation $B \prec A$ must have $\mathit{tag}(B) < \mathit{tag}(A)$. The number of such operations is bounded by $\mathit{tag}(A)$.

Now consider Point 2. Fix $A, B \in \Pi$ and assume that the response event for operation A precedes the invocation event for operation B.

Since $A, B \in \Pi$ there exists a state s of α and two indexes i, j such that (i, A) and (j, B) are good in s. By definition of *tag* we have that $i = tag(A)$ and $j = tag(B)$. Hence we need to prove that $i < j$.

Let s' be the state when B gets invoked. Since the response event of A precedes the invocation event of B, we have that the response event of A precedes s'.

Since A received a response before state s', we have that there exists a state s'' preceding s' such that $s''.req\text{-}answrd(i)_p = \texttt{true}$ for some process[2] p. By the code (see action P2P-SEND$(\langle``ans", a\rangle)_{p,q}$), we have that $s''.req\text{-}answrd(k)_p = \texttt{true}$ for any $k < i$. By Lemma 7.3.7 we have that in state s'' all indexes $k \leq i$ are good in s''. Since s'' precedes s', indexes $k \leq i$ are good in s' too.

This implies that for each index $k \leq i$ there exists an operation A_k such that (k, A_k) is good in s'. Moreover none of these operations A_k can be equal to B because B is invoked in state s' and thus cannot be good in state s'.

Remember that (j, B) is good in s. By Lemma 7.3.6 it cannot be that $j \leq i$ because for any index $k \leq i$ there exists an operation $A_k \neq B$ such that (k, A_k) is good in s'.

Hence it must be that $i < j$, as needed to prove Point 2.

Finally consider Point 3. This condition is true because responses are given in order of *tag* and by Lemma 7.3.6 this order is consistent for all operations in π. ∎

7.4 Remarks

The RAB algorithm implements an atomic shared object. As a particular case we may have a read-write register. In Chapter 6 the algorithm ABD also implements an atomic shared read-write register. The latter is built upon the DC service while the former is built upon the DLC service which is a variation of the DC service that defines a leader within each configuration. The two algorithms are similar (indeed they use similar services as building block): both rely on spreading each operation to a quorum of processes in order to keep data consistency. The main difference is that the RAB algorithm uses the leader of the current configuration in order to centralize the handling of the requests from the clients; within each configuration, the leader of the configuration is responsible for providing answers to the requests. With such an approach only the leader needs to have the most up to date information and thus, when the system is stable, this approach is more efficient. In the ABD algorithm at any time there is a quorum of processes which have the most up to date information.

[2]This process p is the leader of the configuration in which the answer to operation A is computed. However this is not important for this proof.

As the Liskov-Oki algorithm [76], the RAB algorithm uses a centralized approach where a distinguished process is responsible to perform requested operations; however, such a process needs the cooperation of a quorum of other processes in order to provide answers to the requested operations. Our algorithm is dynamic and does allow change in the universe of processes while the Liskov-Oki algorithm assumes a fixed universe of processes. The RAB algorithm uses a more conservative approach in providing answers to requested operations: the leader does not respond to a requested operation until it knows that a quorum of processes have recorded that operation. In the Liskov-Oki approach the leader immediately respond to requested operations; this is more efficient when there are no failures, but in case of failures it is less efficient (roll back might be necessary).

The MULTIPAXOS algorithm [61] can also be used to implement a replicated atomic object. Indeed processors can agree on the sequence of operations to perform on the shared object by running a sequence of instances of a consensus algorithm. The usefulness of developing the RAB algorithm is that we use a building block which provides a powerful service and thus much of the computation that needs to be done is delegated to the DLC building block. This results in a simpler algorithm. The overall algorithm is similar to an algorithm that would use the MULTIPAXOS approach; however the philosophy underlying the building blocks approach is that building blocks are built once and then can be used by many applications which can take advantage of powerful properties offered by the building block. With such a perspective designing RAB is easier than designing a replicated atomic object based on MULTIPAXOS (the interested reader can compare the code of MULTIPAXOS provided in [29] with the code of RAB).

Chapter 8

Conclusions

In this thesis we have provided a set of group communication specifications. We have also given implementations of the specifications and we have constructed applications on top of the specifications.

The main theme has been that of providing "dynamic" group communication specifications, that is, specifications for group communication services that adapt well to dynamic changes of the underlying distributed system. This is crucial in systems where processes can join or leave the system routinely because of process or link failures.

In such settings, it is possible that the underlying system suffers partitions. In the presence of partitions two approaches can be followed: one is to allow every component of the partition to proceed independently; another one is to select a unique "primary" component of the partition and allow progress only in that component. The former approach improves availability at the expense of shared data coherence. The latter is to be used when replicated data needs to be maintained coherently. Most group communication services and specifications take the first approach: they are partitionable.

When applications require a primary component but run over a partitionable group communication service, it is the responsibility of the application to figure out whether it is in a primary component or not. Establishing whether the current component is primary or not is clearly independent of the particular application. Thus it would be better to move this problem from the application to a lower level layer. One possibility is to use a primary component group communication service as building block.

In Chapter 5 we have considered the extension of existing partitionable group communication services to primary ones. We have provided a specification for a dynamic primary view group communication service called DVS. The communication tools provided by DVS are those typical of a group communication service; the membership service provides the client with primary views.

We have also shown that the DVS service is implementable. Our implementation is based on the VS service of Fekete, Lynch and Shvartsman [41] and uses ideas from the dynamic membership algorithm of Yeger Lotem, Keidar and Dolev [89]. The implementation filters the views provided by the VS service in order to establish whether the systems has partitioned and in such a case to report to the clients only views satisfying particular intersection properties with previous views. Such views are the primary components of the partition. By reporting to the clients only these primary views, the service enforces that computation proceed only in the primary component.
In order to show the usefulness of the DVS specification we have developed an application on top of it. The application we have developed provides a totally ordered broadcast service: clients are allowed to broadcast messages to all other members of the system and the service guarantees that each member of the systems receives the messages in the same order. This is a very powerful service to develop replicated data algorithms or any other application that necessitates data coherence.

In Chapter 6 we tackled the problem of extending dynamic primary view services to dynamic "configuration" services. A configuration is a view with a quorum system defined on the membership set of the view. The use of quorums is a well-known technique to improve availability and efficiency in a distributed system. With quorums usually a client request is serviced by a quorum of the set of all the members of the system (as opposed to the whole set of members). Our goal has been that of integrating the use of quorums in a group communication system. In particular we extended the DVS service to handle configurations. The result has been a specification for a primary configuration group communication service, called DC. The main difficulty in developing DC has been that of defining a dynamic primary configuration. The notion of dynamic primary view has been well studied (e.g., [55, 89]). As far as we know, there was no corresponding notion for configurations. We have developed such a notion and used it to specify the DC service.
As for the DVS service, we have proved that DC is implementable and useful. The implementation is very similar to that of DVS. It uses a static service internally, which provides any new configurations. Then it filters these configurations to find those that satisfy certain intersection properties. These configurations are the primary configurations which are given to the clients of the service.
The application we have developed on top of DC is a multi-writer/multi-reader atomic register. This application is based on the work of Attiya, Bar-Noy and Dolev [12] and that of Lynch and Shvartsman [66]. The algorithm exploits the quorum-oriented framework provided by the DC service.

Finally in Chapter 7 we have explored the use of the techniques deployed in the development of DVS and DC to the design of dynamic algorithms, i.e., algorithms that work well in dynamic distributed settings.
Lamport's PAXOS algorithm uses quorums to solve the consensus problem; however it is designed for

a static settings. We have used the DC service[1] in order to design a dynamic version of the PAXOS algorithm, a version that adapts well to system changes, even permanent ones.
We also have provided a dynamic primary copy data replication algorithm. As the DPAXOS algorithm, also this algorithm uses (a variant of) the DC service as a building block. We have sketched the proof of correctness of this algorithm (the formal proof is left as future work.)

Applications developed on top of powerful building blocks are easier to build than those built from scratch, because such applications can benefit from the guarantees provided by the building blocks. We have shown that our group communication building blocks are powerful enough to build interesting applications. We think that other applications can be built on top of the group communication services (or variations of them) provided in this thesis.

An interesting feature of the DC specification is that it integrates a state exchange mechanism within the service. When a new configuration is delivered to the client, the client is supposed to submit its current state to the service. Once the service receives the state from all the members of the configuration, it computes a new up-to-date state and delivers this state to each member of the configuration. In this way the state transfer is relegated to the DC service.
The DC service requires all members of the configuration to submit the state before computing a new up-to-date state. It would be interesting to explore the possibility of computing the new up-to-date state when only a quorum of the processes have submitted their state. Clearly the resulting service would be weaker, but it is possible that useful applications can still be constructed on top of this weaker service. The advantage would be a more available service.

One of the major goal of the current research in this area is to provide simple, universally accepted specifications that describe the semantics of the existing group communication services already in use in real-world application. Probably it is not possible to give a unique specification good for all applications: different applications will require different group communication services, which will be tailored to the applications. Another approach consists of defining independent protocol layers that implement different service levels and semantics (e.g., as is done for example in Horus and Ensemble). The application developer can use any combination of these layers building the right semantics for the needed group communication service.

Though much work has been focused in providing specifications for group communication services (we refer the reader to Chapter 2 for pointers to the literature or to [87] for a survey), the overall goal has not been achieved yet. It would be good to provide a set of universally accepted specifications for group communication services that cover all possible needs of applications. System implementors

[1] We actually have used a variant of the DC tailored to the particular application that we have developed. See Chapter 7 for more details.

could then concentrate on efficient implementations of such specifications and application developers could build their applications on top of the guarantees provided by the specifications.

