Group Communication as a Base for a Load-Balancing Replicated Data Service

by

Roger I Khazan

B.A., Computer Science and Mathematics
Summa Cum Laude, Highest Honors in Computer Science
Brandeis University (1996)

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

Master of Science

in

Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1998

Author . . . .
Department of Electrical Engineering and Computer Science
May 22, 1998

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

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

Group Communication as a Base for a Load-Balancing Replicated Data Service

by

Roger I Khazan

Submitted to the Department of Electrical Engineering and Computer Science on May 22, 1998, in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering and Computer Science

Abstract

Replication and load-balancing are two fundamental techniques for improving availability and performance of distributed systems. However, correct and efficient realization of these techniques is intricate when the distributed environment may partition and merge because of processor and communication failures.

In this thesis, we show how a view-synchronous group communication service recently specified by Fekete, Lynch, and Shvartsman can be used to support a sequentially consistent replicated data service that load balances queries and tolerates partitioning and merging of the underlying network.

Our work is done in the framework of the I/O automaton model of Lynch and Tuttle.

First, we present an I/O automaton specification that defines the allowed behavior of a sequentially consistent replicated service. Then, we successfully refine this specification to arrive at an I/O automaton that models a distributed implementation of this service. In this implementation, update requests are processed in the same order at all servers of the system, thus guaranteeing mutual consistency of all replicas; query requests are processed at single servers determined by a load-balancing strategy which equalizes the number of queries assigned to each member of the same group. Third, we give a rigorous hierarchical proof that the implementation automaton implements the specification automaton in the sense of trace inclusion. This proof establishes partial correctness of the implementation. Finally, we prove a liveness-related claim that servers are always able to process the queries assigned to them; that is, the servers are never blocked by missing update information.

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

Acknowledgments

I am grateful to Dr. Nancy Lynch for her guidance and many insightful discussions throughout this project. I am also grateful to Dr. Alan Fekete who, together with Dr. Nancy Lynch, suggested this project and helped me make the first steps.

I owe many thanks to Dr. Alex Shvartsman for introducing me to Distributed Computing back when I was an undergraduate and for encouraging me to pursue this field during my graduate studies.

I thank my fellow graduate students: Carl Livadas, Victor Luchangco, Roberto De Prisco, Mandana Vaziri, and Josh Tauber for their companionship. I am especially thankful to Carl Livadas and Victor Luchangco for insightful discussions and help.

For the past four years, Inna Zaslavsky has been an endless source of inspiration, emotional support, encouragement, and love. I thank her for that and for everything else she has given me.

I dedicate this thesis to my parents, Dr. Leonid Khazan and Lana Brodsky, who taught me so many precious lessons in life. Their devotion and belief in my abilities has been the driving force behind my accomplishments. To them I owe eternal gratitude.

Contents

List of Figures

Chapter 1

Introduction

Multicast group communication services are important building blocks for fault-tolerant applications that require reliable and ordered communication among multiple parties. These services manage their clients as collections of dynamically changing groups and provide strong intra-group multicast primitives. Recently, in an effort to remedy the existing lack of good specifications for these services and to facilitate consensus on what properties these services should exhibit, Fekete, Lynch, and Shvartsman gave a simple automaton specification *VS* for a *view-synchronous* group communication service and demonstrated its power by using it to support a totally-ordered broadcast application *TO* [13, 14]. In this thesis, we further investigate the power of *VS* by using it to support an intricate and important application: a *replicated data service* that *load-balances* queries, guarantees *sequential consistency*, and tolerates *partitioning* and *merging* of the underlying network.

1.1 Service Description

The service maintains a data object replicated at a fixed set of servers in a consistent and transparent fashion and enables the clients to *update* and *query* this object. We assume the underlying network is *asynchronous*, *strongly-connected*, and subject to processor and communication failures and recoveries. The failures and recoveries may cause the network or its components to partition and merge. The greatest challenge for the service is coping with network partitioning while preserving correctness and maintaining liveness.

We assume that executed updates cannot be undone, which implies that update operations must be processed in the same order at each replica. To avoid inconsistencies, the algorithm allows updates to occur only in *primary* components. Following the commonly used definition, primary components are defined as those containing a *majority* (or more generally, a *quorum*) of all servers. The nonempty intersection of any two majorities (quorums) guarantees the existence of at most one primary at a given time and allows for the necessary flow of information between consecutive primaries. Our service guarantees processing of update requests whenever there is a stable primary component, regardless of the past network perturbations.

On the other hand, processing of queries is not restricted to primary components and is guaranteed provided that the client's component eventually stabilizes. The service uses a round-robin load-balancing strategy to distribute queries to each server evenly within each component. This strategy makes sense in commonly occurring situations when queries take approximately the same amount of time, and this time is significant. Each query is processed with respect to a data state that is at least as advanced as the last state witnessed by the query's client. The service is arranged in such a way that the servers are always able to process the queries assigned to them; that is, they are not blocked by missing update information.

Architecturally, the service consists of two layers: the servers' layer and the communication layer. The servers' layer is symmetric: all servers run identical state-machines. The communication layer consists of two parts, a group communication service satisfying *VS*, and a collection of individual channels providing reliable reordering point-to-point communication between all pairs of servers. The servers use the group communication service to disseminate update and query requests to the members of their groups and rely on the properties of this service to enforce the formation of identical sequences of update requests at all servers and to schedule query requests correctly. The point-to-point channels are used to return the results of processed queries back to their original servers.

1.2 Related Work

1.2.1 Group Communication

A good overview of the rational and usefulness of group communication services is given in [4]. Examples of implemented group communication services are Isis [5], Transis [10], Totem [24], Newtop [12], Relacs [3], and Horus [27]. Different services differ in the way they manage groups and in the specific message ordering and delivery properties of their multicast primitives. Even though there is no consensus on which properties these services should provide, a typical requirement is to deliver messages in *total order within a single group.*

To be most useful, group communication services have to come with precise descriptions of their behavior. Many specifications have been proposed using a range of different formalisms [3, 6, 9, 11, 15, 23, 26]. Fekete, Lynch, and Shvartsman recently presented the *VS* specification for a partitionable group communication service. Chapter 5 of this thesis presents a short summary of the *VS* specification. For a more detailed description of *VS* and a comparison of *VS* with other specifications, the reader is referred to [13].

Several papers have since extended the *VS* specification. Chockler, Huleihel, and Dolev [8, 7] have used the same style to specify a virtually synchronous FIFO group communication service and to model an adaptive totally-ordered group communication service. De Prisco, Fekete, Lynch, and Shvartsman [25] have presented a specification for group communication service that provides a dynamic notion of a primary view.

1.2.2 Replication and Load Balancing

The most popular application of group communication services is for maintaining coherent replicated data through applying all operations in the same sequence at all replicas. The details of doing this in partitionable systems have been studied by Amir, Dolev, Friedman, Keidar, Melliar-Smith, Moser, and Vaysburd [18, 1, 2, 19, 16, 17].

In his recent book [4, p. 329], Birman points out that process groups are ideally suited for fault-tolerant load-balancing. He proposes two styles of load-balancing algorithms. In the first, more traditional, style, scheduling decisions are made by clients, and tasks are sent directly to the assigned servers. In the second style, tasks are multicast to all servers in the group; each server then applies a deterministic rule to decide on whether to accept each particular task.

In this thesis, we use a round-robin strategy originally suggested by Birman [4, p. 329]. In accordance with this strategy, tasks are sent to the servers using a totally-ordered multicast; the ith task delivered in a group of n servers is assigned to the server whose rank within this group is $(i \mod n)$. Since tasks are delivered to all members of the same group in the same order, each member is assigned the same number of tasks as any other member of its group. The load-balancing algorithm presented in this thesis extends this round-robin strategy with a *fail-over* policy that reissues query requests when group membership changes.

1.2.3 Sequential Consistency

There are many different ways in which a collection of replicas may provide the appearance of a single shared data object. The seminal work in defining these precisely is Lamport's notion of sequential consistency [20]. A system provides sequential consistency when for every execution of the system there is an execution with a single shared object that is indistinguishable to each individual client. A much stronger coherence property is *atomicity*, where a universal observer can't distinguish the execution of the system from one with a single shared object. The algorithm presented in this thesis provides an intermediate condition where the updates are atomic, but queries are sequentially consistent. Overall, the algorithm satisfies sequential consistency.

1.3 Our Contributions

This thesis presents a new distributed algorithm for providing replicated data on top of a partitionable group communication system, in which the work of processing queries is rotated among the group replicas in a round-robin fashion. While the replication part of the algorithm relates to the ideas of [18, 1, 2, 19] and the load-balancing strategy relates to the one in [4, p. 329], the novelty of the algorithm comes from the integration of these two parts. In particular, the algorithm supports sequentially consistent processing of queries in all, not just primary, components and guarantees that the servers of these components always have sufficiently advanced replicas in order to be able to process the queries assigned to them.

A major accomplishment of this work is in modeling and verifying the presented algorithm formally. This is done in the framework of the I/O automaton model for asynchronous computing [21] and includes the following components:

- An I/O automaton that specifies the allowed behavior of a sequentially consistent replicated service.
- An I/O automaton that models a distributed implementation of this specification as a composition of a servers' layer and a communication layer. The servers' layer consists of a number of identical state-machines, one for each server. The communication layer consists of a group communication service specification satisfying *VS* and of a collection of specifications for reliable reordering channels between any pair of servers.
- A hierarchical simulation proof that establishes that all traces of the implementation automaton are valid traces of the specification automaton. This proof relies on a number of high-level properties of the reachable states of the implementation automaton.
- An assertional proof of the high-level properties. The proof of these properties is based on an interesting approach: we invent a derived function $\mathcal{X}$ that expresses recursively the highest state reached by each server in each group. In a sense, this function presents a law according to which the replication part of the algorithm operates. As seen in Section 7.4, the recursive nature of this function makes proofs by induction easy: proving an inductive step simply involves unwinding a recursive step of the derived function $\mathcal{X}$.
- A proof that the load-balancing part of the algorithm is *uniform* and *non-blocking*. For uniformity, we show that each member of a group is assigned the same number of query requests as any other member of that group. For non-blockage, we show that the servers are always able to sufficiently advance the state of their replicas in order to process the queries assigned to them.

Another important contribution of this thesis is its support of *VS* as an adequate specification for a group communication service. A critical advancement of this work over the previous one [13] is that it explores some of the stronger properties of VS. Previous work [13] verified TO, an application in which all servers within a group process messages identically. In a sense, the TO application is anonymous, since a server uses its identity only to generate unique labels. While the proof in [13] uses the property of agreed message sequence, it does not account for the identical membership of message recipients. In contrast, the load-balancing part of our algorithm uses the fact that query recipients have the same idea of their membership when they decide which of them has to process a query.

Finally, the algorithm and the correctness proof presented in this work reveal several generic approaches that can be used to formally model other replication and load-balancing algorithms that are based on formally specified group communication services.

1.4 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 presents an overview of the I/O automaton model of Lynch and Tuttle [21] and introduces basic mathematical notation used in the thesis. Chapter 3 presents an I/O automaton that specifies the allowed behavior of the replicated service from the point of view of its blocking clients. Chapter 4 presents an intermediate specification for the service, the purpose of which is to simplify the proof of correctness. Chapter 5 presents an I/O automaton for the service implementation. Chapters 6 and 7 contain a proof that the implementation automaton implements the specification automaton in the sense of trace inclusion. Chapter 8 argues that the load-balancing strategy is uniform and non-blocking.

Chapter 2

Presentation Formalism

In this chapter, we summarize the I/O automaton model (without fairness) of Lynch and Tuttle [21], and present the mathematical notation used in this thesis.

2.1 The I/O Automaton Model

The following summary of the I/O automaton model for asynchronous computing is based on Chapter 8 of [22, pages 199-234]. We present only those aspects of the model that are used in this thesis.

2.1.1 I/O Automata

An I/O automaton models a distributed system component that can interact with other system components. It is a simple state machine in which the transitions are associated with named *actions*, each one of which is classified as either *input*, *output*, or *internal*. The inputs and outputs are used for communication with the automaton's environment and are externally visible; The internal actions are visible only to the automaton itself. The input actions are assumed not to be under the automaton's control — they arrive from the outside—while the automaton itself specifies what output and internal actions should be performed. The output and internal actions of the automaton are said to be *locally controlled*.

An I/O automaton is defined by the following four components: *signature* (input, output and internal actions), set of *states*, set of *start states* (a nonempty subset of states), and a *state-transition relation* (a subset of the cross-product of states, actions, and states). Note that the definition in [21, 22] has a fifth component, *tasks*, which is used to define fairness conditions on an execution of the automaton. We do not use this component because our current work deals only with the safety properties of the presented automata.

We call an element (s, ψ, s') of the state-transition relation a *transition*, or a step. If for a particular state s and action ψ, the automaton has some transition of the form (s, ψ, s'), then we say that ψ is *enabled*; States s and s' are respectively called *prestate* and *poststate*. A step (s, ψ, s') such that s' equals s is called an *empty* transition and is denoted by λ.

I/O automata are said to be *input-enabled*, since the input actions are not under the automatons control, and hence, are always enabled.

2.1.2 Executions and Traces

An *execution fragment* of an I/O automaton A is either a finite sequence, $s_0, \psi_1, s_2, \psi_2, \ldots, \psi_r, s_r$, or an infinite sequence, $s_0, \psi_1, s_2, \psi_2, \ldots, \psi_r, s_r, \ldots$, of alternating states and actions of A such that $(s_k, \psi_{k+1}, s_{k+1})$ is a transition of A for every $k \geq 0$. If the sequence is finite, it must end with a state. An execution fragment beginning with a start state is called an *execution*. 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 is a subsequence of α consisting of all external actions. We say that β is a trace of A if β is a trace of an execution of A.

2.1.3 Operations on Automata

Composition

The composition operation allows an automaton representing a complex system to be constructed by composing automata representing individual system components. The composition identifies actions with the same name in different component automata. When any component automaton performs a step involving ψ, so do all component automata that have ψ in their signatures.

The composition operation is restricted to *compatible* automata, which for our purposes have to satisfy the following two conditions:

1. The internal actions of each automaton have to be disjoint from all actions of the other automata. This condition is necessary to keep the internal actions of one automaton unobservable by other automata.
2. The output actions of all automata have to be disjoint. This condition is necessary to have at most one automaton control any given action.

In addition, the definition of compatibility in [22, pages 207-211] requires that each action be an action of only finitely many of the component automata. We do not use this condition because the compositions in this thesis contain only finite number of components.

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.

The states and start states of the composition automaton are vectors of states and start states, respectively, of the component automata. The transition of the composition are obtained by allowing all the component automata that have a particular action ψ in their signature to participate simultaneously in steps involving ψ, while all the other component automata do nothing.

In order to prove properties of a composed system of automata, it is often helpful to reason about the component automata individually. Section 8.5.4 of [22] shows that the composition operation has the nice properties that we expect it to have.

The composition of a countable, compatible collection of I/O automata $\{A_i\}_{i \in I}$ is denoted by $\prod_{i \in I} A_i$. If I is a finite set, we sometimes use the infix operator symbol $\times$ to denote the composition. E.g., if $I = \{1, \ldots, n\}$, $A_1, \times \cdots \times A_n$ denotes $\prod_{i \in I} A_i$.

Hiding

The *hiding* operation reclassifies the specified output actions of an automaton as internal. This prevents them from being used for further communication and means that they are no longer included in traces.

2.2 Properties and Proof Methods

In this section we describe the main techniques used to prove correctness of I/O automata: invariant assertions and hierarchical proofs. The material in this section is closely based on [22, pages 216-228].

2.2.1 Invariants

The most fundamental type of property to be proved about an automaton is an *invariant assertion*, or just *invariant*, for short. An invariant assertion of an automaton A is defined as any property that is true in every single reachable state of A.