In this perspective, the contribution of this thesis is that of having provided formal specifications for two particular group communication services tailored to applications that run in dynamic systems and that require primary views and configurations.

Possible future work that follows the direction of this thesis may include the following.

One could provide performance and fault-tolerance analysis of the algorithms presented in the thesis. We have focused our attention on the safety properties; though our algorithms are not naive[2], we have not proved any performance property. Such a performance analysis could be based on assumptions on the underlying physical distributed system (as it is done in [41]). Moreover since we were concerned only with safety our algorithms are not tuned for optimal performance.

Hence one could optimize the algorithms presented in the thesis and compare them with other algorithms. In particular it would be interesting to provide a dynamic version of the MULTIPAXOS algorithm built upon the DLC group communication service. In Chapter 7 we have provided a dynamic version of the PAXOS algorithm but not a dynamic version of the MULTIPAXOS algorithm. One could provide such a dynamic version of MULTIPAXOS and compare its performance with that of the original MULTIPAXOS algorithm (see [61, 29]). We have provided two algorithms that implement atomic objects: the ABD-SYS algorithm, which is based on a decentralized approach, and the RAB algorithm, which is based on a centralized approach. We have not tuned the code of these algorithms for efficiency. It would be interesting to provide optimized code and compare the performance of these algorithms with others that solve similar problems (e.g., the Liskov-Oki algorithm [76]).

Another possibility is to provide variations of the group communication services presented in this thesis. An interesting one is that of weakening the DC specification presented in Chapter 6 in order to allow the computation of the starting state for a new configuration as soon as the processes in a quorum of the configuration have submitted their state (the version we have used requires all the members of the configuration to submit their state). We believe that this weaker version is still powerful enough to build useful applications.

More generally it would be interesting to build other algorithms on top of the group communication service building blocks provided in this thesis and also to provide new building blocks tailored to other applications.

[2]An algorithm that does nothing is a safe algorithm.

Part II

Distributed k-set Consensus

Chapter 9

Distributed k-set Consensus: Overview

The problem of reaching consensus in a distributed system arises in many forms and various contexts, such as, for example, distributed data replication, distributed databases, flight control systems. Data replication is used in practice to provide high availability: having more than one copy of the data allows easier access to the data, i.e., the nearest copy of the data can be used. However, consistency among the copies must be maintained. A consensus algorithm can be used to maintain consistency. Another practical example of the use of data replication is an airline reservation system. The data consists of the current booking information for the flights and it can be replicated at agencies spread over the world. The current booking information can be accessed at any of the replicas. Reservations or cancellations must be agreed upon by all the copies.

Distributed consensus has been extensively studied; a good survey of early results is provided in [42]. We refer the reader to [65] for a more up-to-date treatment of consensus problems.

One of the most celebrated results about distributed consensus is the impossibility result of Fischer, Lynch and Paterson [43]. This impossibility result, popularly known as FLP, states that it is impossible to achieve distributed consensus in asynchronous systems even if only one stop failures is possible. This surprising result sparkled various directions of research aimed to solve the problem by either restricting the asynchrony of the computation model [31, 35], or using randomized protocols [18, 21, 80], or weakening the problem definition [24, 32, 39, 40].

The last of these three directions of research falls in the more general research area of demarcating what is deterministically computable and what is deterministically impossible in asynchronous distributed systems in the presence of failures. The FLP impossibility seemed to suggest that no nontrivial problem could be solved deterministically and asynchronously in the presence of faults. Attiya, Bar-Noy, Dolev, Peleg and Reischuk [13] showed that the renaming problem can be solved in a deterministic way in asynchronous system in the presence of failures. Informally, in the renaming problem processors start the computation with a "name" taken from some unbounded ordered name space and have to "rename" themselves with names chosen from a new small name space. This result revived the research trend of exploring computable and impossible in deterministic asynchronous distributed systems subject to failures. Following this direction, Chaudhuri [24] defined the k-set consensus (or k-consensus for short) problem, which is a natural generalization of the consensus problem obtained by allowing processes to decide on k different values, instead of requiring them to agree on a single value. The 1-consensus problem is the classical consensus problem.

Chaudhuri provided an algorithm to solve the k-consensus problem that tolerates up to a threshold t of process failures strictly smaller than k. This result proved that the k-consensus problem, for $k \geq 2$, allows more resilience than the 1-consensus problem. Chaudhuri conjectured that the k-consensus problem was impossible to solve while tolerating k or more failures. This conjecture was proven true by three independent research teams: Borowsky and Gafni [20], Herlihy and Shavit [52] and Saks and Zaharoglou [84]. Attiya [11] provided an alternative proof of the same result.

The results of [24, 20, 52, 84] completely characterize the k-consensus problem in asynchronous systems with stop failures. In such a model the k-consensus problem is solvable if and only if $t < k$.

The formal definition of the k-consensus problem requires three conditions to be satisfied: *agreement*, *termination* and *validity.* The agreement condition requires that each process decide on a value in such a way that the set of decided values has cardinality at most k. The termination condition simply requires that each (correct) process decide. For what concern the validity condition, several variants have been considered in the literature. The validity condition used in [24, 20, 52, 84] requires that each of the decision be equal to some input value.

An alternative definition of the validity condition considered for the 1-consensus problem with stop failures requires that if all the inputs to the processes of the systems are equal then any decision must be equal to the input (see, for example [65, Ch. 6]). This condition is the one considered for the k-consensus problem.

In a Byzantine environment faulty processes can "mask" their inputs. Hence a more suitable validity condition considered for the 1-consensus problem with Byzantine failures requires that if all the correct processes have the same input then any decision be the input of a correct process [62, 78].

In this thesis we explore the solvability of the k-set consensus problem in asynchronous message passing models in which processes fail by stopping or fail arbitrarily (Byzantine failures). The main theme is that the validity condition has a profound impact on when the problem is solvable. We consider six different validity conditions and use these conditions to demarcate when k-set consensus is solvable for each system model. In several cases we completely characterize solvability. In some we characterize solvability with very little uncertainty (i.e., a small gap between computable and impossible) and in a few cases we leave a substantial gap.

More in details, we start from the validity condition used by Chaudhuri, which we call the "regular" validity condition (the decision of any correct process is the input of some process) and denote by RV1, and consider a weakened version (if there are no failures then every decision is the input of some process), denoted with WV1, and a strengthened version (the decision of any correct process is the input of some correct process), denoted by SV1. For each of these three validity condition we consider a corresponding weakened version obtained by requiring the condition only if all the processes start with the same input value. We denote these validity conditions with SV2 (if all correct processes start with v the correct processes decide v), RV2 (if all processes start with v then correct processes decide v) and WV2 (if there are no failures and all processes start with v, then any decision is equal to v).

For the crash failures model we completely characterize the line that separates possible from impossible for each of the above six validity conditions, with the exception of validity SV2 where a tiny gap is left open.

For the Byzantine failures model we characterize the line line that separates possible from impossible leaving a tine gap for SV2, RV2 and WV2, and a substantial gap for WV1.

The rest of this part is structured as follows. In Chapter 10 we give the formal definition of the k-consensus problem. We study the k-set consensus problem for crash failure in Chapter 11. Chapter 12 presents the results for Byzantine failures.

Chapter 10

The problem

We consider a distributed system consisting of n processes denoted by $p_1, p_2, ..., p_n$. A process that follows its algorithmic specification throughout an execution is said to be *correct*, and a process that departs from its specification is said to be *faulty.* In a *fail-stop* model (also known as a *crash* model), faulty processes are allowed to prematurely halt execution only. In a *Byzantine* model, a faulty process can deviate from its specification arbitrarily. We assume that at most t processes fail, where $t \geq 1$ is a known, positive integer.

We assume that the system is asynchronous. Processes communicate by sending messages over a network. We are not concerned with the particular topology of the network. Since we consider asynchronous systems, messages can take arbitrarily long time to reach their destination. We only assume that the network of processes is connected, that is, any process can send a message to any other process. Moreover, messages are not altered, lost or duplicated while in transit on the network.

For any k, $1 \leq k \leq n$, we denote a k-set consensus problem by $\mathcal{KC}(k)$ or simply $\mathcal{KC}$ when k is not relevant. The $\mathcal{KC}(k)$ problem is defined as follows. Each process p_i starts the computation with an input value v_i. Each correct process has to irreversibly "decide" on a value in such a way that three conditions, called *termination*, *agreement* and *validity*, hold. These conditions are:

Termination: Every correct process eventually decides.

Agreement: The set of values decided by correct processes has size at most k.

Validity: One of the following conditions.

- SV1 (strong V1): The decision of any correct process is equal to the input of some correct process.
- SV2 (strong V2): If all correct processes start with v then correct processes decide v.
- RV1 (regular V1): The decision of any correct process is equal to the input of some process.

RV2 (regular V2): If all processes start with v then correct processes decide v.

WV1 (weak V1): If there are no failures, then the decision of any process is equal to the input of some process.

WV2 (weak V2): If there are no failures and all processes start with v, then the decision of any process is equal to v.

Given a validity condition C, we denote by $\mathcal{KC}(k,C)$ the $\mathcal{KC}(k)$ problem defined with validity C. We also use the notation $\mathcal{KC}(C)$ if k is not relevant. We use the notation $\mathcal{KC}(k,t)$ to denote a $\mathcal{KC}(k)$ consensus problem with at most t failures allowed. The notation $\mathcal{KC}(k,t,C)$ denotes $\mathcal{KC}(k,t)$ with validity C.