Invariants are typically proved by induction on the number of steps in an execution leading to the state in question. While proving an inductive step, we consider only *critical actions*, which affect the state variables appearing in the invariant. In this thesis, there are also several invariants that are proved by inductive arguments that do not rely directly on the length of the execution sequence.

2.2.2 Hierarchical Proofs

One of the important proof strategies is based on a hierarchy of automata. This hierarchy represents a series of descriptions of a system or algorithm, at different levels of abstraction. The process of moving through the series of abstractions, from the highest level to the lowest level, is known as *successive refinement*. The top level may be nothing more than a problem specification written in the form of an automaton. The next level is typically a very abstract representation of the system: it may be centralized rather than distributed, or have actions with large granularity, or have simple but inefficient data structures. Lower levels in the hierarchy look more and more like the actual system or algorithm that will be used in practice: they may be more distributed, have actions with small granularity, and contain optimizations. Because of all this extra detail, lower levels in the hierarchy are usually harder to understand than the higher levels. The best way to prove properties of the lower-level automata is by relating these automata to automata at higher levels in the hierarchy, rather than by carrying out direct proofs from scratch.

The simplest way to relate two automata, say D and S, is to present a *refinement mapping* $DS()$ from the reachable states of D to the reachable state of S such that it satisfies the following two conditions:

1. If d is an initial state of D, then $DS(d)$ is an initial state of S.
2. If d and $DS(d)$ are reachable states of D and S respectively, and (d, σ, d') is a step of D, then there exists an execution fragment of S beginning at state $DS(d)$ and ending at state $DS(d)'$, with its trace being the same as the trace of σ and its final state $DS(d)'$ being the same as $DS(d')$.

The first condition, or *initial condition*, asserts that any initial state of D has some corresponding initial state of S. The second condition, or *step condition*, asserts that any step of D has a corresponding sequence of steps of S. This corresponding sequence can consist of one step, many steps, or even no steps, as long as the correspondence between the states is preserved and the external behavior is the same.

The following theorem gives the key property of refinement mappings:

Theorem 2.1 *If there is a refinement mapping from D to S, then*

$$traces(D) \subseteq traces(S).$$

If automata D and S have the same external signature and traces of D are traces of S, then we say that D *implements* S *in the sense of trace inclusion*, which means that D never does anything that S couldn't do. Theorem 2.1 implies that, in order to prove that one automaton implements another in the sense of trace inclusion, it is enough to produce a refinement mapping from the former to the latter.

Proving that one automaton implements another in the sense of trace inclusion constitutes only *partial correctness*, as it implies *safety* but not *liveness*. In other words, partial correctness ensures than "bad" things never happen, but it does not say anything whether some "good" thing eventually happens.

In this thesis, we concentrate on proving the partial correctness of our implementation. In addition, we prove a liveness-related claim that servers are always enabled to advance their replica states sufficiently far to be able to process the queries assigned to them. Future work will consider liveness properties, such as performance and fault-tolerance, stated conditionally to hold in periods of good behavior of the underlying network.

2.3 Mathematical Foundation and Notation

We use standard notation on sets, functions, and sequences, with the following specifics.

2.3.1 Sets and Functions

If A is a set, then $\mathcal{P}()$ denotes the power set of A, i.e., the set consisting of all the subsets of A: $\{S \mid S \subseteq A\}$.

If A and B are two sets, then $A \times B$ denotes the set $\{\langle a, b\rangle \mid a \in A \wedge b \in B\}$.

Total functions are denoted by "$\rightarrow$" and partial functions are denoted by "$\hookrightarrow$". If $f : A \rightarrow B$ then the domain of f, denoted $dom(f)$, is the entire set A; so for any $a \in A$, $f(a)$ is an element of B. If $g : A \hookrightarrow B$ then the domain of g is defined as a set $\{a \mid \exists\, b \,.\, \langle a, b\rangle \in g\}$; so for any $a \in A$, $f(a)$ is an element of B if $a \in dom(f)$, or is $\perp$ otherwise.

Given a total or a partial function f from A to B and elements $a \in A$ and $b \in B$, $f[a : b]$ denotes a function that is the same as f except it maps a to b. Sometimes we treat functions as sets of elements, where each element is a pair of an abscissa and ordinate.

Given a set A and some condition, $con(a)$, on its elements, the set $A|_{con(a)}$ denotes a subset of A that consists solely of elements that satisfy this condition:

$$A|_{con(a)} = \{a \mid a \in A \ \wedge \ con(a)\}.$$

2.3.2 Disjoint Unions

Somewhat non-standard is our use of *disjoint unions* (+), which differs from the usual set union ($\cup$) in that each element is *implicitly* tagged with what component it comes from. For simplicity, we use variable name conventions to avoid more formal "injection functions" and "matching constructs." Thus, for example, if *Update* and *Query* are the respective types for update and query requests, then type $Request = Update + Query$ defines a general request type. Furthermore, if $req \in Request$, and u and q are the established variable conventions for *Update* and *Query* types, then "$req \leftarrow u$" and "$req = q$" are both valid statements, denoting an assignment statement and an equality statement respectively.

2.3.3 Sequences

If x is a sequence then $|x|$ denotes the length of x. Sequences of zero length are denoted by []. If x is a sequence and $1 \leq i \leq j \leq |x|$, then $x[i]$ is the ith element of x, $x[i..j]$ is the subsequence $x[i], \ldots, x[j]$ of x, and $[i..]$ is the suffix of x starting at the ith element. Indexing of sequences starts either from 0 or 1. If x is an instance of the former, then the first element is $x[0]$, and the last element is $x[|x|-1]$; if x is an instance of the latter, then the first element is $x[1]$, and the last element is $x[|x|]$. If x and y are sequences, then $x + y$ is the concatenation of them (we sometimes abuse this notation by letting x or y be a single element). Notation $x \leq y$ expresses the fact that x is a prefix of y. Two sequences are *consistent*, denoted by $x \leq\geq y$, if one is a prefix of another. We use $x <_{||} y$ and $x \leq_{||} y$ to denote the length-wise comparison of x and y.

We use the dot notation to project sequences of tuples on their individual elements. For example, if z is a sequence $\langle x_1, \ y_1\rangle, \ldots, \langle x_n, \ y_n\rangle$, then $z.x$ denotes the sequence $x_1, \ldots, x_n$.

If z is a sequence each element of which is an element of a disjoint union, we denote the subsequence of z consisting solely of the elements that belong to the same basic type by subscripting z with a conventional symbol for that type. For example, if z is a sequence each element of which is of type (*Update* + *Query*), then z_u denotes the subsequence of z that consists solely of elements of *Update*.

2.3.4 Helpful Functions

Given two partial functions, $f : X \hookrightarrow Y$ and $g : X \hookrightarrow Y$, function $overlay(f, g) : X \hookrightarrow Y$ is defined as g over $dom(g)$ and as f over $(dom(f) - dom(g))$.

If a is an element of a totally ordered set A, then $rank(a, A)$ is defined to be the number of elements that are smaller than a.

If x is a sequence "$f_1, f_2, \ldots, f_n$" with each element f_i being a function of the type $A \to A$, and if a is an element of A, then $apply(x, a)$ ="$f_1(a), f_2(a), \ldots, f_n(a)$", $compose(x)$ ="$(f_n \circ \ldots \circ f_2 \circ f_1)$", and $scan(x)$ ="$f_1, (f_2 \circ f_1), \ldots, (f_n \circ \ldots \circ f_2 \circ f_1)$".

2.3.5 Notation

To access components of compound objects we use the *dot* notation. Thus, if *dbs* is a state variable of an automaton, then its instance in a state s is expressed as $s.dbs$. Likewise, if *view* is a state variable of a server p, then its instance in a state t is expressed as $t[p].view$; When the state is clear from the discussion, we write $p.view$.

We describe the transition relation in a *precondition-effect* style (as in [22]), which groups together all transitions that involve each particular type of action into a single atomic piece of code.

Chapter 3

Service Specification S

In this chapter, we present an I/O automaton S that specifies the allowed behavior of the replicated data service from the clients' point of view. Automaton S operates under the assumption that its clients are blocking. Being input-enabled, it cannot restrict its traces to those exhibited only by blocking clients. In order to get an automaton with well-formed traces, we close S by composing it with an automaton C that models a collection of nondeterministic blocking clients.

3.1 Type Information

The complete information on basic and derived types, along with a convention for variable usage, is given in Figure 3.1.

Figure 3.1 Type information

Variable	Type	Description
c	C	Finite set of clients.
db	DB	Database type with a distinguished initial value db_0.
a	$Answer$	Answer type for queries. (Answers for updates are $\{ok\}$.)
u	$Update : DB \to DB$	Update requests.
q	$Query : DB \to Answer$	Query requests.
r	$Request = Update + Query$	*Request* is a disjoint union of *Update* and *Query* types.
o	$Output = Answer + \{ok\}$	*Output* is a disjoint union of *Answer* and $\{ok\}$ types.

3.2 I/O Automaton S

The entire code for the I/O automaton S appears in Figure 3.2.

Figure 3.2 Specification S

Signature:

Input:
`request`$(r)_c, r \in \mathit{Request}, c \in C$
Output:
`reply`$(o)_c, o \in \mathit{Output}, c \in C$

Internal:
`update`$(c, u), c \in C, u \in \mathit{Update}$
`query`$(c, q, l), c \in C, q \in \mathit{Query}, l \in \mathcal{N}$

State:

$\mathit{dbs} \in \mathit{SEQ}0\ \mathit{DB}$, initially db_0.	Sequence of database states. Indexing starts from 0.
$\mathit{last} \in C \to \mathcal{N}$, initially $\{* \to 0\}$.	Index of the last db state witnessed by c.
$\mathit{map} \in C \hookrightarrow (\mathit{Request} + \mathit{Output})$, initially $\perp$.	Buffer for the clients' pending requests or replies.

Transitions:

`request`$(r)_c$
 Eff: $\mathit{map}(c) \leftarrow r$

`reply`$(o)_c$
 Pre: $\mathit{map}(c) = o$
 Eff: $\mathit{map}(c) \leftarrow \perp$

`update`(c, u)
 Pre: $u = \mathit{map}(c)$
 Eff: $\mathit{dbs} \leftarrow \mathit{dbs} + u(\mathit{dbs}[|\mathit{dbs}| - 1])$
  $\mathit{map}(c) \leftarrow \mathit{ok}$
  $\mathit{last}(c) \leftarrow |\mathit{dbs}| - 1$

`query`(c, q, l)
 Pre: $q = \mathit{map}(c)$
  $\mathit{last}(c) \leq l \leq |\mathit{dbs}| - 1$
 Eff: $\mathit{map}(c) \leftarrow q(\mathit{dbs}[l])$
  $\mathit{last}(c) \leftarrow l$

Signature

The interface between the service and its blocking clients is typical of a client-server architecture: Clients' requests are delivered to S via input actions of the form `request`$(r)_c$; S replies to its clients via actions of the form `reply`$(o)_c$.

Submitted requests are processed by internal actions of the form `update`(c, u) and `query`(c, q, l). The former represents processing of an update request u submitted by client c. The latter represents processing of a query request q submitted by client c with respect to the lth database state.

State Variables

If our service were to satisfy *atomicity* (i.e., behave as a non-replicated service), then specification S would include a state variable db of type DB and would process each update and query request with respect to the latest value of this variable. Since, in the actual distributed setting, servers of non-primary components may have outdated database values, satisfying atomicity would restrict processing of queries to the primary components of the system.

In order to eliminate this restriction and increase availability of the service, we give a slightly weaker specification that requires each query to be processed with respect to a value that is only at least as advanced as the last value witnessed by the query's client, not necessarily the latest one. For this purpose, S maintains a history dbs of database states and keeps an index $last(c)$ to the latest state witnessed by each client c. Please note that indexing of the dbs sequence starts from zero, which places the latest database state at the index $|dbs| - 1$.

In addition to dbs and $last$, there is a third state variable, map, that associates each client c with the status of its current request, if there is such. This status can be either the request itself if it has not been processed yet, or an output value if the request was processed but the output value has not been delivered to the client yet.

In the initial state, the variables have the following values. Sequence dbs contains only a single element db_0. The value of $last(c)$ for each client c is 0, as it points to the initial database db_0 in dbs. The map is empty since at this time there are no submitted requests.

Transitions

When a client c submits a request r via action $\texttt{request}(r)_c$, automaton S adds to the partial function map an association between c and r.

If the request r is an update request, u, then this association enables an internal action $\texttt{update}(c, u)$. When this action is executed, automaton S applies update request u to the latest database state $dbs[|dbs| - 1]$ and appends the resultant database state to the dbs sequence. As another two consequences of this action, automaton S reassociates c with ok in map, and sets $last$ to point to the last element of dbs.

Otherwise, if the request r is a query request, q, then the association of c with r in map enables the following internal actions: $\{\texttt{query}(c, q, l) \mid last(c) \leq l \leq |dbs| - 1\}$. The condition on l ensures that query request q will be processed with respect to a state that is at least as advanced as the last one witnessed by client c. When one of these enabled actions is executed, automaton S applies query request q to the lth database state, reassociates c in *map* with the answer to this query, and sets $last(c)$ to point to l, which now represents the last database state witnessed by client c.

After a request r is processed by one of the internal actions, the entry in *map* associated with its client c points to an output value. This association enables an output action $\texttt{reply}(o)_c$. When this action is executed, automaton S removes c from *map*.

3.3 Sequential Consistency

Even though our service is not atomic, it still appears to each particular client as a non-replicated one, and thus, satisfies *sequential consistency.* Note that, since the atomicity has been relaxed only for queries, the service is actually stronger than the weakest one allowed by sequential consistency.

3.4 Client Specification C

Automaton S operates under the assumption that its clients do not submit subsequent requests before they receive replies for the earlier submitted requests. Figure 3.3 presents an automaton C_c for such a well-formed client c.

Figure 3.3 Specification C_c for a nondeterministic blocking client c

Signature:

Input:	Output:
$\texttt{reply}(o)_c, o \in Output$	$\texttt{request}(r)_c, r \in Request$

State:
$busy \in Bool$, initially $false$. A status flag that keep track of whether there is a pending request.

Transitions:

$\texttt{request}(r)_c$ Pre: $busy = false$ Eff: $busy \leftarrow true$	$\texttt{reply}(o)_c$ Eff: $busy \leftarrow false$

The state of this automaton is a single boolean variable *busy*, initially equal to *false*. This variable reflects whether the client is currently awaiting a reply for a submitted earlier request.

Whenever *busy* is *false*, the automaton is enabled to submit an arbitrary request r via an output action $\texttt{request}(r)_c$. When it does so, it sets *busy* to *true*, which blocks any subsequent submissions until a reply is received. When an input action $\texttt{reply}(o)_c$ occurs, the automaton resets *busy* to *false*, thus enabling further request submissions.

Automaton C_c is nondeterministic in a sense that it submits arbitrary requests at arbitrary times. A real blocking client can be shown to implement such a nondeterministic one.

Traces of the automaton C_c are alternating sequences of $\texttt{request}(r)_c$ and $\texttt{reply}(o)_c$ actions that begin with $\texttt{request}(r)_c$.

3.5 Closed Automaton $\overline{S}$

Being input-enabled, automaton S allows for `request` actions to occur at any time, possibly deviating from the assumed pattern. In order to constrain the allowed executions of automaton S to follow the assumed pattern, we form a closed automaton $\overline{S}$ by composing S with the automata for the nondeterministic blocking clients.

$$\overline{S} = S \times \prod_{c \in C}(C_c).$$

In the rest of the thesis, when we present other automata, we will deal with their closed versions and will denote them with a bar (as in $\overline{S}$).