We define a partial order on the $\mathcal{KC}$ problems based on the strength of the validity conditions. We say that $\mathcal{KC}(C)$ is *weaker* than $\mathcal{KC}(D)$ if any algorithm for solving $\mathcal{KC}(D)$ can be used to solve $\mathcal{KC}(C)$ in a given model. Clearly $\mathcal{KC}(C)$ is weaker than $\mathcal{KC}(D)$ if any impossibility result that holds for $\mathcal{KC}(C)$ holds also for $\mathcal{KC}(D)$. Conversely, we say that $\mathcal{KC}(C)$ is *stronger* than $\mathcal{KC}(D)$ if $\mathcal{KC}(D)$ is weaker than $\mathcal{KC}(C)$. Figure 10-1 shows the "weaker than" relation between the validity conditions.

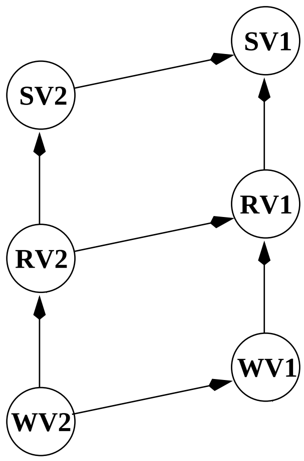

Figure 10-1: Validity conditions. An arrow from a validity condition C to a validity condition D means that $\mathcal{KC}(C)$ is weaker than $\mathcal{KC}(D)$ (and that $\mathcal{KC}(D)$ is stronger than $\mathcal{KC}(C)$).

$\mathcal{KC}(k,$RV1$)$ is the consensus problem as considered by Chaudhuri [24]. $\mathcal{KC}(1,$RV1$)$ and $\mathcal{KC}(1,$RV2$)$ are classical consensus problems (see, e.g., [65, Ch. 6]). $\mathcal{KC}(1,$SV2$)$ has been considered in the Byzantine setting [62, 78]. $\mathcal{KC}(1,$WV2$)$ is weak Byzantine agreement [60].

It is well known that the case $k = 1$ cannot be solved for any nontrivial validity condition and, in particular, for any of the validity conditions that we consider here, for any $t \geq 1$ [43]. On the other hand, if $k = n$ then $\mathcal{KC}(k)$ is trivially solvable (each process decides its own value), even in the Byzantine setting, for any t and with the strongest validity condition we are considering, that is, validity SV1. Thus, we will henceforth be concerned only for the cases $2 \leq k \leq n - 1$. Since the problem is easily solvable for $t = 0$ we also assume that $t \geq 1$.

Chapter 11

Crash failures

In this section we consider the crash failures model. In Section 11.1 we recall known results. In Sections 11.2 and 11.3, we provide further impossibility results and protocols, respectively. Figure 11-1 shows a graphical representation of the results provided in this section.

11.1 Known results

As noted in Section 9, for the crash failure models we already know the line between computable and impossible for $\mathcal{KC}(k,\ t,\text{RV}1)$:

Lemma 11.1.1 ([24]) *In the crash model, there is a protocol for $\mathcal{KC}(k,t,\text{RV}1)$, for $t < k$.*

Lemma 11.1.2 ([20, 52, 84]) *In the crash model, there is no protocol for $\mathcal{KC}(k,t,\text{RV}1)$, for $t \geq k$.*

By Lemma 11.1.1, we have that $\mathcal{KC}(k,t,\text{RV}2)$, $\mathcal{KC}(k,t,\text{WV}1)$ and $\mathcal{KC}(k,t,\text{WV}2)$ are solvable for $t < k$ because RV2, WV1 and WV2 are weaker than RV1. By Lemma 11.1.2, $\mathcal{KC}(k,t,\text{SV}1)$ cannot be solved for $t \geq k$ because SV1 is stronger than RV1.

11.2 Impossibilities

In this section we provide impossibility results for the crash model. An ingredient in most of our impossibility results is the fact that in any protocol tolerating t failures, a process must be able to decide after communicating with at most $n-t$ processes (including itself). Indeed, if a process waited to communicate with more than $n-t$ processes, termination could not be achieved: the runs in which there were exactly t faulty processes that do not send any messages, would not terminate.

Lemma 11.2.1 *In the crash model, there is no protocol for $\mathcal{KC}(k,t,\text{WV}2)$, for $t \geq \frac{(k-1)n+1}{k}$.*

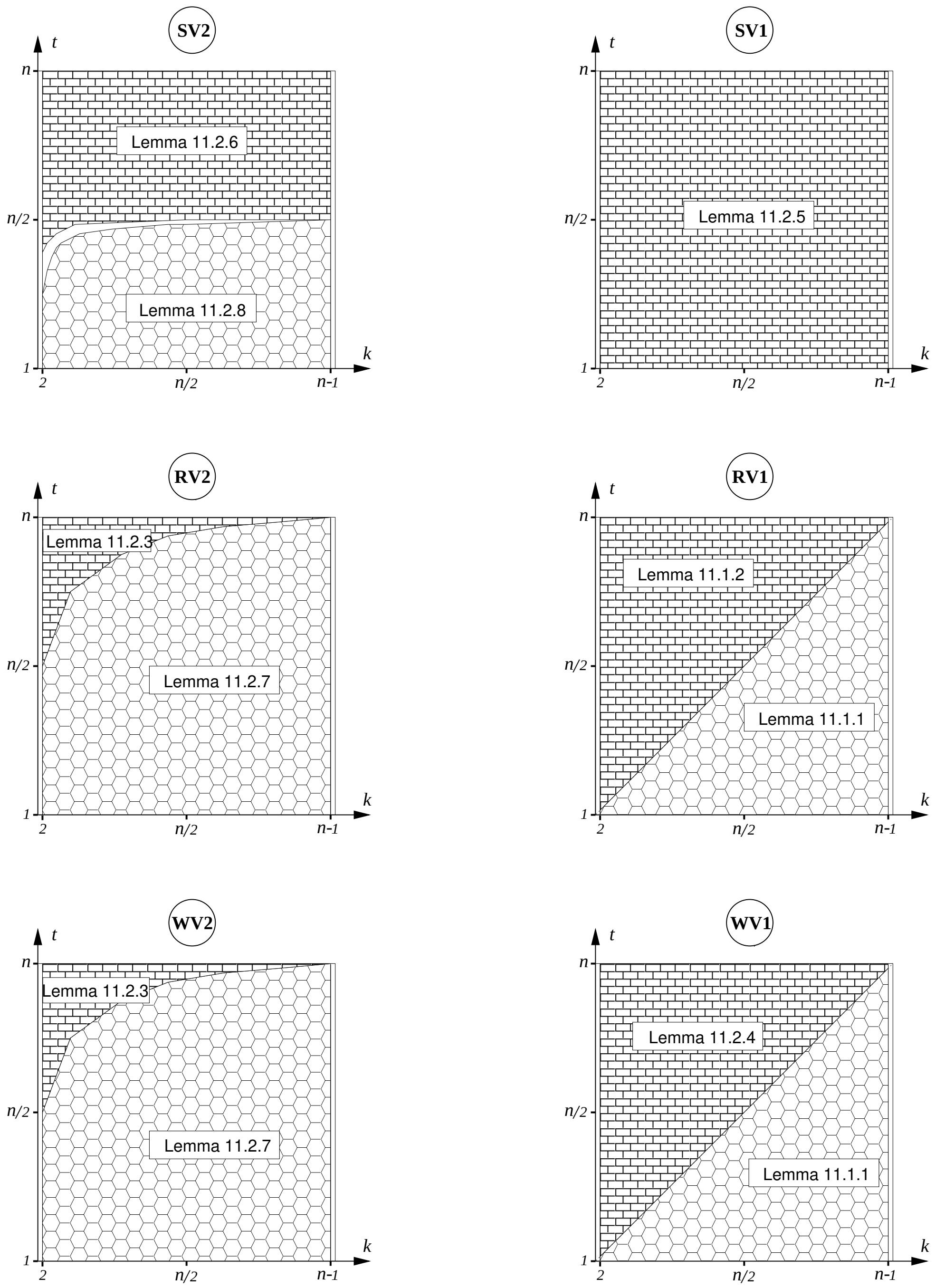

Figure 11-1: Crash model. Regions filled in brick pattern indicate impossibility. Regions filled in honeycomb pattern indicate solvability. Unfilled regions indicate open problems. Figures are drawn to scale $n = 64$.

Proof: For a contradiction, assume that such a protocol A exists. In the rest of the proof we use the notation $\mathcal{KC}_P(k,t,C)$ to explicitly state the set P of processes among which k-set consensus is to be solved. Denoting by $\mathcal{P}$ the set of all processes, we have that A solves $\mathcal{KC}_{\mathcal{P}}(k,t,\text{WV}2)$.

Since $t \geq ((k-1)n+1)/k$ implies $n \geq k(n-t)+1$, we can partition the n processes into k groups $g_1, g_2, ..., g_k$ of disjoint processes with $g_1, , ..., g_{k-1}$ containing exactly $n-t$ processes and g_k containing at least $n-t+1$ processes. If $t = n$ we let $g_1, g_2, ..., g_{k-1}$ be singleton sets of processes and we let g_k contain at least two processes (this is possible because we only consider $k < n$).

First we claim that there is a run of A where only processes in g_k take steps and such that two values are decided. To see why, assume that all the runs involving only processes of g_k are such that only one value is decided. Then we could use A to solve $\mathcal{KC}_{g_k}(1,1,\text{WV}2)$: g_k contains at least $n-t+1$ processes, so that even if one of them is faulty we still have at least $n-t$ correct processes in g_k and hence the protocol has to terminate. However, this contradicts [43], since no such protocol exists. Hence there is a run α_k in which only processes in g_k take steps and they decide on at least two different values, say v_k, v_{k+1}. Let $v_1, ..., v_{k-1}$ be $k-1$ values different from v_k, v_{k+1}.

Fix i, $i \in \{1, 2, ..., k-1\}$ and consider the following run α_i: all processes are correct, all start with v_i and all messages sent to processes in g_j, $j = 1, 2, ..., k$ by processes not in g_j are delayed until all processes in g_j make a decision. We can use A to solve $\mathcal{KC}_{\mathcal{P}}(k,t,\text{WV}2)$ and by validity WV2 we have that all processes, in particular those in group g_i, decide v_i.

Now consider the following run α. All processes are correct, for each i, $i = 1, 2, ..., k-1$, each process in g_i starts with v_i and processes in g_k start with the same values they start in α_k. Moreover for each i, $i = 1, 2, ..., k$, all messages sent to processes in group g_i by processes not in g_i are delayed until all processes in g_i have decided. We can use A to solve $\mathcal{KC}_{\mathcal{P}}(k,t,\text{WV}2)$ in α. However, for each i, $i = 1, 2, ..., k$, processes in g_i cannot distinguish between run α_i and run α. Indeed in both runs they only communicate with processes in g_i before making a decision and in both runs processes in g_i start with the same value. Since, for $i = 1, 2, ..., k-1$, in run α_i processes in g_i decide v_i, they must decide v_i also in α. Since in run α_k processes in g_k decide on v_k and v_{k+1}, they must decide v_k and v_{k+1} also in α. Hence we have that $k+1$ values are decided in α. Thus the agreement condition is violated and this contradicts the hypothesis that A solves $\mathcal{KC}_{\mathcal{P}}(k,t,\text{WV}2)$. ∎

Lemma 11.2.2 *In the crash model, there is no protocol for $\mathcal{KC}(k,t,\text{WV}1)$, for $t \geq k$.*

Proof: For a contradiction assume that there exists such a protocol A. We claim that A can be used to solve $\mathcal{KC}(k,t,\text{RV}1)$ for $t \geq k$. To see why, consider any run α in which $f \leq t$ processes are faulty and let g be the set of correct processes and g' be the set of faulty processes. Now consider a run α' that is identical to α except that all processes are correct and any message sent by any $p \in g'$ in α' after the time that p failed in α is delayed until after all processes in g decide. That is, for each $p_i \in g$ and each $p_j \in g'$, p_i receives a message from p_j at time T in α' iff p_i receives the same message at time T from p_j in α. By the validity condition WV1, each process decides on some process' input in α'. Clearly, processes in g cannot distinguish between α and α'. Hence, processes in g decide the same value in α as they decide in α', and so validity RV1 is satisfied in α. In other words, protocol A solves $\mathcal{KC}(k,t,\text{RV}1)$ for $t \geq k$, contradicting Lemma 11.1.2. $\square$

Lemma 11.2.3 *In the crash model, there is no protocol for $\mathcal{KC}(k,t,\text{SV}1)$.*

Proof: For a contradiction assume that there exists such a protocol A. Let α be an execution of A in which all processes are correct and they all start with different values. Let v a decision made by at least two processes (there is always such a decision since $k < n$). Because of validity SV1, v is the input of some process p_i and since all inputs are different only p_i has v as input. Now consider the run α' that is the same as α except that process p_i fails right after sending its last message. Clearly α and α' are indistinguishable and thus each process (maybe with the exception of p_i) makes the same decision in both runs. Hence in α' value v is decided by at least one process p_j, $j \neq i$. But only p_i has v as input and p_i is not correct in α', and so validity SV1 is violated. $\square$

Lemma 11.2.4 *In the crash model, there is no protocol for $\mathcal{KC}(k,t,\text{SV}2)$, for $t \geq \frac{k}{2k+1}n$.*

Proof: For a contradiction assume that there exists such a protocol A. Consider first the case $t \geq \frac{n}{2}$. Partition the system into two non-intersecting sets of processes, g, g', each containing at least $n-t$ processes (e.g., $|g| = |g'| = n/2$). This is always possible because $t \geq n/2$. Let α be a run of A in which all processes are correct, all start with different initial values denoted $v_1, v_2, ..., v_n$, and all communication between g and g' is delayed until after the decisions are made. We claim that n values are decided in α. To see this, fix any process $p_i \in g$, and consider the following run α_i. The processes in g start with the same values as in α, and all except p_i crash after p_i reaches a decision. All the processes in g' start with v_i but communication between g and g' is delayed until after p_i makes a decision. By $sv2$, p_i must decide v_i in α_i, and by indistinguishability of α from α_i, p_i must decide v_i in α. Similarly, runs α'_i can be constructed for every process $p'_i \in g'$, and hence all processes must decide their own values in α. This contradicts the hypothesis that A solves the problem (for $k < n$).

Now consider the case $t < \frac{n}{2}$. In this case, $n - 2t > 0$ and the condition $t \geq n\frac{k}{2k+1}$ is equivalent to $k \leq \frac{n-t}{n-2t} - 1$. Let g be a subset of the system containing $n-t$ processes, and let $g_1, ..., g_{\lfloor \frac{n-t}{n-2t} \rfloor}$ be a partition of g into disjoint sets of size at least $n-2t$ each. Let α be a run of A in which all the processes are correct, communication between g and the rest of the system is delayed until after all processes have decided and, for each i, processes in g_i start with a distinct value v_i. Fix i, and let $p_i \in g_i$ be some process. Consider a run α_i of A as follows: Processes in g_i are correct, all processes in $g \setminus g_i$ are faulty, and crash after p_i decides. All communication between g and the rest of the system is delayed until after p_i decides. By $sv2$, p_i must decide v_i, but since α is indistinguishable to p_i from α_i, p_i must decide v_i in α. Therefore, in α, at least $\lfloor \frac{n-t}{n-2t} \rfloor$ different values are decided on. This contradicts the hypothesis that A solves the problem since $k \leq \frac{n-t}{n-2t} - 1 < \lfloor \frac{n-t}{n-2t} \rfloor$. ∎

11.3 Protocols

In this section we provide two protocols for the crash model.

> PROTOCOL A: Each process broadcasts its input and waits for $n-t$ messages. If all $n-t$ messages contain the same value v, then the process decides v, else it decides a default value v_0.

Lemma 11.3.1 PROTOCOL A *solves* $\mathcal{KC}(k,t,\text{RV}2)$ *in the crash model for* $t < \frac{k-1}{k}n$.

Proof: We start by proving termination. The number of actual failures is less or equal to t. Hence there are at least $n-t$ correct processes. Thus each correct process eventually receives at least $n-t$ messages and is able to make a decision.

Now we prove agreement. By the sake of contradiction assume that $k+1$ values are decided. One of them could be the default value, but at least k values, different from the default value, are decided. By the protocol it is necessary that there be k disjoint sets $g_1, g_2, ..., g_k$, each consisting of at least $n-t$ processes such that each process in g_i sends a value v_i (with $v_i \neq v_j$ for $i \neq j$). Hence there must be at least $k(n-t)$ processes. However since $t < \frac{k-1}{k}n$ we have that $n-t > n/k$ and that $k(n-t) > n$, which implies that there must be more than n processes. This is impossible since we have n processes.

Finally we prove validity. Assume that all processes start with value v. Clearly a process cannot receive two different values since v is the only value being sent. Hence by the protocol each process that makes a decision, decides v. ∎

> PROTOCOL B: Each process broadcasts its input and waits for $n-t$ messages. One of these $n-t$ messages is the process' own message. If $n-2t$ messages contain the same value as its own, say v, the process decides v, else it decides a default value v_0.

Lemma 11.3.2 PROTOCOL B *solves* $\mathcal{KC}(k,t,\text{SV}2)$ *in the crash model for* $t < \frac{k-1}{2k}n$.

Proof: We start by proving termination. The number of actual failures is less or equal to t. Hence there are at least $n-t$ correct processes. Thus each correct process eventually receives at least $n-t$ messages and is able to make a decision.

Now we prove agreement. By the sake of contradiction assume that $k+1$ values are decided. One of them could be the default value, but at least k values, different from the default value, are decided. By the protocol it is necessary that there be k disjoint sets $g_1, g_2, ..., g_k$, each consisting of at least $n-2t$ processes such that each process in g_i sends a value v_i (with $v_i \neq v_j$ for $i \neq j$). Hence there must be at least $k(n-2t)$ processes. However since $t < \frac{k-1}{2k}n$ we have that $k(n-2t) > n$, which implies that there must be more than n processes. This is impossible since we have n processes.

Finally we prove validity. Assume that all correct processes start with value v. We have to prove that a correct process decides v. Let p be a correct process. First we observe that since p starts with v it decides v or v_0. Hence it suffices to prove that p receives at least $n-2t$ messages with v. Among the $n-t$ messages p receives at least $n-2t$ are from correct processes. Hence process p receives at least $n-2t$ messages with v. $\square$

11.4 Remarks

For $\mathcal{KC}(\text{RV}2)$ and $\mathcal{KC}(\text{WV}2)$, there is a very tiny gap between our possibility and impossibility results (Lemmas 11.2.1 and 11.3.1), formed by the cases where n is a multiple of k. These are isolated points on the line that separates possible from impossible. Since for all other points on this line the problem is not solvable it would be very surprising if for those isolated points the problem is solvable. For $\mathcal{KC}(\text{SV}2)$ there is also small gap between our possibility and impossibility results (Lemmas 11.2.4 and 11.3.2).

Chapter 12

Byzantine failures

In this section we consider the Byzantine failures model. In Section 12.1 we are concerned with impossibilities and in Section 12.2 we provide protocols. Figure 12-1 shows a graphical representation of the results.

12.1 Impossibilities

In this section we provide impossibility results for the Byzantine model. Clearly the impossibilities proved for the crash model still hold. In particular the impossibilities for $\mathcal{KC}(\text{SV1})$ and $\mathcal{KC}(\text{WV1})$ are directly derived from the corresponding ones for the crash model. Next we provide additional impossibilities.