Invariant 3.1 *For each client $c \in C$, $(c.busy = false)$ if and only if $(map(c) = \bot)$.*

Proof 3.1: A proof by induction is straightforward: As far as the basis, the invariant is true in the initial state because $c.busy = false$ and $map(c) = \bot$. As far as the inductive step, we observe the following. The values of the two sides of the proposition both remain the same as their pre-state values when `update` and `query` actions take place, and they both reverse their pre-state values when `request` and `reply` actions take place. From this observation, it follows that the proposition is true in the post-state, assuming it is true in the pre-state. ■

Chapter 4

Intermediate Specification D

In this chapter we introduce an intermediate specification D and prove that automaton $\overline{D}$ implements automaton $\overline{S}$ in the sense of trace inclusion. Later, when we present an implementation automaton T, we prove the same result about automata $\overline{T}$ and $\overline{D}$, which by transitivity of the "implements" relation implies that automaton $\overline{T}$ implements automaton $\overline{S}$ in the sense of trace inclusion.

4.1 Motivation

Action `update` of specification S accomplishes two logical tasks: It updates the centralized database, and it sets client-specific variables, $map(c)$ and $last(c)$, to their new values. In a distributed setting, these two tasks are generally accomplished by two separate transitions. To simplify the refinement mapping between the implementation and the specification, we introduce an intermediate specification D, in which these two tasks are separated.

4.2 I/O Automaton D

Automaton D is formed by splitting each `update` action of S into two, `update` and `service`. The first one extends dbs with a new database state, but instead of setting $map(c)$ to "ok" and $last(c)$ to its new value as in S, it saves this value (i.e., the index to the most recent database state witnessed by c) in *delay* buffer. The second action sets $map(c)$ to "*ok*" and uses information stored in *delay* to set $last(c)$ to its value. The code for automaton D appears in Figure 4.1.

Figure 4.1 Intermediate Specification D

Signature: Same as in S, with the addition of an internal action $\texttt{service}(c), c \in C$.
State: Same as in S, with the addition of a state variable $delay \in C \hookrightarrow \mathcal{N}$, initially $\perp$.
Transitions: Same as in S, except `update` is modified and `service` is defined.

$\texttt{update}(c, u)$
Pre: $u = map(c)$
$c \notin dom(delay)$
Eff: $dbs \leftarrow dbs + u(dbs[|dbs| - 1])$
$delay(c) \leftarrow |dbs| - 1$

$\texttt{service}(c)$
Pre: $c \in dom(delay)$
Eff: $map(c) \leftarrow ok$
$last(c) \leftarrow delay(c)$
$delay(c) \leftarrow \perp$

4.3 Correctness of D

In this section, we prove that the closed automaton $\overline{D}$ implements the closed automaton $\overline{S}$ in the sense of trace inclusion. First, we study the invariants of $\overline{D}$ needed for the correctness proof. Then, we present a mapping between the reachable states of $\overline{D}$ and $\overline{S}$ and prove that this mapping is a refinement, which implies the correctness result.

4.3.1 Invariants on $\overline{D}$

Invariant 4.1 *For each client $c \in C$, each of the following propositions is true.*

1. $(c.busy = false) \Rightarrow (map(c) = \perp)$
2. $(map(c) = \perp) \Rightarrow (c.busy = false)$
3. $(c.busy = false) \Rightarrow (delay(c) = \perp)$

Proof 4.1: This multipart invariant can be proved by induction on the length of the execution sequence similarly to the proof of Invariant 3.1 in Chapter 3. ■

Invariant 4.2 *For each client $c \in C$, if $delay(c) \neq \perp$ then $map(c) \in Update$.*

Proof 4.2: A proof by induction on the length of the execution sequence is straightforward. The parts of the inductive step that involve actions `request` and `service` rely on Invariant 4.1. ■

4.3.2 Refinement Mapping $\boldsymbol{DS} : \overline{D} \rightarrow \overline{S}$

We want to construct a mapping $DS()$ that maps each reachable state of $\overline{D}$ to a reachable state of $\overline{S}$ and satisfies the following two properties:

1. If d is an initial state of $\overline{D}$, then $DS(d)$ is an initial state of $\overline{S}$.
2. If d and $DS(d)$ are reachable states of $\overline{D}$ and $\overline{S}$ respectively, and (d, σ, d') is a step of $\overline{D}$, then there exists an execution fragment of $\overline{S}$ beginning at state $DS(d)$ and ending at state $DS(d)'$, with its trace being the same as σ and its final state $DS(d)'$ being the same as $DS(d')$.

A state of $\overline{S}$ consists of the following components: *dbs*, *map*, *last*, and *c.busy* for all $c \in C$. The refinement mapping $DS()$ has to specify how these components are constructed from a reachable state d of $\overline{D}$ in a way that preserves the two properties above.

The difference between automata $\overline{D}$ and $\overline{S}$ is that $\overline{D}$ delays the propagation of new values in to *map* and *last* by temporarily storying them in *delay*. Thus, the entries in *delay* are always more up-to-date than the corresponding entries in *map* and *last*. In automaton $\overline{S}$, on the other hand, the values in *map* and *last* are always up-to-date.

Definition: Given two partial functions $f, g : X \hookrightarrow Y$, we define a partial function $overlay(f, g) : X \hookrightarrow Y$ to be as g over $dom(g)$ and as f over $(dom(f) - dom(g))$.

Lemma 4.1 *The following function $DS()$ is a* refinement *from automaton $\overline{D}$ to automaton $\overline{S}$ with respect to the reachable states of $\overline{D}$ and $\overline{S}$.*

$$
\begin{array}{llcl}
DS(d : \overline{D}) & \rightarrow \overline{S} & = & s, \ where \\
 & s.dbs & = & d.dbs \\
 & s.map & = & overlay(d.map, \{\langle c, \ ok \rangle \mid c \in dom(d.delay)\}) \\
 & s.last & = & overlay(d.last, d.delay) \\
 & s[c].busy & = & d[c].busy \quad for \ all \ c \in C
\end{array}
$$

Proof 4.1:

1. *Basis:* We want to show that the initial state d_0 of $\overline{D}$ maps to the initial state s_0 of $\overline{S}$. Consider the initial state d_0. It is straightforward to see that $DS(d_0).dbs = s_0.dbs$ and $DS(d_0)[c].dbs = s_0[c].dbs$ for all $c \in C$. For the remaining two state variables, notice that $d_0.delay = \perp$ implies that $DS(d_0).map = d_0.map$ and $DS(d_0).last = d_0.last$. Since *map* and *last* are the same in the initial states of $\overline{D}$ and $\overline{S}$, it follows that $DS(d_0).map = s_0.map$ and $DS(d_0).last = s_0.last$. Thus $DS(d_0) = s_0$.

2. *Inductive Step:* Assume that d and $DS(d)$ are reachable states of $\overline{D}$ and $\overline{S}$ respectively, and that (d, σ, d') is a step of $\overline{D}$. We show that, for all σ, the step (d, σ, d') of $\overline{D}$ simulates the step $(DS(d), \psi, DS(d)')$ of $\overline{S}$ with $\psi = \sigma$ and $DS(d') = DS(d)'$, except when σ is a `service` action, then the step (d, σ, d') of $\overline{D}$ simulates an empty transition of $\overline{S}$.

Even though the code for actions `request`, `reply`, and `query` is identical in $\overline{S}$ and $\overline{D}$, the preservation of the refinement mapping for the transitions involving these actions is not trivial because state variables $d.map$ and $d.last$ do not map directly to their counterparts $s.map$ and $s.last$ in $\overline{S}$.

Consider a transition (d, σ, d') of $\overline{D}$ that involves any one of these three actions and assume it is enabled in state d. Then, $d.map(c) \notin Update$ in the pre-state (for `request` use Invariant 4.1.1). By Invariant 4.2, $d.delay(c)$ is undefined, and thus, $c \notin domain(d.delay)$. From this and the fact that $d'.delay = d.delay$ it is straightforward to show that the refinement mapping is preserved.

The two remaining actions of $\overline{D}$ are `update` and `service`. Let's examine them closer:

$\sigma =$ `update`(c, u) — A transition of $\overline{D}$ with this action corresponds to a transition of $\overline{S}$ with the same action. The precondition on this action implies that $c \notin dom(d.delay)$, and thus $DS(d).map(c) = u$. This means that the corresponding action of S, `update`(c), is enabled. It is straightforward to see that the refinement holds for state variables dbs and $c.busy$. For map and $last$ we use the fact that $dom(d'.delay) = dom(d.delay) \cup \{c\}$.

$\sigma =$ `service`(c) — A transition of $\overline{D}$ with this action corresponds to an empty transition of $\overline{S}$. We have to show that $DS(d) = DS(d')$. The precondition on σ implies that $c \in dom(d.delay)$. This means that $DS(d).map(c) = ok$ and $DS(d).last(c) = d.delay(c)$. In the post-state $DS(d')$, these variables are the same because even though c is no longer in $dom(d'.delay)$, $DS(d').map(c)$ and $DS(d').last(c)$ have been set respectively to ok and $d.delay(c)$ by σ. Finally, $d'.dbs = d.dbs$, and for all $c \in C$, $d'[c].busy = d[c].busy$ since these state variables are not affected by σ. ∎

Theorem 4.2 *Automaton $\overline{D}$ implements automaton $\overline{S}$ in the sense of trace inclusion.*

Proof : Follows immediately from Lemma 4.1. ∎

Chapter 5

Service Implementation T

In this chapter, we present an I/O automaton T that models a distributed implementation of the replicated data service specified by the automaton S of Chapter 3. The proof of correctness that establishes T as an implementation of S appears in the next two chapters.

5.1 Architectural Structure of T

The distributed implementation of the service consists of the servers' layer and the communication layer. The servers' layer is symmetric: all servers run identical state-machines. The communication layer consists of two parts, a group communication service satisfying VS, and a collection of individual channels providing reliable reordering point-to-point communication between all pairs of servers.

The servers use the group communication service to disseminate update and query requests to the members of their groups and rely on the properties of this service to enforce the formation of identical sequences of update requests at all servers and to schedule query requests correctly. The point-to-point channels are used to send the results of processed queries directly to the original servers.

Figure 5.0 below depicts the major components of the system and their interactions. Set P represents the set of servers. Each server $p \in P$ runs an identical state-machine $VStoD_p$ and serves the clients whose $c.proc$ equals p.

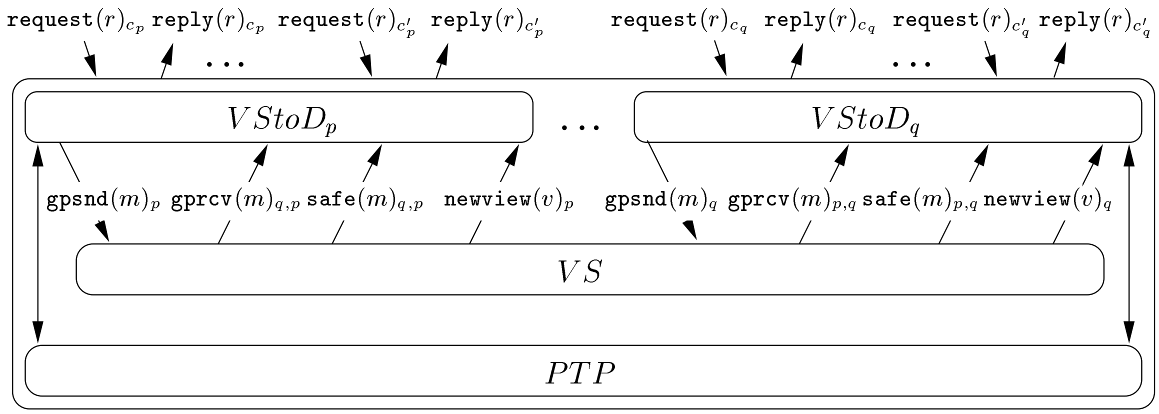

Figure 5.0 System components and their interactions.

The I/O automaton T for the service implementation is a composition of the servers' layer $I = \prod_{p \in P}(VStoD_p)$ with the group-communication service specification VS and a collection PTP of reliable reordering point-to-point channels between any pair of servers, with all the output actions of this composition hidden, except for the servers' replies.

$$T = hide_{out(I \times VS \times PTP) - \{\mathtt{reply}(o)_c\}}\big(I \times VS \times PTP\big).$$

5.2 Communication Layer

5.2.1 The *VS* Specification

The state-machine VS of [13, 14, without the performance/fault-tolerance property *VS-property*] is reprinted in Figure 5.1.

For the rest of the paper, we fix M to be a message alphabet for the group communication service, and $\langle G, <_G, g_0 \rangle$ to be a totally-ordered set of view identifiers with an initial view identifier. An element of the set $V = G \times \mathcal{P}(P)$ is called a *view*. If v is a view, we write $v.id$ and $v.set$ to denote its components.

Automaton VS specifies a partitionable service in which, at any moment of time, every client has precise knowledge of its current view. VS does not require clients to learn about every view of which they are members, nor does it place any consistency restrictions on the membership of concurrent views held by different clients. Its only

Figure 5.1 Automaton *VS*

Signature:

Input:
$\texttt{gpsnd}(m)_p$, $m \in M$, $p \in P$
Internal:
$\texttt{createview}(v)$, $v \in views$
$\texttt{vs-order}(m, p, g)$, $m \in M$, $p \in P$, $g \in G$

Output:
$\texttt{gprcv}(m)_{p,q}$ $m \in M$, $p \in P$, $q \in P$, hidden $g \in G$
$\texttt{safe}(m)_{p,q}$ $m \in M$, $p, q \in P$, hidden $v \in views$
$\texttt{newview}(v)_p$, $v \in views$, $p \in P$, $p \in v.set$

State:

$created \subseteq V$, initially $\{\langle g_0, P\rangle\}$
for each $p \in P$:
  $current\_viewid[p] \in G$, initially g_0
for each $g \in G$:
  $queue[g]$, a sequence of $M \times P$, initially empty

for each $p \in P$, $g \in G$:
  $pending[p, g]$, a sequence of M, initially empty
  $next[p, g] \in \mathcal{N}^{>0}$, initially 1
  $next\_safe[p, g] \in \mathcal{N}^{>0}$, initially 1

Transitions:

$\texttt{createview}(v)$
 Pre: $v.id > max(g : \exists S, \langle g, S\rangle \in created)$
 Eff: $created \leftarrow created \cup \{v\}$

$\texttt{newview}(v)_p$
 Pre: $v \in created$
   $v.id > current\_viewid[p]$
 Eff: $current\_viewid[p] \leftarrow v.id$

$\texttt{gpsnd}(m)_p$
 Eff: append m to $pending[p, current\_viewid[p]]$

$\texttt{vs-order}(m, p, g)$
 Pre: m is head of $pending[p, g]$
 Eff: remove head of $pending[p, g]$
   append $\langle m, p\rangle$ to $queue[g]$

$\texttt{gprcv}(m)_{p,q}$, hidden g
 Pre: $g = current\_viewid[q]$
   $queue[g](next[q, g]) = \langle m, p\rangle$
 Eff: $next[q, g] \leftarrow next[q, g] + 1$

$\texttt{safe}(m)_{p,q}$, hidden g, S
 Pre: $g = current\_viewid[q]$
   $\langle g, S\rangle \in created$
   $queue[g](next\_safe[q, g]) = \langle m, p\rangle$
   for all $r \in S$:
    $next[r, g] > next\_safe[q, g]$
 Eff: $next\_safe[q, g] \leftarrow next\_safe[q, g] + 1$

view-related requirement is that views are presented to each client according to the total order on view identifiers. *VS* provides a multicast service that imposes a total order on messages submitted within each view and delivers these messages according to this order, with no omissions, and strictly within a view. In other words, the sequence of messages received by each client while in a certain view is a prefix of the total order on messages associated with that view. Separately from the multicast service, *VS* provides a "safe" notification once a message has been delivered to all members of the view.

5.2.2 The *PTP* Specification

The automaton PTP for the collection of reliable reordering point-to-point channels between all pairs of clients is a composition $\prod_{p,p' \in P}(L_{p,p'})$ of individual channels $L_{p,p'}$ for all $p, p' \in P$.