Lemma 12.1.1 *In the Byzantine model, there is no protocol that solves $\mathcal{KC}(k,t,\text{WV2})$, for $t \geq \frac{k}{2k+1}n$ and $t \geq k$.*

Proof: For a contradiction assume that such a protocol A exists. We distinguish two cases: (i) $t \geq n/2$ and (ii) $t < n/2$.

Consider case (i). Let $v_1, v_2, ..., v_{t+1}$ be $t+1$ different values. Let α be the following run of A. The number of actual failures in α is $f = n - t - 1$. Let F be the set of faulty processes and let $p_1, ...p_{t+1}$ be the correct processes. Process p_i has input v_i, for $i = 1, 2, ..., t+1$. Messages between any two correct processes are delayed until all correct processes decide, that is, correct processes communicate only with processes in F.

We now show that at least $k+1$ values are decided in α, which contradicts the hypothesis that A solves the problem. For each $i = 1, 2, ..., t+1$ consider the following run α_i. All processes are correct, all have input v_i, messages between processes not belonging to F are delayed until all processes not in F decide. By validity WV2, we have that in α_i all processes must decide v_i. Process p_i, for $i = 1, 2, ..., t+1$, cannot distinguish between α and α_i, if in α, the members of F behave as if they

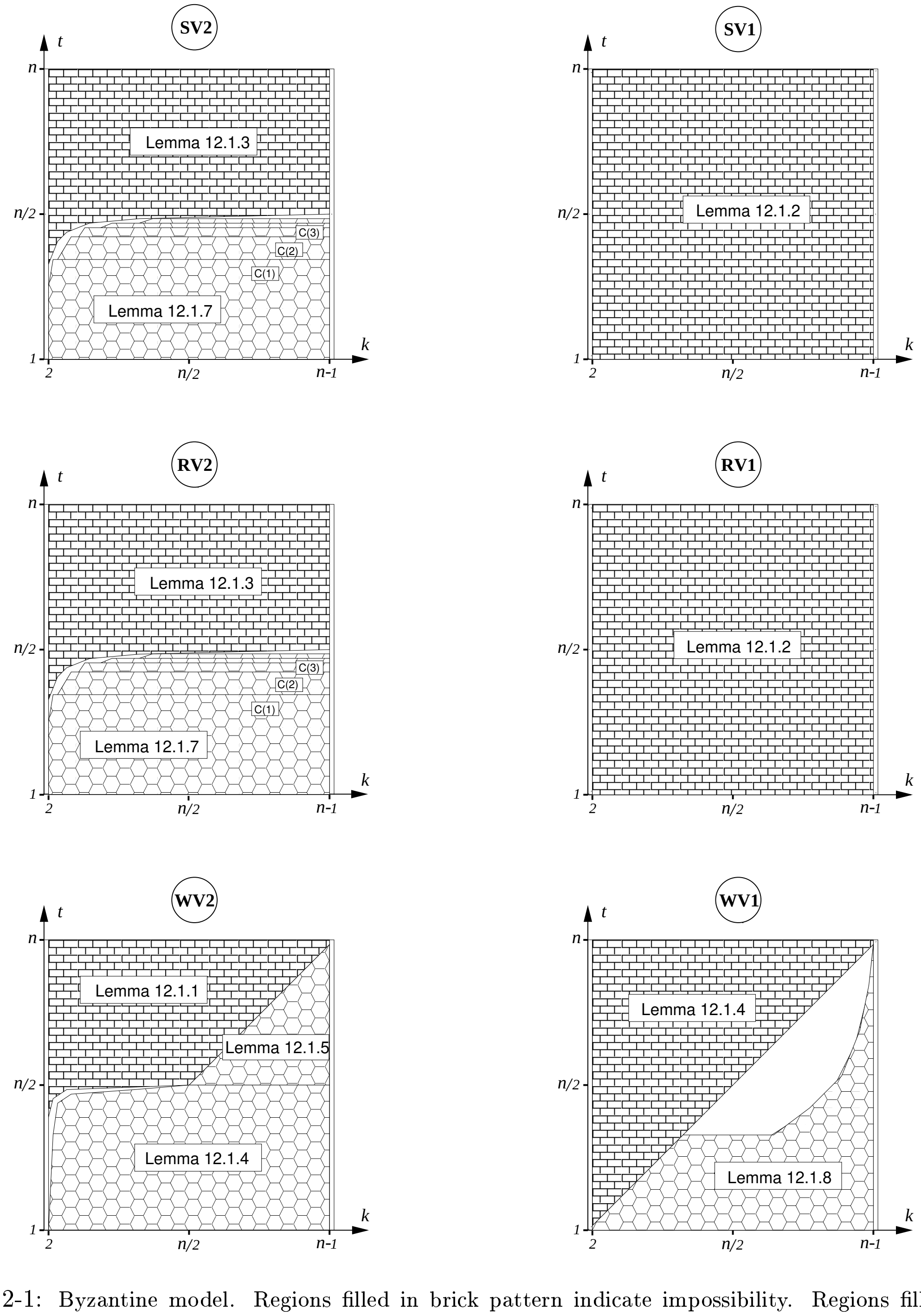

Figure 12-1: Byzantine model. Regions filled in brick pattern indicate impossibility. Regions filled in honeycomb pattern indicate solvability. Unfilled regions indicate open problems. Figures are drawn to scale $n = 64$.

were correct and had v_i initially. Hence p_i has to decide the same value in both runs. We have that process p_i decides v_i also in α. Since $v_1, v_2, ..., v_{t+1}$ are different, we have that $t+1$ values are decided in α. But $t \geq k$, hence at least $k+1$ values are decided in α.

Consider case (ii). Since $t < n/2$ we have that $n - 2t > 0$ and thus the condition $t \geq \frac{k}{2k+1}n$ is equivalent to $\frac{n-t}{n-2t} \geq k+1$. Then, we can partition the processes into $k+2$ groups, the first $k+1$ of which, denoted $g_1, g_2, ..., g_{k+1}$, each consists of at least $n-2t$ processes, and the last of which, denoted F, consists of t processes. Let α be the following run of A. Let $v_1, v_2, ..., v_{k+1}$ be $k+1$ different values. Processes in g_i start with v_i, for $i = 1, 2, ..., k+1$, and processes in F are faulty. Processes in group g_i communicate only within g_i and with processes in F. For each group g_i processes in F behave as correct processes with input v_i.

We now show that at least $k+1$ values are decided in α, which contradicts the hypothesis that A solves the problem. For each $i = 1, 2, ..., k+1$ consider the following run α_i. All processes are correct, all have input v_i, processes in group g_i communicate only within g_i and with processes in F. By validity WV2, we have that in α_i all processes must decide v_i. Processes in g_i, for $i = 1, 2, ..., k+1$, cannot distinguish between α and α_i. Hence they have to decide the same value in both runs, and so processes in g_i decide v_i also in α. Since $v_1, v_2, ..., v_{k+1}$ are different, we have that $k+1$ values are decided in α. □

Lemma 12.1.2 *In the Byzantine model, there is no protocol that solves $\mathcal{KC}(k,t,\text{RV}1)$.*

Proof: For a contradiction assume that such a protocol A exists. Let α_1 be a run of A in which all processes are correct and each start with a different input value. Let $v_1, ..., v_z$ be the set of values decided by correct processes. Because A satisfies validity RV1, each of the v_i is the input of some process. Since $z \leq k < n$, we have that there exists a value v_i, $1 \leq i \leq z$, decided by at least two processes, say p_1 and p_2.

Let process q be the process whose input in α_1 is v_i for some $i \in \{1, ..., z\}$. Use A in the run α_2 in which q is faulty but behaves as in α_1, claiming that v_i is its input, but it has v_i' as its input, with v_i' different from v_i and also from any other input. Since correct processes cannot distinguish between α_1 and α_2 they have to decide on the same value. We now distinguish three possible cases: (1) q is different from both p_1 and p_2, (2) q is p_1 and (3) q is p_2. If q is different from both p_1 and p_2 then both p_1 and p_2 are correct and thus they decide on v_i in α_2. However v_i is not an input value in α_2. Hence validity is violated. If q is p_1 (resp. p_2) then p_2 (resp. p_1) is correct and thus decides v_i in α_2. However v_i is not an input value in α_2. Hence validity RV1 is violated. This contradicts the hypothesis that A solves $\mathcal{KC}(k,t,\text{RV}1)$. □

Lemma 12.1.3 *In the Byzantine model, there is no protocol for $\mathcal{KC}(k,t,\text{RV}2)$, for $t \geq \frac{k}{2(k+1)}n$.*

Proof: The proof is similar to that for Lemma 11.2.4. For a contradiction assume that such a protocol A exists. We distinguish two cases: (i) $t < n/2$ and (ii) $t \geq n/2$. Consider case (i). Since $t < n/2$ we have that $n - 2t > 0$ and thus the condition $t \geq \frac{k}{2(k+1)}n$ is equivalent to $\frac{n}{n-2t} \geq k + 1$. Then, we can partition the processes in $k + 1$ groups each consisting of at least $n - 2t$ processes. Consider case (ii). In this case we partition the processes in $k+1$ groups each consisting of at least one process.

In both cases, let $g_1, g_2, ..., g_k, g_{k+1}$ be the $k + 1$ groups of processes. Let $v_1, ...v_{k+1}$ be $k + 1$ different values and consider the following run α. All processes are correct, processes in group g_i start with v_i. For each group g_i, there is a set of t processes not belonging to g_i, call it F_i, such that, for each i, communication is allowed only among processes in g_i and F_i until all processes have decided. Notice that the cardinality of $g_i \cup F_i$ is at least $n - t$ in both cases.

We now show that $k+1$ values are decided in α, which contradicts the hypothesis that A solves the problem. Fix i, $1 \leq i \leq k+1$, and consider run α_i. There are exactly t faulty processes and these processes are those in F_i. Processes in g_i are correct. All processes start with v_i. Faulty processes behave exactly as they do in run α. Processes in g_i communicate only with other processes in g_i and F_i. We can use A to solve $\mathcal{KC}(k,t,\text{RV}2)$, and by the validity RV2 we have that all correct processes, and in particular those in g_i decide v_i. Processes in g_i cannot distinguish run α and run α_i. Hence, since they decide v_i in α_i they have to decide v_i also in α. It follows that $k + 1$ values are decided in α. ∎

12.2 Protocols

In this section we provide protocols for the Byzantine model. We start by observing that PROTOCOL A, used for the crash model, solves $\mathcal{KC}(\text{WV}2)$ also in the Byzantine model, though only for a restricted range of values of k and t.

Lemma 12.2.1 PROTOCOL A *solves* $\mathcal{KC}(k,t,\text{WV}2)$ *in the Byzantine model for* $t < n/2$ *and* $k \geq \frac{n-t}{n-2t} + 1$.

Proof: We start by proving termination. Since there are at most t failures, correct processes are guaranteed to receive at least $n - t$ messages and thus they decide.

Next we prove agreement. To have a bound on the number of possible decisions we look at how many values different from the default value can be decided. Let f be the number of actual failures. We have that any group of $n - t - f$ correct processes that start with the same value can be forced by the f faulty processes to decide that value. Notice that since $f \leq t < n/2$ we have that $n - t - f \geq 1$. Hence the number of decisions can be as big as the number of possible disjoint groups of $n - t - f$ correct processes, plus one to take into account the default value. There can be at most $(n-f)/(n-t-f)$ such groups. This function is an increasing function of f and thus it achieves its maximum value for $f=t$. Hence the number of different decisions we can have is at most $(n-t)/(n-2t)+1$. Since $k \geq (n-t)/(n-2t)+1$ agreement is satisfied.

Finally we prove validity. Assume that all processes are correct and start with v. Then clearly v is the only decision. □

Lemma 12.2.2 PROTOCOL A *solves $\mathcal{KC}(k,t,\text{WV}2)$ in the Byzantine model for $t \geq n/2$ and $k \geq t+1$.*

Proof: Termination and validity are as in the previous lemma. Next we prove agreement. Let f be the number of actual failures. We distinguish two cases: (i) $f \leq n-t-2$ and (ii) $f > n-t-2$. In case (i) we have that for any $n-t$ messages received by a process, at least two of them are sent by correct processes. Hence for each different value $v \neq v_0$ decided by some process at least two correct processes have sent that value. Hence no more than $n/2$ values different from the default value v_0 can be decided. Hence at most $n/2+1$ different values can be decided in case (i). In case (ii) the number of correct processes is strictly less than $t+2$. Hence we cannot have more than $t+1$ different decisions. Putting together the two cases, we have that the number of different decisions is at most $\max\{n/2+1, t+1\} = t+1 \leq k$. □

Next we provide a generalized version of the "echo" protocol of Bracha and Toueg [22], which we call ℓ-echo, where $\ell \geq 2$. (The 1-echo protocol is Bracha and Toueg's echo protocol.) The ℓ-echo protocols will be used to provide a family of protocols for $\mathcal{KC}(\text{SV}2)$.

> ℓ-`echo protocol:` To ℓ-echo broadcast a message m, the sender s sends the message $\langle$`init`$,s,m\rangle$ to all other processes. When a process p receives the first $\langle$`init`$,s,m\rangle$ from s, it sends the message $\langle$`echo`$,s,m\rangle$ to all other processes. Subsequent `init` messages from s are ignored. If process p receives message $\langle$`echo` $,s,m\rangle$ from more than $(n+\ell t)/(\ell+1)$ processes, then process p *accepts* message m as sent by the sender process s.

Lemma 12.2.3 *In a system with $t < \ell n/(2\ell+1)$, if a sender s uses the ℓ-echo protocol to send a message m then:*

(i) Correct processes accept at most ℓ different messages.

(ii) If s is correct, every correct process accepts m.

Proof: First we prove (i). By sake of contradiction assume that correct processes accept $\ell+1$ different messages $m_1, m_2, ..., m_{\ell+1}$. Then there must be $\ell+1$ correct processes, say $p_1, p_2, ..., p_{\ell+1}$, such that process p_i receives more than $(n+\ell t)/(\ell+1)$ echos with m_i, for each $i = 1, 2, ..., \ell+1$. Thus there must be a total of more than $n+\ell t$ echos sent for the messages $m_1, m_2, ..., m_{\ell+1}$. Let f be the actual number of faulty processes. Since a faulty process can send $\ell+1$ different echos (it can echo m_1

to p_1, m_2 to p_2 and so on) we have that strictly more than $n+\ell t-(\ell+1)f \geq n+\ell f-(\ell+1)f = n-f$ echos are sent by correct processes. This implies that at least one correct process sent two different echos, which is not possible.

Now we prove (ii). If the sender is correct, then it sends an `init` message for m to all other processes. Any correct process will receive this and broadcast an echo message for m. Because there are at most $t \leq (n+\ell t)/(\ell+1)$ faulty processes, no correct process accepts any message other than m. Since there are at least $n-t$ correct processes, it is sufficient that $n-t$ be strictly greater than $(n+\ell t)/(\ell+1)$ in order to guarantee that any correct process receives enough echo messages to be able to accept m. Since $t < \ell n/(2\ell+1)$ we have that $n-t > (n+\ell t)/(\ell+1)$. ∎

The ℓ-echo protocol is used to define a family of protocols for $\mathcal{KC}(k,t,\text{SV}2)$ as follows.

> PROTOCOL C(ℓ): Each process broadcasts its input using the ℓ-echo protocol and waits for $n-t$ messages to be accepted, where one of these $n-t$ messages is the process' own message. If $n-2t$ messages contain the same value v, then the process decides v, else it decides a default value v_0.

Lemma 12.2.4 PROTOCOL C(ℓ) *solves $\mathcal{KC}(k,t,\text{SV}2)$ in the Byzantine model for $t < \frac{k-1}{2k+\ell-1}n$ and $t < \frac{\ell}{2\ell+1}n$.*

Proof: We start by proving termination. Since there are at least $n-t$ correct processes, each correct process eventually accepts at least $n-t$ messages broadcast by ℓ-echo and is able to make a decision.

Now we prove agreement. For a contradiction assume that $k+1$ values are decided. One of them could be the default value, but at least k values, different from the default value, are decided. By the protocol it is necessary that there be k sets $g_1, g_2, ..., g_k$, each consisting of at least $n-2t$ processes, such that some correct process accepts a value v_i from each process in g_i (with $v_i \neq v_j$ for $i \neq j$). Hence correct processes accept at least $k(n-2t)$ values broadcast by ℓ-echo. Each faulty process can contribute ℓ different values, and so the number of different senders is at least $k(n-2t)-(\ell-1)t$. However since $t < \frac{k-1}{2k+\ell-1}n$, we have that $k > \frac{n+(\ell-1)t}{n-2t}$ and thus $k(n-2t)-(\ell-1)t > n$, which implies that there must be more than n processes, a contradiction.

Finally we prove validity. Assume that all correct processes start with value v. We have to prove that a correct process decides v. Let p be a correct process. First we observe that since p starts with v it either decides v or v_0. Hence it suffices to prove that p receives at least $n-2t$ messages with v. Among the $n-t$ messages p receives at least $n-2t$ are from correct processes. Hence process p receives at least $n-2t$ messages with v. ∎

Finally we provide a protocol for $\mathcal{KC}(\text{WV}1)$.

> PROTOCOL D: Processes $p_1, p_2, ..., p_{t+1}$ each broadcasts its input value. A process that receives a value v_i from p_i, $i \in \{1,2,...,t+1\}$, broadcasts an $\langle$`echo`,v_i,$p_i\rangle$ message and

never echos a value for p_i again. Processes $p_1, p_2, ..., p_k$ each decides on its own value. Every other process decides the first value v_i, $i \in \{1, \dots, t+1\}$, for which it receives identical echos $\langle \texttt{echo}, v_i, p_i \rangle$ from $n - t$ processes.

In PROTOCOL D, we say that a process *accepts* a value v_i from p_i if it receives identical echos for v_i from at least $n - t$ processes. We define the following functions

$$V(n,t,f) = \begin{cases} n-f & \text{if } n-t-f \leq 0 \\ t+1-f+f\lfloor \frac{n-f}{n-t-f} \rfloor & \text{if } n-t-f > 0 \end{cases}$$

and

$$Z(n,t) = \max_{0 \leq f \leq t} \{\min\{V(n,t,f), n-f\}\}.$$

Lemma 12.2.5 PROTOCOL D *solves* $\mathcal{KC}$*(k,t,*WV1*) in the Byzantine model for* $k \geq Z(n,t)$.

Proof: We start by proving termination. At least one process among $p_1, ..., p_{t+1}$ is correct, and at least $n-t$ receive its value and echo it. Hence it is guaranteed that each correct process receives at least one set of identical $n-t$ echo messages and thus is able to decide.

Next we prove validity. Assume that there are no failures. Then all processes are correct and thus the values accepted by any process are input values. All decisions are one of the accepted values. Hence validity WV1 is satisfied.