The code for the I/O automaton $L_{p,p'}$ appears in Figure 5.2; it is similar to the automaton A in [22, pages 460-461].

For the rest of the paper, we fix *Packet* to be a message alphabet for the point-to-point channels and denote by *pkt* an element of this alphabet.

Figure 5.2 Specification $L_{p,p'}$ for a reliable reordering channel from p to p'

Signature:
Inp: $\texttt{ptpsnd}(pkt)_{p,p'}, pkt \in Packet, p \in P, p' \in P$
Out: $\texttt{ptprcv}(pkt)_{p',p}, pkt \in Packet, p \in P, p' \in P$

State:
$\textit{in-transit}_{p,p'}$, a multiset of elements of *Packet*, initially $\emptyset$

Transitions:
$\texttt{ptpsnd}(pkt)_{p,p'}$
Eff: add pkt to $\textit{in-transit}_{p,p'}$

$\texttt{ptprcv}(pkt)_{p,p'}$
Pre: $pkt \in \textit{in-transit}_{p,p'}$
Eff: remove pkt from $\textit{in-transit}_{p,p'}$

5.3 Servers' Layer

The servers' layer is composed of $|P|$ identical state-machines, one for each server $p \in P$. We next describe an automaton $VStoD_p$ that models the state-machine of a server p.

5.3.1 Type Information

As an extension to Figure 3.1 on page 27, Figure 5.3 presents additional information on types and a convention for variable-name usage.

Figure 5.3 Additional type declaration

Variable	Type	Description
	P	Fixed set of servers.
	$\mathcal{Q} \subseteq \mathcal{P}(P)$	Fixed set of quorums. Notice: $P \in \mathcal{Q}$.
g	$\langle G, <_G, g_0 \rangle$	Totally-ordered set of view identifiers.
v	$V = G \times \mathcal{P}(P)$	Set of views.
x	$X = G \times (C \times Update)^* \times \mathcal{N}$	Expertise information for exchange process.
m	$M = C \times Update + C \times Query \times \mathcal{N} + X$	Messages sent via *VS*.
pkt	$Packet = C \times Answer \times \mathcal{N} \times G$	Packets sent via *PTP*.

For the rest of the paper, we fix a set $\mathcal{Q}$ of *quorums*, each of which is a subset of P. We assume that every pair Q, Q' in $\mathcal{Q}$ satisfies the *intersection property* $Q \cap Q' \neq \emptyset$.

The sets V and G of *VS* views and their identifiers are introduced on page 38.

There are three different types of messages that are transmitted via the group communication service: update, query, and expertise. An update message consists of an update requests paired with its client id. If m is an update message, then $m.u$ and $m.c$ denote its components. A query message is a triple consisting of a query request, its client id, and an index to the last database state witnessed by this client. If m is a query message, then $m.q$, $m.c$, and $m.l$ denote its components. An expertise message is a triple consisting of a view identifier, a sequence of update messages, and a natural number representing the length of a certain prefix of this sequence. If x is an expertise message, then $x.xl$, $x.us$, and $x.su$ respectively denote the just mentioned components. Expertise messages are used during expertise-exchange process, which is explained in the next section.

Packets transmitted via point-to-point channels carry information pertaining to the processed query requests. Each packet consists of four components: a client id, an answer to the client's query, an index to the database states with respect to which the query was processed, and a view identifier of a view in which the query was processed. If *pkt* is a packet, then *pkt.c*, *pkt.a*, *pkt.l*, and *pkt.g* respectively denote the just mentioned components.

5.3.2 I/O Automaton $\boldsymbol{VStoD_p}$

The I/O code for the $VStoD_p$ state machine is given in Figures 5.4 and 5.5.

Signature

The external actions of the automaton $VStoD_p$ represent its interface with the outside world, namely, with the group communication service, the point-to-point channels, and the clients. Thus, the external actions of $VStoD_p$ are exactly the external actions of *VS* projected on p, of $\prod_{p' \in P}(L_{p,p'} \times L_{p',p})$, and of $\prod_{\{c \in C \,|\, c.proc=p\}}(C_c)$.

The internal actions of the automaton $VStoD_p$ are labeled `update` and `query`. An instance of the former applies an update request u from a client c to the current database state. An instance of the latter applies a query request q from a client c to the current database state which is at least as advanced as the lth state.

Figure 5.4 Implementation $VStoD_p$: Signature and State Variables

Signature:

Input:

$\texttt{request}(r)_c, r \in Request, c \in C, c.proc = p$
$\texttt{gprcv}(m)_{p',p}, m \in M, p' \in P$
$\texttt{safe}(m)_{p',p}, m \in M, p' \in P$
$\texttt{newview}(v)_p, v \in V$
$\texttt{ptprcv}(pkt)_{p',p}, pkt \in Packet, p' \in P$

Output:

$\texttt{reply}(o)_c, o \in Output, c \in C, c.proc = p$
$\texttt{gpsnd}(m)_p, m \in M$
$\texttt{ptpsnd}(pkt)_{p,p'}, pkt \in Packet, p' \in P$

Internal:

$\texttt{update}(c, u), c \in C, u \in Update$
$\texttt{query}(c, q, l), c \in C, u \in Update$

State:

$db \in DB$, initially db_0.	Local database replica.
$last \in C\vert_{(c.proc=p)} \to \mathcal{N}$, initially $C\vert_{(c.proc=p)} \to 0$.	Index of the last db state seen by each client.
$map \in C\vert_{(c.proc=p)} \hookrightarrow Request + Output$, initially $\perp$.	Buffer for pending requests or replies.
$pending \in \mathcal{P}(C\vert_{(c.proc=p)})$, initially $\emptyset$.	Clients whose requests are being processed.
$updates \in (C \times Update)^*$, initially $[\,]$.	Sequence of update requests (indexing from 1).
$safe\_to\_update \in \mathcal{N}$, initially 0.	Index of the last safe element in $updates$.
$last\_update \in \mathcal{N}$, initially 0.	Index of the last executed element in $updates$.
$query\_counter \in \mathcal{N}$, initially 0.	Load-balancing counter.
$queries \in C \hookrightarrow (Query \times \mathcal{N}) + (Answer \times \mathcal{N})$, initially $\perp$.	Query requests paired with their $last(c)$ or query answers paired with their $last(c)$.
$view \in V$, initially $V_0 = \langle g_0, P \rangle$.	Current view of p.
$expertise\_level \in G$, initially g_0.	The highest primary view id known to p.
$expertise\_max \in X$, initially $\langle g_0, [\,], 0 \rangle$.	The highest expertise received so far.
$expert\_counter1 \in \mathcal{N}$, initially 0.	Number of expertise messages received so far.
$expert\_counter2 \in \mathcal{N}$, initially 0.	Number of expertise messages received safely.
$mode \in \{normal,\ expertise\_broadcast,\ expertise\_collection\}$, initially $normal$.	Modes: The first item is normal activity, the last two are recovery activity.

State Variables

State variable db models the server's local database replica. The initial state of this variable is db_0. Notice that $VStoD_p$ maintains only one database state, the current one. This is in contrast to the specifications S and D which maintain a history of database states.

State variables $last$ and map have the same purpose as in the specification automata: $last(c)$ is an index to the last database state witnessed by the client c of p; $map(c)$ is either undefined if there is no outstanding request from the client c, or $Request$ if $VStoD_p$ has not begun processing a submitted by client c request, or $Output$ if $VStoD_p$ has already obtained an output value for a request submitted by client c but has not delivered it to c yet. When $VStoD_p$ begins processing submitted by a client c request, it enters c in a set $pending$ until the output value for the request is obtained.

$VStoD_p$ maintains a sequence $updates$ of update requests paired with their client identifiers. The purpose of this sequence is to enforce the order in which update requests are applied to the local database replica *db*. The sequence has two distinguished prefixes $updates[1..safe\_to\_update]$ and $updates[1..last\_update]$, called *safe* and *done*, that mark respectively those update requests that have been determined as safe to execute and those that have been already executed.

A *query_counter* keeps track of the number of query requests delivered in the current view. Based on this number, the server decides whether or not to accept for processing each query request. Partial function *queries* associates clients, whose requests have been accepted, with either of the two types of pairs: A pair of the first type contains a query request and an index to a minimum database state with respect to which the query has to be processed. A pair of the second type contains an answer to a query and an index to a database state with respect to which this answer was obtained.

A *view* variable contains the current view of the server. A view can be either *primary* or not depending on whether the members of the view comprise a quorum. In order for the server to determine how advanced its state is compared to others, it keeps track of a view identifier of the latest primary view of which it has knowledge. This identifier is stored in the *expertise_level* variable.

The purpose of the *mode* variable is to enable/disable certain transitions depending on the current state of the automaton.

In certain automaton states, the server receives expertise messages from other servers of its view. The state variable *expertise_max* keeps track of the highest expertise received so far. The two counters, *expert_counter1* and *expert_counter2*, keep track respectively of how many expertise messages and of how many safe notifications for these messages have been delivered to the server in its current view.

Transitions

Transitions of $VStoD_p$ can be classified as either *front end*, *processing of query requests*, *processing of update requests*, or *recovery activity*.

Front end transitions involve actions of the form $\texttt{request}(r)_c$ and $\texttt{reply}(o)_c$, which result in the submitted requests being checked in to and the replied output values being checked out of the *map* buffer.

Figure 5.5 Implementation $VStoD_p$: Transitions

Transitions:

$\texttt{request}(r)_c$
  Eff: $map(c) \leftarrow r$

$\texttt{gpsnd}(c, q, l)_p$
  Pre: $mode = normal$
    $q = map(c) \ \wedge \ c \notin pending$
    $l = last(c)$
  Eff: $pending \leftarrow pending \ \cup \ c$

$\texttt{gprcv}(c, q, l)_{p',p}$
  Eff: $query\_counter \leftarrow query\_counter + 1$
    if $(\mathrm{rank}(p, view.set) =$
      $query\_counter \mod |view.set|)$
    then $queries(c) \leftarrow \langle q, \ l \rangle$

$\texttt{query}(c, q, l)$
  Pre: $\langle q, \ l \rangle \in queries(c)$
    $last\_update \geq l$
  Eff: $queries(c) \leftarrow \langle q(db), \ last\_update \rangle$

$\texttt{ptpsnd}(c, a, l, g)_{p,p'}$
  Pre: $c \in dom(queries) \ \wedge \ c.proc = p'$
    $\langle a, \ l \rangle \in queries(c)$
    $g = view.id$
  Eff: $queries(c) \leftarrow \perp$

$\texttt{ptprcv}(c, a, l, g)_{p',p}$
  Eff: **if** $(g = view.id \ \wedge \ c.proc = p)$ **then**
    $pending \leftarrow pending \ - \ c$
    $map(c) \leftarrow a$
    $last(c) \leftarrow l$

$\texttt{newview}(v)_p$
  Eff: $queries \leftarrow \perp$; $query\_counter \leftarrow 0$
    $pending \leftarrow pending \ - \ \{c \mid (\exists\, q \,.\, \langle c, \ q \rangle \in map)\}$
    $\boxed{safe\_to\_update \leftarrow safe\_to\_update + \Delta}$
    $expertise\_max \leftarrow expertise\_max_0$
    $expert\_counter1 \leftarrow 0$; $expert\_counter2 \leftarrow 0$
    $mode \leftarrow expertise\_broadcast$
    $view \leftarrow v$

$\texttt{gpsnd}(x)_p$
  Pre: $mode = expertise\_broadcast$
    $x = \langle expertise\_level, updates, safe\_to\_update \rangle$
  Eff: $mode \leftarrow expertise\_collection$

$\texttt{reply}(o)_c$
  Pre: $map(c) = o$
  Eff: $map(c) \leftarrow \perp$

$\texttt{gpsnd}(c, u)_p$
  Pre: $mode = normal \ \wedge \ view.set \in \mathcal{Q}$
    $u = map(c) \ \wedge \ c \notin pending$
  Eff: $pending \leftarrow pending \ \cup \ c$

$\texttt{gprcv}(c, u)_{p',p}$
  Eff: $updates \leftarrow updates + \langle c, \ u \rangle$

$\texttt{safe}(c, u)_{p',p}$
  Eff: $safe\_to\_update \leftarrow safe\_to\_update + 1$

$\texttt{update}(c, u)$
  Pre: $last\_update < safe\_to\_update$
    $\langle c, \ u \rangle = updates[last\_update + 1]$
  Eff: $last\_update \leftarrow last\_update + 1$
    $db \leftarrow u(db)$
    if $(c.proc = p)$ **then**
      $pending \leftarrow pending \ - \ c$
      $map(c) \leftarrow ok$
      $last(c) \leftarrow last\_update$

$\texttt{gprcv}(x)_{p',p}$
  Eff: $expertise\_max \leftarrow max_{\mathcal{X}}(expertise\_max, x)$
    $expert\_counter1 \leftarrow expert\_counter1 + 1$
    if $(expert\_counter1 = |view.set|)$ **then**
      $expertise\_level \leftarrow expertise\_max.xl$
      $updates \leftarrow expertise\_max.us$
      $safe\_to\_update \leftarrow expertise\_max.su$
      if $(view.set \in \mathcal{Q})$ **then**
        $expertise\_level \leftarrow view.id$

$\texttt{safe}(x)_{p',p}$
  Eff: $expert\_counter2 \leftarrow expert\_counter2 + 1$
    if $(expert\_counter2 = |view.set|)$ **then**
      if $(view.set \in \mathcal{Q})$ **then**
        $safe\_to\_update \leftarrow |expertise\_max.us|$
        $pending \leftarrow pending \ -$
          $\{c \mid c \in pending \wedge$
            $c \notin updates[(last\_update + 1) \,..\, safe\_to\_update].c\}$
      $mode \leftarrow normal$

The server initiates processing of the submitted requests by multicasting them to the members of its current view (including itself) using the `gpsnd` primitive of *VS*. This operation is limited to the times when the server's *mode* is *normal*, which corresponds to the server being in an *established* view.

When a view of the server changes, the server suspends processing of new requests by switching its *mode* away from *normal* and starts recovery activity. The goal of the recovery activity is to establish the server's new view so it may resume its *normal* mode of operation. Successful completion of this activity requires collaboration of all the servers of the new view.

We will now describe each of the different transition categories in more detail. Please refer to the corresponding code fragments in Figure 5.5.

Front End

Transitions involving input actions of the form $\texttt{request}(r)_c$ cause the server to extend its *map* buffer with the corresponding client/request associations. Presence of client/output-value associations in the server's *map* buffer enables transitions involving output actions of the form $\texttt{reply}(o)_c$.

The code models *map* buffer as a partial function because it asserts that, at any given time, for each client c, there can be at most one outstanding request. The validity of this assertion depends on the well-formedness of clients.

Processing of Query Requests

Processing of query requests is handled by actions of the form $\texttt{gpsnd}(c, q, l)_p$, $\texttt{gprcv}(c, q, l)_{p',p}$, $\texttt{query}(c, q, l)$, $\texttt{ptpsnd}(c, a, l, g)_{p,p'}$, $\texttt{ptprcv}(c, a, l, g)_{p',p}$, and $\texttt{newview}(v)$.

The server initiates processing of each submitted query request by multicasting it to the members of its view, using a $\texttt{gpsnd}(c, q, l)_p$ action. The last argument, l, is an index to the least database state with respect to which query q has to be processed in order to ensure sequential consistency. The value of l is taken as the value of $last(c)$.

When *VS* delivers a query request to the server using a $\texttt{gprcv}(c, q, l)_{p',p}$ action, the server checks if it is its turn to service this query, and if so, accepts the request. The fact that all members of the server's view receive query requests in the same order guarantees that the delivered in this view queries are scheduled uniformly among its members.

Accepted query requests are serviced by internal actions of the form $\texttt{query}(c, q, l)$, which is enabled only when the state of the server's database db is no smaller than the one specified by the request. This condition is captured by *last_update* being greater than or equal to l. The non-trivial guarantee of the algorithm is that the servers are always enabled to advance their database states sufficiently far to be able to process the accepted queries.