Finally we prove agreement. We compute an upper bound on the number of different decisions for each possible value of f, that is the number of actual failures. By definition, $0 \leq f \leq t$. We distinguish two cases: (i) $n-t-f \leq 0$ and (ii) $n-t-f > 0$. In case (i) a correct process may be forced to communicate only with faulty processes. In this case we simply bound the number of decisions with the number of correct processes, that is $n - f$. In case (ii) the total number of values that correct processes accept from one faulty process is bounded by $\lfloor \frac{n-f}{n-t-f} \rfloor$. Indeed, a correct process accepts a value when receiving at least $n-t$ echos, at least $n-t-f$ of which are from correct processes. Thus the total number of values from $p_1, ..., p_{t+1}$ accepted by correct processes is at most $(t+1-f) + f\lfloor \frac{n-f}{n-t-f} \rfloor$, that is the number of values sent by correct processes plus the number of values that correct processes may be forced to accept because of the Byzantine behavior of faulty processes. Hence the number of different decisions that we can have is $t+1-f+f\lfloor \frac{n-f}{n-t-f} \rfloor$. It is possible that this bound is bigger than $n-f$. In such a case, we can bound the number of different decisions by $n-f$. Summarizing the two cases we have that for any f, we bound the number of decisions by $n-f$ if $n-t-f \leq 0$ and by $\min\{t+1-f+f\lfloor \frac{n-f}{n-t-f} \rfloor, n-f\}$ if $n-t-f > 0$. The maximum over all possible values of f is given by $Z(n,t)$. Hence we have that the number of decisions is always at most $Z(n,t)$, as required. $\square$

We note that when $t < \frac{n}{3}$, $\lfloor \frac{n-f}{n-t-f} \rfloor = 1$ for all $0 \leq f \leq t$, and therefore, the protocol above guarantees agreement for any $k > t$ (see Figure 12-1).

12.3 Remarks

For the Byzantine model, the impossibility results and protocols we have provided in this section leave a small gap for the $\mathcal{KC}$ problem defined with validities WV2, RV2 and SV2 and a substantial gap for $\mathcal{KC}$(WV1). An interesting open problem is to fill in this gap.

Chapter 13

Conclusions

The k-set consensus problem is an abstraction of many coordination problems in a distributed system that can suffer process failures. In this thesis we have investigated the k-set consensus problem in asynchronous message passing distributed systems. We have extended previous work by exploring several variations of the problem definition and model, including for the first time investigation of Byzantine failures. We have shown that the precise definition of the validity requirement, which characterizes what decision values are allowed as a function of the input values and whether failures occur, is crucial to the solvability of the problem. For example, we show that allowing default decisions in case of failures makes the problem solvable for most values of k in face of minority-failure, even in face of the most severe type of failures (Byzantine). We have introduced six validity conditions for this problem (all considered in various contexts in the literature), and demarcate the line between possible and impossible for each case. In many cases this line is different from the one of the originally studied k-set consensus problem.

In this thesis we have considered asynchronous systems. A natural question to ask is: what happens in synchronous systems? Clearly any algorithm that works in asynchronous systems works also in synchronous systems.

Let us first consider the case of stop failures. The FloodSet algorithm (see for example [65, Ch. 7]) solves the $\mathcal{KC}$ problem in synchronous systems with stop failures. It tolerates any number of failures, that is, it works for any $t \leq n$. The validity condition considered is validity RV1. Hence this algorithm works also for validities RV2, WV1 and WV2. The impossibility proof for $\mathcal{KC}$ validity SV1 that we have provided for asynchronous systems works also for synchronous systems. Hence there is no $\mathcal{KC}$ protocol for validity SV1, synchronous systems, stop failures. The above cover pretty much all the cases we have considered. The only open case is validity SV2: we can use the algorithm for asynchronous system that solves the problem for $t < n/4$ when $k =$ and for $t < n/3$ for $k \geq 3$. For other cases we don't know.

For the Byzantine case there is no work on $\mathcal{KC}$ for synchronous system. It is known that $\mathcal{KC}$(1,SV2) can be solved if and only if $t < n/3$ [78, 62]. The EIGbyz algorithm (see [65]) provides a solution $\mathcal{KC}$(1,SV2) when $t < n/3$. Lamport [60] proved that also $\mathcal{KC}(1,t,\text{WV2})$ can be solved if and only if $t < n/3$. Clearly EIGbyz solves also $\mathcal{KC}$(1,WV2). No results for $\mathcal{KC}(k)$, $k \geq 2$, are known for synchronous systems with Byzantine failures. Obviously one can use the algorithm provided in this thesis for asynchronous system. However this still leaves large gaps for the values of k and t for which we don't know if the problem is solvable or not.

Another natural question to ask is what happens in shared memory systems. Algorithms that work for message passing systems work also for shared memory system because a channel can be simulated with shared memory. The FLP impossibility proof can be generalized to shared memory ([64]). The impossibility result of [20, 52, 84] works also for shared memory. In some of the impossibility proofs we provided in this thesis we used the fact that the system is message-passing; hence we do not know whether the impossibility results still hold in the shared memory settings. We conjecture that the techniques used in this thesis can be used to provide a similar analysis for the shared memory models.

Bibliography

[1] ACM. *Communications of the ACM 39(4), special issue on Group Communications Systems*, April 1996.

[2] D. Agrawal and A. El Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 9(1):1–20, 1991.

[3] Ehab S Al-Shaer, Hussein Abdel-Wahab, and Kurt Maly. HiFi: A new monitoring architecture for distributed system management. In *19th International Conference on Distributed Computing Systems (ICDCS)*, pages 171–178, June 1999.

[4] Y. Amir, D. Breitgand, G. Chockler, and D. Dolev. Group communication as an infrastructure for distributed system management. In *3rd International Workshop on Services in Distributed and Networked Environment (SDNE)*, pages 84–91, June 1996.

[5] Y. Amir, G. V. Chokler, D. Dolev, and R. Vitenberg. Efficient state transfer in partitionable environments. In *2nd European Research Seminar on Advances in Distributed Systems (ERSADS'97)*, pages 183–192. BROADCAST (ESPRIT WG 22455), Operating Systems Laboratory, Swiss Federal Institute of Technology, Lausanne, March 1997. Full version available as Technical Report CS98-12, Institute of Computer Science, The Hebrew University, Jerusalem, Israel.

[6] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. In *22nd IEEE Fault-Tolerant Computing Symposium (FTCS)*, July 1992.

[7] Y. Amir, D. Dolev, P. Melliar-Smith, and L. Moser. Robust and efficient replication using group communication. Technical Report CS94-20, Institute of Computer Science, Hebrew University, Jerusalem, Israel, 1994.

[8] Y. Amir and A. Wool. Optimal availability quorum systems: Theory and practice. *Information Processing Letters*, 65:223–228, 1998.

[9] T. Anker, G. Chockler, I. Keidar, M. Rozman, and J. Wexler. Exploiting group communication for highly available video-on-demand services. In *Proceedings of the IEEE 13th International Conference on Advanced Science and Technology (ICAST 97) and the 2nd International Conference on Multimedia Information Systems (ICMIS 97)*, pages 265–270, April 1997.

[10] T. Anker, D. Dolev, and I. Keidar. Fault tolerant video-on-demand services. In *19th International Conference on Distributed Computing Systems (ICDCS)*, pages 244–252, June 1999.

[11] H. Attiya. A direct proof of the asynchronous lower bound for k-set consensus. In *Proceedings of the* 17^{th} *ACM Symposium on Principle of Distributed Computing (PODC)*, page 314. Puerto Vallarta, Mexico, 1998.

[12] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message passing systems. *Communications of the ACM*, 42(1):124–142, 1995.

[13] H. Attiya, A. Bar-Noy, D. Dolev, and D. Peleg. Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524–548, July 1990.

[14] Ö. Babaoğlu, R. Davoli, L. Giachini, and M. Baker. Relacs: A communication infrastructure for constructing reliable applications in large-scale distributed systems. TR UBLCS94-15, Department of Computer Science, University of Bologna, 1994.

[15] Ö. Babaoğlu, R. Davoli, L. Giachini, and P. Sabattini. The inherent cost of strong-partial view synchronous communication. In *Proceedings of Workshop on Distributed Algorithms on Graphs*, pages 72–86, 1995.

[16] Ö. Babaoğlu, R. Davoli, and A. Montresor. Failure detectors, group membership and view-synchronous communication in partitionable asynchronous systems. TR UBLCS95-18, Department of Computer Science, University of Bologna, 1995.

[17] Ö. Babaoğlu, R. Davoli, and A. Montresor. Partitionable Group Membership: Specification and Algorithms. TR UBLCS97-1, Department of Computer Science, University of Bologna, January 1997.

[18] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the* 2^{nd} *ACM Symposium on Principle of Distributed Computing (PODC)*, pages 27–30. Montreal, Canada, 1983.

[19] K.P. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, Los Alamitos, CA, 1994.

[20] E. Borowsky and E. Gafni. Generalized flp impossibility result for t-resilient asynchronous computations. In *Proceedings of the* 25^{th} *ACM Symposium on Theory of Computing (STOC)*, pages 91–100, 1993.

[21] G. Bracha. An $o(n \log n)$ expected rounds randomized byzantine generals algorithm. In *Proceedings of the* 4^{th} *ACM Symposium on Principle of Distributed Computing (PODC)*, 1985.

[22] G. Bracha and S. Toueg. Resilient consensus protocols. In *Proceedings of the* 2^{nd} *ACM Symposium on Principle of Distributed Computing (PODC)*, pages 12–26, 1983.

[23] T.D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 322–330, Philadelphia, Pennsylvania, May 1996.

[24] S. Chaudhuri. Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1), July 1993.

[25] F. Cristian. Reaching agreement on processor group membership in synchronous distributed systems. *Distributed Computing*, 4(4), 1991.

[26] F. Cristian. Group, majority and strict agreement in timed asynchronous distributed systems. In *Proceedings of the 26th Conference on Fault-Tolerant Computer Systems*, pages 178–187, 1996.

[27] F. Cristian and F. Schmuck. Agreeing on processor group membership in asynchronous distributed systems. Technical Report CSE95-428, University of California-San Diego, La Jolla, CA 92093-0114, 1995.