Answers for the serviced queries are forwarded to the queries' original servers using actions of the form $\mathtt{ptpsnd}(c, a, l, g)_{p,p'}$. The first argument, c, specifies the client of p' to whom answer a has to be relayed. The third argument, l, specifies an index of the database state with respect to which the answer was obtained. The last argument, g, specifies the current view of the server. It is included in the packet in order to ensure the within-view delivery to p'.

When a point-to-point channel $PTP_{p',p}$ delivers to the server a packet $\langle c, a, l, g\rangle$ carrying an answer to a query, the server accepts this answer only if the following two conditions are met. The first condition checks if the server's current view matches the one specified in the packet. This condition implements the within-view delivery of answers and is necessary in order to eliminate unexpected packet arrivals (see next paragraph). The second condition checks if c is among the server's clients. This condition is just a technical one: It always holds (as shown by Invariant 6.5 on page 54) but is included to ensure well-typing (as the domains of *map*, *pending*, and *last* are restricted to those cs whose $c.proc = p$).

When the server learns of its new view, it executes a simple query-related recovery procedure , in which it moves its own pending queries for reprocessing and erases any information pertaining to the queries of others.

Processing of Update Requests

Processing of update requests is handled by actions of the type $\mathtt{gpsnd}(c, u)_p$, $\mathtt{gprcv}(c, u)_{p',p}$, $\mathtt{safe}(c, u)_{p',p}$, and $\mathtt{update}(c, u)$.

The server initiates processing of each submitted update request by multicasting it to the members of its view, using a $\mathtt{gpsnd}(c, u)_p$ action. This action is allowed only when the current view of the server is a primary one.

When *VS* delivers an update request to the server using a $\mathtt{gprcv}(c, u)_{p',p}$ action, the server fixes a tentative position of this request by appending it to its *updates* sequence. The algorithm ensures that, when this request is delivered to other members of the service's view, it is assigned the same tentative position in the *updates* sequences of these members. This is necessary in order to maintain the *updates* sequences of different servers mutually consistent. Attainment of this behavior directly relies on the following four properties of the algorithm: First, servers start with the same, empty, *updates* sequences. Second, *updates* sequences of the members of the same primary (non-initial) view are the same when each is taken at the time when its server *establishes* this view. Third, when update requests are delivered to servers, the servers' views are already established. Finally, members of the same view receive update requests in the same order with no omissions. The second and third properties are established as a result of preceding recovery activity; The last one is guaranteed solely by *VS*.

When *VS* delivers a safe notification to the server using a $\texttt{safe}(c,u)_{p',p}$ action, the server extends the safe prefix of its *updates* sequence to cover an adjacent to the prefix request. The code asserts that the covered request is in fact the one for which the server has received the safe notification. The validity of this assertion rests on the following five properties of the algorithm: First, servers start with the same, empty, safe prefixes. Second, safe prefixes of all the members of the same primary (non-initial) view are the same when each is taken at the time when its server switches to *normal* mode. Third, when each server switches to *normal* mode, the only unsafe requests on its *updates* sequence are those that it has received in its current view. Fourth, when safe notifications for update requests are delivered to servers, the servers' modes are already *normal*. Finally, safe notifications arrive in the same order with no omissions as the delivered requests. The second, third, and fourth properties are established as a result of preceding recovery activity; The last one is guaranteed solely by *VS*.

Update requests that are covered by the server's safe prefix are applied to the server's database replica by internal actions of the form $\texttt{update}(c,u)$, in the order specified by the prefix. If the applied request happens to be *native*, i.e., belong to one of the server's clients, then the server also removes c from *pending*, sets $map(c)$ to *ok* and $last(c)$ to *last_update* (which is the index of the current database state).

Notice that a simple operation of extending a safe prefix to cover an adjacent to the prefix request has crucial implications: It allows for the covered request to be applied to the server's database (once all the preceding, say $i-1$, requests in the prefix are applied), and as a consequence, requires that the ith update request applied by any server to its database be this particular request (or otherwise mutual consistency of data replicas is violated). The ability of the algorithm to allow these implications and still guarantee correctness rests in large on the way recovery activity is organized.

Recovery Activity

The server's recovery activity is initiated when the server is informed of its new view, and is handled by actions of the form $\texttt{newview}(v)_p$, $\texttt{gpsnd}(x)_p$, $\texttt{gprcv}(x)_{p',p}$, and $\texttt{safe}(x)_{p',p}$.

When *VS* informs the server of its new view using a $\texttt{newview}(v)_p$ action, the server performs the following three tasks: First, it executes a small query-related recovery procedure described on page 46. Second, it adjusts its *safe_to_update* index to be at least as advanced as each of the *last* indices witnessed by its clients; This is discussed in more detail on page 49. Third, it enters an *expertise-exchange* procedure that constitutes the rest of the recovery activity.

During an expertise exchange procedure, servers exchange their *expertise*, which is defined as a triple consisting of *expertise_level*, *updates*, and *safe_to_update*. The purpose of the expertise-exchange procedure is to bring everyone's expertise to a common base that is consistent with these and other servers' execution histories and is suitable for the resumption of their normal modes of operation.

Definition 5.1 *The cumulative expertise,* $\max_{\mathcal{X}}(X)$*, of a set or a sequence,* X*, of expertise elements is defined as the following triple*

$$\begin{aligned}\max_{\mathcal{X}}(X) = \big\langle & \max_{<_G}\{x.xl \mid x \in X\}, \\ & \max_{<_{||}}\{x.us \mid (x \in X) \wedge (x.xl \in \max_{<_G}\{x.xl \mid x \in X\})\}, \\ & \max_{<_{\mathcal{N}}}\{x.su \mid x \in X\}\big\rangle\end{aligned}$$

As a first step, the server's collaboration with others during an expertise-exchange process aims at advancing everyone's expertise to the highest one know to them as a group. This step is completed with a delivery of the last expertise message via action $\texttt{gprcv}(x)_{p',p}$.

Advancing the servers' expertise achieves two purposes. First, it ensures the propagation of update requests to previously inaccessible replicas. Second, it ensures the future ability of servers to process the queries that are assigned to them.

In addition to advancing their expertise, the servers of primary views have to ensure their ability to process new update requests once they resume their normal activity, which subsumes that they have to start normal activity with identical *updates* sequences, the entire content of which is *safe* and contains as prefixes the *safe* prefixes of all other servers in the system. For this purpose, once the server of a primary view learns that all expertise-exchange messages have been delivered to all servers of this view, it extends its *safe* prefix to cover the entire *updates* sequence adopted during the expertise-exchange process.

The resultant *safe* prefix acts as a new base that all servers of the future primary views will contain in their *updates* sequences. Attainment of this behavior depends on the intersection property of primary views and the fact that subsequent primary views have higher identifiers. (This property is expressed in part 2 of Invariant 7.30 on page 91.)

The established base works as a divider: partially processed update requests that are not included in the base will never find a way to a safe prefix unless they are resubmitted by their original servers. Therefore, once a server of a primary view establishes the base, it moves all pending update requests that are not in this base back for reprocessing. After this step, the server may resume its normal activity, which enables it to process new update and query requests.

The Δ statement

One of the effects of a `newview` action adjusts the *safe_to_update* index of a server by Δ, a derived variable defined below (see Definition 5.2). For convenience, we will refer to this statement as *the* Δ *statement*. We will now take a close look at this statement.

Motivation: In a primary view, different servers may possess different *safe_to_update* indices, which may result in queries being performed with respect to database states that are more advanced than those that are regarded as safe by the queries' original servers. In order to ensure that the servers of subsequent views have sufficiently advanced database states to process queries, each server, upon learning of a new view, adjusts its *safe_to_update* index to be at least as large as the $last(c)$ index of any of its clients.

Definition of Δ: For each server p, we define two derived variables, *last_max* and Δ, to be respectively the largest database index witnessed by clients of p and the amount by which this index surpasses the *safe_to_update* index of p. An invariant will show that $p.\Delta$ can be greater than zero only when $p.view$ is primary and $p.mode$ is *normal*.

Definition 5.2 *For each server p, define last_max and Δ as follows:*

$$p.last\_max = \max_{<_{\mathcal{N}}}\{p.last(c) \mid c \in C \ \wedge \ c.proc = p\}$$

$$p.\Delta = \begin{cases} p.last\_max - p.safe\_to\_update & \text{if } p.last\_max > p.safe\_to\_update \\ 0 & otherwise \end{cases}$$

Handling of the Δ statement: The Δ statements ensure that servers of future views have sufficiently advanced *safe_to_update* indices to enable database replicas to reach sufficiently advanced states that may be required in order to process certain queries. We prove this property in Chapter 8. Note that this property is a part of *liveness*; it ensures that something "good" happens.

On the other hand, the absence of the Δ statement would not violate the *safety* properties of the system; The traces of the implementation would still be included in the traces of the specification. However as a result, the servers would be unable to process certain queries in minority views, because they may require database states that are more advanced than those that the servers know as safe.

For convenience, we introduce the following notation that distinguishes between the two versions of the automata, with and without the Δ statements:

Notation: Denote by $VStoD_p$, I, and T the I/O automata respectively for the server's state-machine, the servers' layer, and the service implementation, each *without* the Δ statements. Likewise, denote by $VStoD'_p$, I', and T' these I/O automata but *with* the Δ statements.

In this notation, our ultimate goal is to show that implementation automaton $\overline{T'}$ implements specification automaton $\overline{S}$ in the sense of trace inclusion. There are two ways we can go about achieving this goal: First, we can give a direct simulation from $\overline{T'}$ to $\overline{D}$. Second, we can give two simulations, one from $\overline{T}$ to $\overline{D}$ and another from $\overline{T'}$ to $\overline{T}$.

An obvious advantage of the first approach is that it is direct. An advantage of the second approach is that it keeps the proof of the replication part of the algorithm separate from that of the load-balancing part, and thus, allows its future reuse in similar settings (e.g., same replication but different load-balancing algorithms). We chose the second approach because it has long-term benefits, and because its complexity is similar to (if not less than) that of the first approach.

Proof Outline: There are two stages that we have to accomplish. First, we have to prove that $\overline{T}$ implements $\overline{D}$ in the sense of trace inclusion. Then, we have to prove that $\overline{T'}$ implements $\overline{T}$ in the sense of trace inclusion.

The second stage is straightforward: The refinement mapping between reachable states of $\overline{T'}$ and $\overline{T}$ is identity. The action correspondence is also identity, except for `newview` actions; A transitions of $\overline{T'}$ that involves actions of the form $\texttt{newview}(v)_p$ simulates the execution sequence of $\overline{T}$ that contains safe notifications for Δ actions of the form $\texttt{safe}(c, u)_{p',p}$ (possibly separated with safe notifications for query requests, which have no effect on the algorithm) followed by the corresponding $\texttt{newview}(v)_p$ action. As a result of this simulation, the *p.safe_to_update* index of $\overline{T}$ is advanced forward by Δ, which corresponds to what is accomplished with the Δ statement in $\overline{T'}$. The only non-trivial part of this stage is to prove that this execution sequence is possible, i.e., that safe notifications for Δ update requests are enabled. We prove this part in Lemma 8.2 of Chapter 8 on 103. Assuming that this lemma holds, we conclude the following lemma:

Lemma 5.1 *Automaton $\overline{T'}$ implements automaton $\overline{T}$ in the sense of trace inclusion.*

We note that, if we went with the first, direct, approach, then the simulation proof that $\overline{T'}$ implements $\overline{D}$ in the sense of trace inclusion would be *identical* to the proof that $\overline{T}$ implements $\overline{D}$ in the sense of trace inclusion, except for the proofs of several invariants in Section 7.4 of Chapter 7.

Chapter 6

Correctness of T: Simulation

In this chapter, we present a mapping from the reachable states of the closed implementation automaton $\overline{T}$ to the reachable states of the closed intermediate-specification automaton $\overline{D}$, and give a simulation proof that this mapping is a refinement. This result, in conjunction with Theorem 4.2 of Chapter 4 and Lemma 5.1 of Chapter 5, implies that the implementation automaton $\overline{T'}$ implements the specification automaton $\overline{S}$ in the sense of trace inclusion.

6.1 High-Level Invariants

The simulation proof given in this chapter relies on the following five high-level invariant assertions. These assertions are specifically designed to be used in the simulation proof; Many of them depend on more general properties of the algorithm. A detailed proof of these assertions is postponed until the next chapter.

Invariant 6.1 *For each server $p \in P$, the value of $p.last\_update$ is bounded from above by $p.safe\_to\_update$, which in turn is bounded from above by the length of $p.updates$ sequence:*

$$p.last\_update \leq p.safe\_to\_update \leq |p.updates|.$$

The following invariant expresses a very specific consistency property, which is a consequence of a more general one that states that *safe* and *done* prefixes of all servers are consistent. This property is discussed in the next chapter.

Invariant 6.2 *For any two servers p_1 and $p_2 \in P$, if the lengths of their* done *prefixes are the same, then their* done *prefixes are the same:*

$$p_1.last\_update = p_2.last\_update \quad implies$$
$$p_1.updates[1..p_1.last\_update] = p_2.updates[1..p_2.last\_update].$$

The following invariant expresses the fact that all update requests that are safe somewhere but has not been executed at there native location are still reflected in their native *map* and *pending* buffers.

Invariant 6.3 *If $\langle c, u\rangle = p.updates[i]$, $c.proc.last\_update < i$, and $i \leq p.safe\_to\_update$, then*

(a) $\langle c,\ u\rangle \in c.proc.map$
(b) $c \in c.proc.pending$

The following invariant states that at most one unexecuted update request per each client can appear at that client's server. This property is a consequence of the fact that clients are blocking.

Invariant 6.4 *For all clients $c \in C$, there exists at most one index $i \in \mathcal{N}$ such that $i > c.proc.last\_update$ and $c = c.proc.updates[i].c$.*

The following invariant expresses the properties of packets on a point-to-point channel. It is the key to correctness of queries.

Invariant 6.5 *For all packets $\langle c, a, l, g\rangle \in in\text{-}transit_{p',p}$, it follows that $c.proc = p$. Moreover, if $p.view.id = g$ then*

(a) $c \in dom(p.map) \wedge p.map(c) \in Query$
(b) $c \in p.pending$
(c) $a = p.map(c)(compose(p.updates[1..l])(db_0))$
(d) $l \geq p.last(c)$
(e) $l \leq \max_{\wp}\{\wp.last\_update\}$

6.2 Refinement Mapping $\boldsymbol{TD} : \overline{T} \rightarrow \overline{D}$

We want to construct a *refinement mapping* $TD()$ that maps each reachable state of implementation $\overline{T}$ to a reachable state of specification $\overline{D}$. This mapping has to satisfy the following two properties:

1. Basis: If t is any initial state of $\overline{T}$, then $TD(t)$ is an initial state of $\overline{D}$.

2. Inductive Step: If t and $TD(t)$ are reachable states of $\overline{T}$ and $\overline{D}$ respectively, and (t, π, t') is a step of $\overline{T}$, then there exists an execution fragment of $\overline{D}$ from $TD(t)$ to $TD(t')$ with the same trace as π.

A state of $\overline{D}$ consists of the following components: *map*, *dbs*, *last*, *delay*, and *c.busy* for all $c \in C$. Function $TD()$ should specify how to construct these components from a reachable state t of $\overline{T}$, in a way that preserves the two properties above.

6.2.1 The Mapping

Lemma 6.1 *The following function is a refinement from $\overline{T}$ to $\overline{D}$ with respect to reachable states of $\overline{T}$ and $\overline{D}$.*[1]

$$
\begin{array}{rcl}
TD(t : \overline{T}) & \rightarrow & \overline{D} = \\
\textit{let } t.done & = & t[\wp].updates[1..t[\wp].last\_update], \textit{ where } \wp \textit{ is any such that} \\
& & t[\wp].last\_update = \max_p \{t[p].last\_update\} \\
dbs & = & db_0 + apply\,(scan(t.done_{\mathcal{U}}), db_0) \\
map & = & \bigcup_{p \in P} t[p].map \\
last & = & \bigcup_{p \in P} t[p].last \\
delay & = & \{\langle t.done[i].c,\ i\rangle \mid 1 \le i \le |t.done| \ \wedge\ t[t.done[i].c.proc].last\_update < i\} \\
c.busy & = & c.busy \quad \textit{for all } c \in C
\end{array}
$$

Definitions of $TD(t).map$, $TD(t).last$, and $TD(t)[c].busy$ are straightforward.

Invariant 6.2 states that all executed sequences of the same length are in fact the same. Derived variable $t.done$ denotes the longest sequence of update requests processed by

[1] *Notation: If s is a sequence "$\langle c_1, u_1\rangle, \ldots, \langle c_n, u_n\rangle$", then s_u denotes the sequence "$u_1, \ldots, u_n$".*

one of the servers. This sequence corresponds to all modifications applied to the database of $\overline{D}$, which explains the way $TD(t).dbs$ is defined.

Domain of $TD(t).delay$ consists of ids of update requests that have been executed somewhere (i.e., in $t.done$) but not at their native locations (i.e., the $last\_update$ at their native locations have not yet surpassed these update requests). With each c in this domain we associate its position in sequence $t.done$. This position corresponds to the last database state witnessed by client c, which explains the way $d.delay$ is defined.

6.2.2 Action Correspondence

Automaton $\overline{D}$ has five types of actions. Actions of the types $\texttt{request}(r)_c$ and $\texttt{reply}(o)_c$ are simulated when $\overline{T}$ takes the corresponding actions. Actions of the type $\texttt{query}(c)$ are simulated when $\overline{T}$ executes $\texttt{ptprcv}(c, a, l, g)_{p',p}$ with $g = p.view.id$. The last two types of actions, $\texttt{update}(c, u)$ and $\texttt{service}(c)$, are both simulated under certain conditions when $\overline{T}$ executes $\texttt{update}(c, u)_p$: We define an action $\texttt{update}(c, u)_p$ of $\overline{T}$ as *leading* when $t[p].last\_update = \max_{\wp}\{t[\wp].last\_update\}$, and as *native* when $c.proc = p$. Actions that are just leading simulate $\texttt{update}(c, u)$, that are just native simulate $\texttt{service}(c)$, that are leading and native simulate "$\texttt{update}(c, u)$,$\texttt{service}(c)$", and that are neither simulate empty transitions. Transitions of $\overline{T}$ with any other actions simulate empty transitions of $\overline{D}$. This is summarized in Figure 6.1.

Figure 6.1 Circumstances under which each action of $\overline{D}$ is simulated by $\overline{T}$.

$\overline{D}$	$\overline{T}$	Condition
$\texttt{request}(r)_c$	$\texttt{request}(r)_c$	
$\texttt{reply}(o)_c$	$\texttt{reply}(o)_c$	
$\texttt{update}(c, u)$	$\texttt{update}(c, u)_p$	$t[p].last\_update = \max_{\wp}\{t[\wp].last\_update\}$
$\texttt{service}(c)$	$\texttt{update}(c, u)_p$	$c.proc = p$
$\texttt{query}(c, q, l)$	$\texttt{ptprcv}(c, a, l, g)_{p',p}$	$g = t[p].view.id$

6.2.3 Simulation Proof

We now give a simulation proof of Lemma 6.1.

Basis:
The fact that the initial state of $\overline{T}$ maps into the initial state of $\overline{D}$ is straightforward.

Inductive Step:
Before considering each possible transition (t, π, t') of $\overline{T}$, we highlight the tasks that need to be accomplished in order to show that the inductive step holds for this transition:

1. Identify the corresponding action sequence σ of $\overline{D}$;
2. Check that traces exhibited by (t, π, t') and $(TD(t), \sigma, TD(t)')$ are the same;
3. Prove that every action in the sequence σ is enabled; and
4. Prove that $TD(t')$ is equal to $TD(t)'$, the post-state of $(TD(t), \sigma, TD(t)')$.

The correspondence of actions has been presented in Figure 6.1 on page 56. Task 2 is just a check that σ contains the same external action as π. The last two tasks rely on invariants presented in Section 6.1 on page 53 and involve reasoning about π and σ.

We note the following four facts here, in order to eliminate their repetition in many places of the proof.

1. Two states are equal if their corresponding state components are equal. A variable of $\overline{D}$ is the same in states $TD(t)$ and $TD(t')$ if the variables of $\overline{T}$ that define it are the same in states t and t'.
2. If, for all $p \in P$, $t'[p].last\_update = t[p].last\_update$, then $t'.done = t.done$, $TD(t').dbs = TD(t).dbs$, and $TD(t').delay = TD(t).delay$.
3. Many transitions (t, π, t') of $\overline{T}$ correspond to empty transitions of $\overline{D}$. Task 2 is true if π is not an external action of $\overline{T}$. Task 3 is satisfied since empty transitions are always enabled. Task 4 involves showing that $TD(t') = TD(t)$, which is true if none of the variables of $\overline{T}$ that appear in $TD()$ are changed as a result of π.
4. It is straightforward to see that the refinement is preserved for the *busy* components. We, therefore, omit restating this fact in the proof.

We now investigate each of the possible actions π taken by some processor $\wp \in P$:

1. $\pi = \texttt{request}(r)_c$ — the corresponding action of $\overline{D}$ is $\sigma = \texttt{request}(r)_c$. Both are external actions, so their traces are the same. Action σ is enabled since action π is enabled, which means that $t[c].busy$ and $TD(t)[c].busy$ are both *false*. State variables *dbs*, *last*, and *delay* are the same in $TD(t)$ and $TD(t)'$. They are also the same in $TD(t)$ and $TD(t')$ since none of the variables of $\overline{T}$ involved in their definition in $TD()$ are affected by π. Thus, they are the same in $TD(t)'$ and $TD(t')$. The only state component left to consider is *map*:

$$\begin{aligned}
TD(t)'.map &= TD(t).map[c:r] \\
&= \left(\bigcup_{p\in P} t[p].map\right)[c:r] \\
&= \bigcup_{p\in(P-\wp)} t[p].map \cup t[\wp].map[c:r] \\
&= \bigcup_{p\in(P-\wp)} t'[p].map \cup t'[\wp].map \\
&= \bigcup_{p\in P} t'[p].map \\
&= TD(t').map
\end{aligned}$$

2. $\pi = \texttt{reply}(o)_c$ — the corresponding action of $\overline{D}$ is $\sigma = \texttt{reply}(o)_c$. Both are output actions, so their external traces are the same. Since π is enabled, its precondition $t[\wp].map(c) = o$ is satisfied. By the $TD()$ mapping, $TD(t).map(c) = o$, and thus σ is enabled. Using the same reasoning as in 1 we can show that state variables *map*, *dbs*, *last*, and *delay* are the same in $TD(t)'$ and $TD(t')$.

3. $\pi = \texttt{update}(c, u)$ — the corresponding action sequence of $\overline{D}$ is not constant like in the cases above. It depends on the current state of $\overline{T}$. More specifically, it depends on whether or not π is a *leading* update, and on whether or not it is a *native* update.

   Equations ($\mathbb{L}$) and ($\mathbb{N}$) express what it means for an update to be leading and native.

$$t[\wp].last\_update = \max_p\{t[p].last\_update\} \tag{$\mathbb{L}$}$$

$$c.proc = \wp \tag{$\mathbb{N}$}$$

   We write $\overline{(\mathbb{L})}$ and $\overline{(\mathbb{N})}$ when the opposites apply. Also, for convenience, we use i to denote the value of $t[\wp].last\_update + 1$.

   Recall from the code that c and u stand for $t[\wp].updates[i].c$ and $t[\wp].updates[i].u$.

   Delayed spec $\overline{D}$ takes action $\texttt{update}(c, u)$ when ($\mathbb{L}$), and it takes action $\texttt{service}(c)$ when ($\mathbb{N}$). A particular state t may exhibit none, one, or both of these properties, yielding to the following four possibilities for σ:

$$\{(\lambda),\quad (\texttt{update}(c,u)),\quad (\texttt{service}(c)),\quad (\texttt{update}(c,u),\texttt{service}(c))\}.$$

We investigate each of these possibilities.

a) $\sigma = (\lambda)$ when $\overline{(\mathbb{L})}$ and $\overline{(\mathbb{N})}$.
Since $\overline{(\mathbb{N})}$, the if-then clause of π is not executed. This means that $TD(t').map$ $= TD(t).map$ and $TD(t').last = TD(t).last$.

We now show that $TD(t').dbs = TD(t).dbs$ and $TD(t').delay = TD(t).delay$. $\overline{(\mathbb{L})}$ implies that

$$\begin{aligned} t'[\wp].last\_update &= t[\wp].last\_update + 1 \\ &< \max_p\{t[p].last\_update\} + 1 \\ &\leq \max_p\{t[p].last\_update\}. \end{aligned}$$

Since, for all $p \neq \wp$, $t'[p].last\_update = t[p].last\_update$, it follows that

$$\max_p\{t'[p].last\_update\} = \max_p\{t[p].last\_update\}.$$

Together with invariant 6.2, this result implies that $t'.done = t.done$, and thus

$$TD(t').dbs = TD(t).dbs.$$

Moreover, by taking into account $\overline{(\mathbb{N})}$, we get

$$TD(t').delay = TD(t).delay.$$

b) $\sigma = \texttt{update}(c,u)$ when $(\mathbb{L})$ and $\overline{(\mathbb{N})}$.
First we have to show that σ is enabled. Since π is enabled, we know that its precondition is true:

$$t[\wp].last\_update < t[\wp].safe\_to\_update.$$

This implies that the following bound holds for i:

$$t[\wp].last\_update < i \leq t[\wp].safe\_to\_update.$$

($\mathbb{L}$) and invariant 6.3 imply $t[c.proc].map(c) = u$, and thus, the first precondition on σ is true:

$$TD(t).map(c) = u.$$

The fact that the second precondition, $c \notin dom(TD(t).delay)$, on σ is true follows from $t[c.proc].last\_update \leq t[\wp].last\_update$ and Invariant 6.4, which together imply that c does not occur in $t.done$ between indices $t[c.proc].last\_update+1$ and $|t.done|$.

Because of $\overline{(\mathbb{N})}$, the if-then clause of π is not executed. Thus, $TD(t').map = TD(t).map$ and $TD(t').last = TD(t).last$. Action σ does not modify these components either. So $TD(t').map = TD(t)'.map$ and $TD(t').last = TD(t)'.last$.

We now consider the remaining components, dbs and $delay$.

$$\begin{aligned} TD(t)'.dbs &= TD(t).dbs + u(TD(t).dbs.last) \\ &= db_0 + \text{apply}\,(\text{scan}(t.done), db_0) + u(TD(t).dbs.last) \\ &= db_0 + \text{apply}\,(\text{scan}(t.done), db_0) + u(\text{compose}(t.done)(db_0)) \end{aligned}$$

($\mathbb{L}$) and $t'[\wp].last\_update = t[\wp].last\_update + 1$ imply $t'.done = t.done + \langle c,\, u\rangle$

$$\begin{aligned} &= db_0 + \text{apply}\,(\text{scan}(t.done), db_0) + \text{compose}(t'.done)(db_0) \\ &= db_0 + \text{apply}\,(\text{scan}(t'.done), db_0) \\ &= TD(t').dbs. \end{aligned}$$

Finally,

$$\begin{aligned} TD(t)'.delay &= TD(t).delay \,\cup\, \langle c,\, |TD(t).dbs|\rangle \\ &= TD(t).delay \,\cup\, \langle c,\, |t.done| + 1\rangle \\ &= TD(t).delay \,\cup\, \langle c,\, i\rangle \\ &= TD(t').delay, \end{aligned}$$

where the last step is justified by $\overline{(\mathbb{N})}$.

c) $\sigma = \texttt{service}(c)$ when $\overline{(\mathbb{L})}$ and ($\mathbb{N}$).

This action is enabled because its precondition, $c \in dom(TD(t).delay)$, follows from $\langle c,\ i\rangle \in TD(t).delay$, which is true because

$$\overline{(\mathbb{L})} \Rightarrow i = t[\wp].last\_update + 1 \leq |t.done|$$
$$(\mathbb{N}) \Rightarrow t[t.done[i].c.proc].last\_update = t[\wp].last\_update < i.$$

We now consider the state components of $\overline{D}$:

$$\begin{aligned}
TD(t)'.map &= TD(t).map[c : ok] \\
&= (\textstyle\bigcup_{p\in P}\ t[p].map)[c : ok] \\
&= \textstyle\bigcup_{p\in(P-\wp)}\ t[p].map\ \cup\ t[\wp].map[c : ok] \\
&= \textstyle\bigcup_{p\in(P-\wp)}\ t'[p].map\ \cup\ t'[\wp].map \\
&= \textstyle\bigcup_{p\in P}\ t'[p].map \\
&= TD(t').map.
\end{aligned}$$

Using the same reasoning as in a) we can show that

$$TD(t)'.dbs = TD(t').dbs.$$

Similarly to *map* we get

$$\begin{aligned}
TD(t)'.last &= TD(t).last[c : TD(t).delay(c)] \\
&= (\textstyle\bigcup_{p\in P}\ t[p].last)[c : i] \\
&= \textstyle\bigcup_{p\in(P-\wp)}\ t[p].last\ \cup\ t[\wp].last[c : i] \\
&= \textstyle\bigcup_{p\in(P-\wp)}\ t'[p].last\ \cup\ t'[\wp].last \\
&= \textstyle\bigcup_{p\in P}\ t'[p].last \\
&= TD(t').last.
\end{aligned}$$

Finally, since $t'[\wp].last\_update = i$, $\langle c,\ i\rangle$ is no longer in $TD(t').delay$, so

$$TD(t)'.delay = TD(t).delay\ -\ \langle c,\ i\rangle = TD(t').delay.$$

d) $\sigma = (\texttt{update}(c, u),\ \texttt{service}(c))$ when $(\mathbb{L})$ and $(\mathbb{N})$.
The first action is enabled because of the reasons given in b). The second action is enabled because the first one defines *delay* at c. Using a straightforward

combination of b) and c) we can show that $TD(t)' = TD(t')$.

4. $\pi = \texttt{ptprcv}(c, a, l, g)_\wp$ — the corresponding action of $\overline{D}$ is $\texttt{query}(c, q, l)$ provided that $g = t[\wp].view.id$, where $q = TD(t).map(c)$.

   The first precondition on σ is true because of part (a) of Invariant 6.5. Using parts (d) and (e) of Invariant 6.5, we get the following bounds for l:

$$TD(t).last(c) = t[\wp].last(c) \leq l \leq |t.done| = |TD(t).dbs - 1|,$$

   which establishes the truth of the second precondition.