[28] D. Davcev and W. Buckhard. Consistency and recovery control for replicated files. In ACM Symp. on Operating Systems Principles, volume 10, pages 87–96, 1985.

[29] R. De Prisco. Revisiting the Paxos algorithm. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, June 1997. Also MIT/LCS/TR-717.

[30] R. De Prisco, B. Lampson, and N. Lynch. Revisiting the Paxos algorithm. In *Proceedings of the* 11^{th} *Workshop on Distributed Algorithms (WDAG)*, volume 1320 of *Lecture Notes in Computer Science*, pages 111–125. Springer-Verlag, 1997. Will appear in Theoretical Computer Science.

[31] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal syncrhony needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[32] D. Dolev, N. Lynch, S.S. Pinter, E.W. Stark, and W.E. Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 33(3):499–516, July 1986.

[33] D. Dolev and D. Malkhi. The transis approach to high availability cluster communications. *Communications of the ACM*, 39(4):64–70, 1996.

[34] D. Dolev, D. Malkhi, and R. Strong. A framework for partitionable membership service. Technical Report TR95-4, Institute of Computer Science, Hebrew University, Jerusalem, Israel, March 1995.

[35] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[36] A. El Abbadi and S. Dani. A dynamic accessibility protocol for replicated databases. *Data and knowledge engineering*, 6:319–332, 1991.

[37] A. El Abbadi and S. Toueg. Maintaining availability in partitioned replicated databases. *ACM Transactions on Database Systems*, 14(2):264–290, 1989.

[38] P. D. Ezhilchelvan, A. Macedo, and S. K. Shrivastava. Newtop: a fault tolerant group communication protocol. In *15th International Conference on Distributed Computing Systems (ICDCS)*, June 1995.

[39] A. Fekete. Asymptotically optimal algorithms for approximate agreement. *Distributed Computing*, 4(1):9–29, March 1990.

[40] A. Fekete. Asynchronous approximate agreement. *Information and Computation*, 115(1):95–124, November 15 1994.

[41] A. Fekete, N. Lynch, and A.A. Shvartsman. Specifying and using a partitionable group communication service. In *Proceedings of the* 16^{th} *ACM Symposium on Principle of Distributed Computing (PODC)*, pages 53–62, Santa Barbara, CA, August 1997.

[42] M.J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). Research Report YALEU/DCS/RR-273, Yale University, Department of Computer Science, New Haven, CT 06520, June 1983.

[43] M.J. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[44] R. Friedman and R. van Renesse. Strong and weak virtual synchrony in Horus. Technical Report TR95-1537, Department of Computer Science, Cornell University, Ithaca, NY, 1995.

[45] R. Friedman and A. Vaysburg. Fast replicated state machines over partitionable networks. In *16th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, October 1997.

[46] R. Friedman and A. Vaysburg. High-performance replicated distributed objects in partitionable environments. Technical Report 97-1639, Dept. of Computer Science, Cornell University, Ithaca, NY 14850, USA, July 1997.

[47] D. Gifford. Weighted voting for replicated data. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 150–162, 1979.

[48] R. Guerraoui and A. Schiper. Transaction model vs virtual synchrony model: bridging the gap. In *Theory and Practice in Distributed Systems*, LNCS 938, pages 121–132. Springer-Verlag, September 1995.

[49] Mark Hayden. *Ensemble Reference Manual.* Cornell University, 1996.

[50] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, 1986.

[51] M. Herlihy. Dynamic quorum adjustment for partitioned data. *ACM Transactions on Database Systems*, 12(2):170–194, June 1987.

[52] M. Herlihy. The asynchronous computability theorem for t-resilient tasks. In *Proceedings of the* 25^{th} *ACM Symposium on Theory of Computing (STOC)*, pages 111–120, 1993.

[53] M. Hiltunen and R. Schlichting. Properties of membership services. In *Proceedings of the 2nd International Symposium on Autonomous Decentralized Systems*, pages 200–207, 1995.

[54] F. Jahanian, S. Fakhouri, and R. Rajkumar. Processor group membership protocols: Specification, design and implementation. In *Proceedings of the 12th IEEE Symposium on Reliable Distributed Systems*, pages 2–11, 1993.

[55] S. Jajodia and D. Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Trans. Database Systems*, 15(2):230–280, 1990.

[56] I. Keidar. A Highly Available Paradigm for Consistent Object Replication. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994. Also Institute of Computer Science, The Hebrew University of Jerusalem Technical Report CS95-5, and available from: `http://www.cs.huji.ac.il/~transis/publications.html`.

[57] I. Keidar and D. Dolev. Efficient message ordering in dynamic networks. In *Proceedings of the* 15^{th} *ACM Symposium on Principle of Distributed Computing (PODC)*, pages 68–76, May 1996.

[58] Roger Khazan. Group communication as a base for a load-balancing, replicated data service. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, June 1998.

[59] Roger Khazan, Alan Fekete, and Nancy Lynch. Multicast group communication as a base for a load-balancing replicated data service. In *12th International Symposium on Distributed Computing*, pages 258–272, Andros, Greece, September 1998.

[60] L. Lamport. The weak byzantine generals problem. *Journal of the ACM*, 30(3):254–280, 1983.

[61] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998. Also Research Report 49, DEC SRC, Palo Alto, CA, 1989.

[62] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[63] N. Lesley and A. Fekete. Providing virtual synchrony for group communication services. Preprint, 1997.

[64] M. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. In *Parallel and Distributed Computing*, pages 163–183. JAI Press, Greenwich CT, 1987. Volume of *Advances in Computing Research*.

[65] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, March 1996.

[66] N. Lynch and A.A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of the* 27^{th} *Annual International Symposium on Fault-Tolerant Computing (FTCS)*, pages 272–281, Seattle, Washington, USA, June 1997. IEEE.

[67] N. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands. Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, November 1988.

[68] D. Malkhi and M.K. Reiter. Byzantine quorum systems. *Distributed Computing*, 11:203–13, 1998.

[69] D. Malkhi, M.K. Reiter, and A. Wool. The load and availability of byzantine quorum systems. In *Proceedings of the* 16^{th} *ACM Symposium on Principle of Distributed Computing (PODC)*, pages 249–257, August 1997.

[70] D. Malkhi, M.K. Reiter, and R. Wright. Probabilistic quorum systems. In *Proceedings of the* 16^{th} *ACM Symposium on Principle of Distributed Computing (PODC)*, pages 267–273, August 1997.

[71] C. Malloth and A. Schiper. View synchronous communication in large scale networks. In *2nd Open Workshop of the ESPRIT project BROADCAST (Number 6360)*, July 1995 (also available as a Technical Report Nr. 94/84 at Ecole Polytechnique Fédérale de Lausanne (Switzerland), October 1994).

[72] L. Moser, Y. Amir, P. Melliar-Smith, and D. Agrawal. Extended virtual synchrony. In *Proceedings of the 14th IEEE International Conference on Distributed Computing Systems*, pages 56–65, Poznan, Poland, June 1994. Full version appears in TR ECE93-22, Dept. of Electrical and Computer Engineering, University of California, Santa Barbara, CA.

[73] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4), April 1996.

[74] M. Naor and A. Wool. The load, capacity and availability of quorum systems. *SIAM Journal on Computing*, 27(2):423–447, April 1998.

[75] G. Neiger. A new look at membership services. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 331–340, Philadelphia, Pennsylvania, May 1996.

[76] B. Oki and B. Liskov. Viewstamped replication: A general primary copy method to support highly available distributed systems. In *Proceedings of the Seventh ACM Symposium on Principles of Distributed Computing*, pages 8–17, Toronto, Ontario, Canada, August 1988.

[77] J. Paris and D. Long. Efficient dynamic voting algorithms. In *Proceedings of the* 13^{th} *International Conference on Very Large Data Base*, pages 268–275, 1988.

[78] M. Pease, R. Shostak, and L. Lamport. Reaching agreeement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[79] D. Peleg and A. Wool. The availability of quorum systems. *Information and Computation*, 123(2):210–223, 1995.

[80] M. Rabin. Randomized byzantine generals. In *Proceedings of the* 15^{th} *ACM Symposium on Theory of Computing (STOC)*, pages 403–409, 1983.

[81] A. Ricciardi. The group membership problem in asynchronous systems. Technical Report TR92-1313, Department of Computer Science, Cornell University, Ithaca, NY, 1992.

[82] A. Ricciardi and K.P. Birman. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the* 10^{th} *ACM Symposium on Principle of Distributed Computing (PODC)*, pages 341–352, August 1991.

[83] A. Ricciardi, A. Schiper, and K.P. Birman. Understanding partitions and the "no partitions" assumption. Technical Report TR93-1355, Department of Computer Science, Cornell University, Ithaca, NY, 1993.

[84] M. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. In *Proceedings of the* 25^{th} *ACM Symposium on Theory of Computing (STOC)*, pages 101–110, 1993.

[85] A. Schiper and M. Raynal. From group communication to transactions in distributed systems. *Communications of the ACM*, 39(4):84–87, April 1996.

[86] R. van Renesse, K.P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.

[87] R. Vitenberg, I. Keidar, G. V. Chockler, and D. Dolev. Group Communication Specifications: A Comprehensive Study. Technical report, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, September 1999. Also Technical Report MIT-LCS-TR-790, Massachusetts Institute of Technology, Laboratory for Computer Science and Technical Report CS0964, Computer Science Department, the Technion, Haifa, Israel.

[88] E. Yeger Lotem, I. Keidar, and D. Dolev. Dynamic voting for consistent primary components. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 63–71, Santa Barbara, CA, August 1997.

[89] E. Yeger Lotem, I. Keidar, and D. Dolev. Dynamic voting for consistent primary components. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 63–71, Santa Barbara, CA, August 1997.