   We now consider the state components of $\overline{D}$. Notice that q is $t[\wp].map(c)$.

$$\begin{aligned} TD(t)'.map &= TD(t).map[c : q(TD(t).dbs[l+1])] \\ &= (\textstyle\bigcup_{p \in P} t[p].map)[c : q(\text{compose}(t[\wp].updates[1..l])(db_0))] \end{aligned}$$

   Using part (c) of Invariant 6.5,

$$\begin{aligned} &= \textstyle\bigcup_{p \in (P - \wp)} t[p].map \,\cup\, t[\wp].map[c : a] \\ &= \textstyle\bigcup_{p \in (P - \wp)} t'[p].map \,\cup\, t'[\wp].map \\ &= \textstyle\bigcup_{p \in P} t'[p].map \\ &= TD(t').map. \end{aligned}$$

   Similarly to map, we get

$$\begin{aligned} TD(t)'.last &= TD(t).last[c : l] \\ &= (\textstyle\bigcup_{p \in P} t[p].last)[c : l] \\ &= \textstyle\bigcup_{p \in (P - \wp)} t[p].last \,\cup\, t[\wp].last[c : l] \\ &= \textstyle\bigcup_{p \in (P - \wp)} t'[p].last \,\cup\, t'[\wp].last \\ &= \textstyle\bigcup_{p \in P} t'[p].last \\ &= TD(t').last. \end{aligned}$$

None of the variables in the definitions of dbs and $delay$ are affected by π, therefore

$$\begin{aligned} TD(t)'.dbs &= TD(t).dbs = TD(t').dbs \\ TD(t)'.delay &= TD(t).delay = TD(t').delay. \end{aligned}$$

5. The rest of the actions of $\overline{T}$ do not affect any of the variables involved in $TD()$. They correspond to empty transitions of $\overline{D}$. For one action, $\texttt{gprcv}(x)_{p',p}$, we have to invoke a theorem (see Corollary 7.34 on page 93) that says that if (t, π, t') is a transition of $\overline{T}$, then, for all p in P, $t[p].updates[1..t[p].last\_update]$ is a prefix of $t'[p].updates[1..t'[p].last\_update]$. In the case of $\texttt{gprcv}(x)_{p',p}$, this theorem implies that $t[p].updates[1..t[p].last\_update]$ remains unchanged.

   This completes the proof of Lemma 6.1 which established function $TD()$ as a refinement mapping from implementation $\overline{T}$ to delayed specification $\overline{D}$.

Theorem 6.2 *Automaton $\overline{T}$ and automaton $\overline{T'}$ implement automaton $\overline{S}$ in the sense of trace inclusion.*

Proof 6.2: Follows immediately from Lemma 5.1, Lemma 6.1, Theorem 4.2, and transitivity of the "implements" relation. ■

Chapter 7

Correctness of T: Invariants

The goal of this chapter is to prove the high-level invariants used in the simulation argument of chapter 6. Most of these invariants (e.g., Invariant 6.2) are the consequences of more general properties of the system. To prove these properties, we state and prove an elaborate collection of invariants. These invariants are given in terms of the state variables of T taken at a single reachable state of T; their statements sometimes involve variables that are derived from the state variables of T.

7.1 View-Related Derived Variables

In this section, we present a number of useful view-related derived variables, such as a derived function that maps processes to the sets of views that contain them as members. These derived variables are based solely on the state components of VS.

First, notice that an identifier of a created view uniquely defines the view and its corresponding membership set (this will be demonstrated by part 1 of Invariant 7.1). We introduce the derived functions $GtoV$ and $GtoS$ that map a view identifier to its view and to its membership set, respectively. We sometimes allow ourselves to exploit the equivalence between views and their identifiers by using these terms interchangeably.

State variable *created* captures all views that have been created by VS. We define $created\_views(p)$ to be a set of all created views that contain p as a member. Notice that for any $p \in P$, the initial view v_0 is always in $created\_views(p)$.

$$created\_views(p) = \{v \mid v \in created \ \wedge \ p \in v.set\}$$

Elements of $created\_views(p)$ are ordered according to the total order on view identifiers. Given an element v of $created\_views(p)$, different from v_0, function $prev\_view$ produces an immediately preceding element of $created\_views(p)$.

$$prev\_view(p, v) = \begin{cases} \max\{v' \mid v' \in created\_views(p) \ \wedge \ v'.id < v.id\} & \text{if } v \neq v_0 \\ \perp & \text{otherwise} \end{cases}$$

Recall that $\mathcal{Q}$ denotes a fixed set of quorums, subsets of P, such that any two have nonempty intersection. We define a derived set $primary\_views$ consisting of created views whose membership sets are quorums. (Invariant 7.3 will show that $t[p].view.set \in \mathcal{Q}$ if and only if $current\_viewid[p] \in primary\_views$).

$$primary\_views = \{v \mid v \in created \ \wedge \ v.set \in \mathcal{Q}\}$$

We use $created\_viewids$, $prev\_viewid$, and $primary\_viewids$ to stand for the variables that are defined as above, but in terms of view identifiers.

$$\begin{aligned} createdid &= \{g \mid \exists S \,.\, \langle g,\, S\rangle \in created\} \\ created\_viewids(p) &= \{g \mid \exists S \,.\, \langle g,\, S\rangle \in created \ \wedge \ p \in S\} \\ prev\_viewid(p, g) &= \begin{cases} \max\{g' \mid g' \in created\_viewids(p) \ \wedge \ g' < g\} & \text{if } g \neq g_0 \\ \perp & \text{otherwise} \end{cases} \\ primary\_viewids &= \{g \mid \exists S \,.\, \langle g,\, S\rangle \in created \ \wedge \ S \in \mathcal{Q}\} \end{aligned}$$

7.2 VS Invariants

This section presents basic invariants of VS. These invariants are preserved when VS is composed with the other automata to yield the service implementation automaton T. We use these invariants in the later sections to prove invariants of T.

All but the last three statements of the following invariant are reprinted from [14, VS

Lemma 4.1, page 10]; the last three statement reveal additional properties of VS.

Invariant 7.1 *For any $p \in P$, $S \subseteq P$, $m \in M$, $g \in G$, the following statements are true:*

1. *If $g \in createdid$ then there is a unique S such that $\langle g, S\rangle \in created$.*
2. *$current\text{-}viewid[p] \in createdid$.*
3. *If $\langle current\text{-}viewid[p], S\rangle \in created$ then $p \in S$.*
4. *If $pending[p, g] \neq \lambda$ then $g \in createdid$.*
5. *If $pending[p, g] \neq \lambda$ then $g \leq current\text{-}viewid[p]$.*
6. *If $queue[g] \neq \lambda$ then $g \in createdid$.*
7. *If $\langle m, p\rangle$ is in $queue[g]$ then $g \leq current\text{-}viewid[p]$.*
8. *$next[p, g] \leq length(queue[g]) + 1$.*
9. *$next\text{-}safe[p, g] \leq length(queue[g]) + 1$.*
10. *$next\text{-}safe[p, q] \leq next[p, g]$.*
11. *If $\langle g, S\rangle \in created$ and $next[p, g] \neq 1$ then $p \in S$.*
12. *If $\langle g, S\rangle \in created$ and $next\text{-}safe[p, g] \neq 1$ then $p \in S$.*
13. *If $\langle g, S\rangle \in created$ then $next\text{-}safe[p, g] \leq next[p', g]$ for all $p' \in S$.*
14. *If $next[p, g] > 1$ or $next\text{-}safe[p, g] > 1$ then $g \leq current\text{-}viewid[p]$.*
15. *If $m \in pending[p, g]$ or $\langle m, p\rangle \in queue[g]$ then $p \in GtoS(g)$.*

Proof 7.1: All are straightforward by induction. ■

7.3 Basic Invariants

In this section, we study basic invariants of the system. First, we exhibit the correspondence between view-related information in $VStoD_p$ and VS. Then, we investigate the structure of the total order that VS provides on messages sent within one view. We prove facts such as "expertise messages sent within one view appear as a prefix of the total order on messages of this view." Finally, we define derived variables that express the notions of views being established and normal, and state properties that relate these notions to the state variables of VS and $VStoD_p$.

7.3.1 Consistency of Current Views

The following invariant states that VS has a correct notion of each server's current view. (Thus, in the future, we use $p.view.id$ and $current\_viewid[p]$ interchangeably.)

Invariant 7.2 $p.view.id = current\_viewid[p]$ *and* $p.view = GtoV(current\_viewid[p])$.

Proof 7.2: Easy induction. The only critical action is `newview`. ■

As a corollary from Invariant 7.2, we note the following invariant.

Invariant 7.3 *The following statements are true for all* $p \in P$*:*

1. $p.view \in created$
2. $p.view \in created\_views(p)$
3. $p.view.set \in \mathcal{Q}$ *if and only if* $current\_viewid[p] \in primary\_viewids$.

Proof 7.3: Follows immediately from $p.view.id = current\_viewid[p]$ (Invariant 7.2) and Invariant 7.1 (parts 1, 2, and 3). ■

7.3.2 Initial and Primary Views

The following invariant presents the basic properties of the initial view v_0. It states that processes in the initial view always operate in normal mode and never communicate expertise messages. The first part involves the state variables of $VStoD_p$, while the second — of VS. These parts appear under the same invariant statement because the proof of the second part depends on the first. (Notice the usage of $queue_x$ to denote the subsequence of $queue$ consisting solely of expertise messages.)

Invariant 7.4 *The following statements are true:*

1. $p.view.id = g_0 \Rightarrow p.mode = normal$
2. $pending_x[*, g_0] = [\,] \wedge queue_x[g_0] = [\,]$

Proof 7.4: For part 1, the critical actions are $\texttt{newview}(v)_p$ (use its precondition and effect) and $\texttt{gpsnd}(x)_p$ (use the inductive hypothesis) — The invariant is vacuously true. For part 2, the critical actions are $\texttt{gpsnd}(x)_p$ (use part 1) and $\texttt{vs-order}(m, p, g)$ (use the inductive hypothesis). ■

The following invariant states that update requests are communicated only by processes that are members of a primary view.

Invariant 7.5 *For all $\langle g, S\rangle \in created$, if there is an update message on $pending[p, g]$ or on $queue[g]$ then $g \in primary\_viewids$.*

Proof 7.5: Basis is trivial. For the inductive step, the critical actions are `gpsnd` and `vsorder`, both are straightforward. ■

7.3.3 Expertise Messages

Per-process uniqueness of expertise messages in one view

The following two invariants show that, in any state of the system, there is at most one expertise message from each process in a given view.

First, for all $p \in P$, if p is in *expertise_broadcast* mode then there are no expertise messages from p in *VS* associated with the current view of p. This is true because when p is in *expertise_broadcast* mode it has not yet sent its expertise message.

Invariant 7.6 *If* $p.mode = expertise\_broadcast$ *then there is no* x *such that* $x \in pending[p, p.view.id]$ *or* $\langle x,\ p\rangle \in queue[p.view.id]$.

Proof 7.6:

Basis: In the initial state, every process is in *normal* mode. Thus, the proposition is vacuously true.

Inductive Step: The critical actions are `newview`, `gpsnd`, and `vs_order`. The first one changes the antecedent from false to true, while the last two may affect the conclusion.

$\underline{\texttt{newview}(v)_p}$: One of the preconditions that must hold for this action to be enabled is $v.id > current\_viewid[p]$. When used with the contrapositives of parts 5 an 7 of Invariant 7.1, it implies that $pending[p, v.id]$ is empty and that there are no messages from p on $queue[v.id]$. Absence of all messages implies absence of expertise messages there. Thus, in the poststate, the proposition is trivially true.

$\underline{\texttt{gpsnd}(x)_p}$: When enabled, this action places x on $pending[p, current\_viewid[p]]$, but its second effect changes the mode of p from *expertise_broadcast* to *expertise_collection*. Thus, in the poststate, the proposition is vacuously true.

$\underline{\texttt{vs\_order}(x, p, g)}$: The precondition on this action and the contrapositive of the inductive hypothesis imply that, in the prestate, $p.mode$ is different from *expertise_broadcast*. Since $p.mode$ is unchanged as the effect of this action, the proposition is vacuously true for the poststate.

■

We can use this invariant to prove the per-process uniqueness of expertise messages in each view.

Invariant 7.7 *For all* $p \in P$ *and for all* $g \in G$, *there is at most one expertise message from* p *in both* $pending[p, g]$ *and* $queue[g]$.

Proof 7.7:

Basis: In the initial state, $pending[p, g]$ and $queue[g]$ are empty. Thus, the proposition is trivially true.

Inductive Step: The critical actions are `gpsnd`, and `vs_order`.

$\underline{\texttt{gpsnd}(x)_p}$: The precondition on this action ensures that, in the prestate, *p.mode* is *expertise_broadcast*. Invariant 7.6 implies that there are no expertise messages from p in $pending[p, p.view.id]$ and $queue[p.view.id]$. Since the effect of this action places x on $pending[p, p.view.id]$, there is only one such message in the poststate. Thus, the proposition holds.

$\underline{\texttt{vs\_order}(x, p, g)}$: The effect of this action moves an expertise message from $pending[p, g]$ to $queue[g]$. Therefore, the number of expertise messages in these sequences remains unchanged. The inductive hypothesis ensures that the proposition holds.

■

Upper-limit on the number of expertise messages in one view

The following invariant states that the total number of expertise messages that may appear in any view is limited from above by the number of members in that view. This property is obvious in the light of Invariant 7.7, which limits to one the number of expertise messages sent by each process.

Invariant 7.8 *For each view v, the number of expertise messages appearing in VS is less than or equal to the size of v's membership set. If g denotes $v.id$, then*

$$|pending_x[*, g]| + |queue_x[g]| \leq |v.set|$$

Proof 7.8: Since the only messages that occur in $pending[*, g]$ and $queue[g]$ are from members of v (part 15 of Invariant 7.1), and since Invariant 7.7 states that there is at most one expertise message from each process associated with g, it follows that the invariant is true. ■

Expertise messages are a prefix of the total order on messages of one view

Recall that state variables *expert_counter1* and *expert_counter2* of $VStoD_p$ track respectively the number of delivered expertise messages and the number of delivered safe notifications for these messages. To show that the subsequence of $queue[g]$ consisting of expertise messages is a prefix of $queue[g]$ we develop a number of invariants that involve these state variables.

The first invariant relates the values of *expert_counter1* and *expert_counter2* to the lengths of the delivered and the safe prefixes of $queue[g]$.

Invariant 7.9 *For all $p \in P$, the following statements are true:*

1. $p.expert\_counter1 = |queue_x[p.view.id][1..(next[p, p.view.id] - 1)]|$
2. $p.expert\_counter2 = |queue_x[p.view.id][1..(next\_safe[p, p.view.id] - 1)]|$

Proof :

Basis: In the initial state, both sides of each of these propositions are equal to zero.

Inductive Step: The critical actions are $\texttt{newview}(v)_p$, $\texttt{vs\_order}(x, p, g)$, $\texttt{gprcv}(x)_{p',p}$, and $\texttt{safe}(x)_{p',p}$.

$\underline{\texttt{newview}(v)_p}$: A precondition on this action ensures that $v.id > current\_viewid[p]$. Part 14 of Invariant 7.1 implies that if $next[p, g]$ or $next\_safe[p, g]$ is greater than 1, then $g \leq current\_viewid[p]$. The contrapositive of this lemma implies that the right sides of the two propositions are equal to zero. One of the effects of this action sets counters *p.expert_counter1* and *p.expert_counter2* to zero. So, both sides are equal, and the proposition holds in the poststate.

$\underline{\texttt{vs\_order}(x, p, g)}$: Parts 8 and 9 of Invariant 7.1 state that *next* and *next_safe* are bounded by the size of their corresponding *queue* sequence. Thus, even though one of the effects of this action appends an element to $queue[g]$, the subsequences appearing on the right sides of the two propositions are not affected. The counters are unaffected, as well.

$\underline{\texttt{gprcv}(x)_{p',p}}$: only the first proposition is affected — both sides are increased by 1.

$\underline{\texttt{safe}(x)_{p',p}}$: only the second proposition is affected — both sides are increased by 1.

■

Invariants 7.8 and 7.9 imply the following corollary:

Invariant 7.10 *The values of $p.expert\_counter1$ and $p.expert\_counter2$ are less than or equal to $|p.view.set|$.*

The following invariant relates the values of the server's *mode* and *expert_counter2*. It states that, if a server p is not in the initial view and is operating under *normal* *mode*, then it has successfully completed an expertise-exchange process, which means that p has received safe notification for expertise messages from all members of its current view, and therefore, its *expert_counter2* equals $|p.view.set|$.

Invariant 7.11 $(p.view.id \neq g_0 \ \wedge \ p.mode = normal) \Leftrightarrow p.expert\_counter2 = |p.view.set|$.

Proof 7.11:

Basis: In the initial view, both sides of the proposition are false, so the proposition is vacuously true (domination law). The left side is false because the viewid of any process in the initial state is g_0. As for the right side: Invariant 7.2 and Invariant 7.1 (part 1, 2, 3) imply $p \in p.view.set$. This means that $|p.view.set|$ is greater than or equal to one. Therefore, $|p.view.set|$ may not be equal to zero, the initial value of $p.expert\_counter2$. Thus, the right side is false as well.

Inductive Step: The critical actions are $\mathtt{newview}(v)_p$ and $\mathtt{safe}(x)_{p',p}$.
$\underline{\mathtt{newview}(v)_p}$: In the poststate, both sides are false. The left side is false because $p.mode$ is set to *expertise_broadcast*. The right side is false because $p.expert\_counter2$ is set to zero, while (as was argued in the Basis case) $|p.view.set|$ cannot be zero.
$\underline{\mathtt{safe}(x)_{p',p}}$: Using the inductive hypothesis, we can show that, if this action happens, then both sides of the proposition are false in the prestate. Indeed, Corollary 7.10 states that $p.expert\_counter2$ cannot be larger than $|p.view.set|$. Since $p.expert\_counter2$ is increased by 1 as an effect of this action, we can conclude that it is strictly less than $|p.view.set|$ in the prestate. So, the right side is false. Inductive hypothesis implies that the left side is false as well. Now, for the poststate, we have two cases:

(a) If $p.expert\_counter2$ reaches $|p.view.set|$ as a result of this action, then the right side of the proposition becomes true as well: $p.mode$ is set to *normal* as an effect of this action, and the fact that $p.view.id$ is not equal to g_0 follows from the contrapositive of Invariant 7.4 (part 1) used with the prestate value of $p.mode$.

(b) Otherwise, if $p.expert\_counter2$ is still less than $|p.view.set|$ as a result of this action, then both sides of the proposition remain false.

■

We need one more intermediate invariant before we can prove the prefix property on expertise messages.

Invariant 7.12 *For all created views $\langle g,\ S\rangle$, other than g_0, if there exists an update or a query message in either $pending[*, g]$ or $queue[g]$ then $|queue_x[g]| = |S|$*

Proof 7.12:

Basis: The proposition is vacuously true for the initial state.

Inductive Step: The critical actions are `createview`(v), and `gpsnd`$(m)_p$ and `vs_order`(m, p, g) when m is an update or a query message (i.e., $\langle c,\ u\rangle$ or $\langle c,\ q\rangle$).

<u>`createview`(v):</u> The precondition on this action ensures that $v.id$ is greater than all previously created viewids. Contrapositives of parts 4 and 6 of Invariant 7.1 imply that $pending[*, v.id]$ and $queue[v.id]$ are empty; that is, there are no messages on these sequences. Therefore, the proposition is vacuously true.

<u>`gpsnd`$(m)_p$:</u> This action has a precondition "$p.mode = normal$." Invariant 7.11 tells us that $p.expert\_counter2 = |p.view.set|$, which, when used with Invariants 7.9 and 7.8, yields the desired conclusion: $|queue_x[p.view.id]| = |p.view.set|$.

<u>`vs_order`(m, p, g):</u> The fact that the proposition holds for the poststate follows immediately from the inductive hypothesis.

■

Finally, we are able to prove the prefix property on expertise messages.

Invariant 7.13 *For all created views g, the subsequence of $queue[g]$ that consists solely of expertise messages is a prefix of $queue[g]$.*

Proof 7.13: If $g = g_0$, then there are no expertise messages on the queue of *VS* (Invariant 7.4) — so we are all set. Otherwise, we proceed by induction. Consider a critical action `vs_order`, which places a message $\langle x,\ p\rangle$ on $queue[g]$. In order for the proposition to hold, there should be no update and no query messages on $queue[g]$. We show this fact by contradiction: If there is an update or a query message on $queue[g]$, it means that, in the prestate, the number of expertise messages on $queue[g]$ is $|GtoS(g)|$ (Invariant 7.12). By Invariant 7.8 this number is the largest possible, implying the impossibility of this action in the first place. ■

7.3.4 Established and Normal Views

In Chapter 5, when we described the server's automaton $VStoD_p$, we used the terms *established* and *normal* to refer to certain states of the server's view. The server was said to "establish" its view when it received an expertise message from the last server in its view. Likewise, the server's view was said to become "normal" when the server received safe notification for the last expertise message in its view. The initial view of any server was considered both established and normal.

Derived Functions

The following two functions map each process to the sets of established and normal views of which it is a member. They are derived from the state variables of VS.

$$\begin{aligned} \mathit{established\_views}(p) &= \{v_0\} \,\cup\, \{v \mid \mathit{next}[p, v.id] > |v.set|\} \\ \mathit{normal\_views}(p) &= \{v_0\} \,\cup\, \{v \mid \mathit{next\_safe}[p, v.id] > |v.set|\} \end{aligned}$$

Given any view v, we can also define the latest preceding established view of p.

$$\mathit{last\_established\_view}(p, v) = \max_{<_G}\{v' \mid v' \in \mathit{established\_views}(p) \,\wedge\, v'.g \leq v.id\}$$

Notice that $\mathit{last\_established\_view}(p, v)$ is well defined for all views v because $v_0 \in \mathit{established\_views}(p)$ and $v.id_0 \leq v.id$.

The following set captures all the views that are established at all their members.

$$\mathit{totally\_established\_views} = \{v \mid \forall\, p \in v.set \,.\, v \in \mathit{established\_views}(p)\}$$

We will use *established_viewids*, *normal_viewids*, *last_established_viewid*, and *totally_established_viewids* to stand for the variables that are defined as above, but in terms of view identifiers. Since there is an equivalence between views and their view identifiers, we sometimes allow ourselves to use these variables interchangeably.

$$\begin{aligned}
established_viewids(p) &= \{g_0\} \,\cup\, \{g \mid next[p,g] > |GtoS(g)|\}\\
normal_viewids(p) &= \{g_0\} \,\cup\, \{g \mid next_safe[p,g] > |GtoS(g)|\}\\
last_established_viewid(p,g) &= \max_{<_G}\{g' \mid g' \in established_viewids(p) \,\wedge\, g' \leq g\}\\
totally_established_viewids &= \{g \mid \forall\, p \in GtoS(g)\,.\, g \in established_viewids(p)\}
\end{aligned}$$

Invariants

We now present invariants that involve the defined above functions.

First, notice the following relationship between *normal_views*, *established_views*, and *totally_established_views*.

Invariant 7.14 *The following statements are true for all* $p \in P$*:*

1. $normal\_views(p) \subseteq established\_views(p)$

2. $normal\_views(p) \subseteq totally\_established\_views$

Proof 7.14: Follows immediately from the definitions of *normal_views*, *established_views* and *totally_established_views*, and from the fact that $next\_safe[p,g] \leq next[p',g]$ for all $p' \in GtoS(g)$ (Invariant 7.1, part 13). ■

The following two invariants relate the notion of a view v being established or normal to the types of messages that appear on $queue[v.id]$.

First, if a non-initial view v is established at some process, then $queue[v.id]$ contains an expertise message from each member of v.

Invariant 7.15 *For all* p *and* g*, if* $g \neq g_0$ *and* $g \in established\_viewids(p)$ *then the set* $\{\wp \mid \langle x,\, \wp\rangle \in queue[g]\}$ *is the same as* $GtoS(g)$*.*

Proof 7.15: Straightforward. Follows from the invariants in Section 7.3.3. ■

Second, if VS delivered an update or a query request to p in a certain view, then this view is established at p. Likewise, if VS delivered a safe notification for an update or a query request to p in a certain view, then this view is normal in p.

Invariant 7.16 *For all p and g,*

1. *If* $\exists\langle\langle c, r\rangle, p'\rangle \in queue[g][1..(next[p,g]-1)]$ *then* $g \in established\_viewids(p)$.

2. *If* $\exists\langle\langle c, r\rangle, p'\rangle \in queue[g][1..(next\_safe[p,g]-1)]$ *then* $g \in normal\_viewids(p)$.

Proof 7.16: Straightforward induction. Follows from the invariants in Section 7.3.3. ■

The following two statements relate the server's mode of operation to the notion that the server's current view is established or is normal. First, if the server's *mode* is *expertise_broadcast*, then its view is not established. This follows from the fact that the server has not yet submitted its expertise message, which means that it could have not received all expertise messages, and therefore, its view is not established. Second, the server's view being normal is equivalent to the server's mode being normal. This statement is trivial for initial views. For non-initial views, it follows from the fact that the server's mode becomes *normal* when it receives the last safe notification for the expertise-exchange process, which is exactly when the server's view is considered to become normal.

Invariant 7.17 $p.mode = expertise\_broadcast \Rightarrow p.view \notin established\_views(p)$.

Invariant 7.18 $p.mode = normal \Leftrightarrow p.view \in normal\_views(p)$

Proof 7.17 and 7.18: Straightforward induction. Relies on invariants in Section 7.3.3. ■

Functions *established_views* and *normal_views* are defined on state variables of VS. We now express the notion that the current view of a server p is established/normal in terms of the state variables of $VStoD_p$, and then, connect these notions together in Invariant 7.19.

Definition 7.1 *A derived boolean flag* $p.established$ *is defined to be true if and only if either* $p.view.id = g_0$ *or* $p.expert\_counter1 = |p.v.set|$. *Likewise, a derived boolean flag* $p.normal$ *is defined to be true if and only if either* $p.view.id = g_0$ *or* $p.expert\_counter2 = |p.v.set|$.

Invariant 7.19 *The following statements are true for all* $p \in P$*:*

1. $p.established$ *is true if and only if* $p.view \in established\_views(p)$
2. $p.normal$ *is true if and only if* $p.view \in normal\_views(p)$

7.4 Derived Expertise $\mathcal{X}$

This section develops an approach for proving major correctness results about our algorithm, such as consistency of *updates* sequences and of their safe and done prefixes at different servers. Proving such results requires reasoning on the values of state variables at different points of the execution. However, the advantage of invariant proofs over operational proofs is exactly in that they avoid reasoning about multiple states and restrict their properties to single reachable states. In order to carry out invariant proofs and still be able to reason about past values of variables, these values have to be accessible within single states of the system. A traditional approach is to define *history variables* that preserve values of regular state variables as they change throughout the execution. In this work, however, we do not introduce history variables, as we are able to access all necessary historic information at a single state of the *VS* specification.

In particular, a single state of the *VS* specification includes information about previously created views and about their *pending* and *queue* buffers. Using this information, we are able to derive a powerful function, $\mathcal{X}(p, g)$, that maps each process p and a view $g \in created\_viewids(p)$ to (what we claim is) the highest expertise attained by p during its participation in the view g.

The power of this function comes from the fact that it expresses the expertise of a process in a given view *recursively* in terms of the expertise of this and other processes in earlier views. In a sense, this function presents the law according to which the replication part of the algorithm operates. The recursive nature of this function makes it simple to establish various properties of this law by induction, because inductive steps can be proved by unwinding the recursive definition of $\mathcal{X}$ to reach the domains of the underlying inductive hypotheses.

7.4.1 Definition of $\mathcal{X}$

Definition 7.2 *For each process $p \in P$ and each view $g \in G$, we define $\mathcal{X}(p, g)$ to be $\perp$ if $g \notin created\_views(p)$. Otherwise, if $g \in created\_views(p)$, then*

$$\boxed{\begin{array}{l}
\mathcal{X}(p,g) = \\
\textit{If } g \in established_viewids(p) \quad \textit{then} \\
\quad xl = \begin{cases} \max_{\mathcal{X}}(queue_x[g]).xl & \textit{if } g \notin primary_viewids, \\ g & \textit{otherwise.} \end{cases} \\
\quad us = \max_{\mathcal{X}}(queue_x[g]).us + queue_u[g][1..(next[p,g]-1)] \\
\quad su = \begin{cases} \max_{\mathcal{X}}(queue_x[g]).su & \textit{if } g \notin primary_viewids \ \vee \ g \notin normal_viewids(p), \\ |\max_{\mathcal{X}}(queue_x[g]).us + queue_u[g][1..(next_safe[p,g]-1)]| & \textit{otherwise.} \end{cases} \\
\textit{else } \mathcal{X}(p, prev_view(p,g))
\end{array}}$$

Notice that $\mathcal{X}(p, g)$ is well-defined because the smallest possible view, g_0, is established at all processes.

The definition of $\mathcal{X}(p, g)$ corresponds to our understanding how the algorithm operates. First, if a process p has never succeeded in establishing a view g, then it has never modified its *expertise_level*, *updates* sequence, and *safe_to_update* index in that view, which explains why its expertise in that view is defined as its own expertise in the preceding to g view. Invariant 7.20 extends this case in stating that the expertise of a process p in a view g is the expertise of p in its last established view.

Second, if process p has succeeded in establishing view g, then the three components of p's expertise are defined as follows: The expertise level of p in view g is either g itself if the view is primary, or it is the expertise that p has acquired as a result of the expertise-exchange process in g (see $\texttt{gprcv}(x)_{p',p}$). The sequence of updates at p is defined as the sequence that p has adopted as a result of the expertise-exchange process in g (see $\texttt{gprcv}(x)_{p',p}$) extended with the update requests that p has received during its participation in g (see $\texttt{gprcv}(c,u)_{p',p}$). Notice that, if g is a non-primary view, then the sequence at p is only the adopted sequence since there are no update requests that p has received in that view. Finally, there are the following three cases for the *safe_to_update* index at p: (a) If g is a non-primary view, then the index is just that which p has adopted as a result of the expertise-exchange process in g (see $\texttt{gprcv}(x)_{p',p}$); (b) If g is a primary view, but is not normal in p (i.e., p has not received safe notifications for all expertise messages), then the index is the same as in the previous case; (c) If g is both primary and normal, then the value of the index is defined as the length of the sequence of updates that p has adopted as a result of the expertise-exchange process in g (see $\texttt{safe}(x)_{p',p}$) plus the number of safe notifications for the update requests delivered to p during its participation in g (see $\texttt{safe}(c,u)_{p',p}$).

The following invariant expresses two basic properties of $\mathcal{X}$.

Invariant 7.20 *For all $p \in P$ and for all $g \in created\_views(p)$,*

1. $g > current\_viewid[p] \Rightarrow \mathcal{X}(p,g) = \mathcal{X}(p, current\_viewid[p])$
2. $\mathcal{X}(p,g) = \mathcal{X}(p, last\_established\_viewid(p,g))$

Proof 7.20: Follows immediately from the total ordering of views in *created_views*, and from the definition of $\mathcal{X}$. ■

Having defined the derived expertise $\mathcal{X}$, we now have to accomplish the following two steps. First, we have to show that the derived expertise $\mathcal{X}$ indeed corresponds to the real expertise of each server. For this purpose, we prove that, in any reachable state of the system, the value of the derived expertise taken at the current view of any server is the same as the real expertise of that server. This result will allow us to extend properties of the derived expertise to those of the real expertise. Second, we have to study various properties of the derived expertise, with a goal of identifying a precise relationship among expertise possessed by different servers in different views. Reaching this goal will allow us to prove important consistency properties of our algorithm.

7.4.2 Correspondence between Derived and Real Expertise

The following invariant shows the correspondence between *p.expertise_max*, a state variable used in expertise-exchange process to keep track of the running expertise maximum, and the messages on the *VS queue*.

Invariant 7.21 *For all $p \in P$, the value of $p.expertise\_max$ is equal to*

$$\max_{\mathcal{X}} \left(expertise\_max_0, queue_x[p.view.id][1..(next[p, p.view.id] - 1)] \right)$$

Proof 7.21: The critical actions are `newview` and `gprcv`— both are straightforward. ■

Now, we are able to link real expertise of a server to that defined by function $\mathcal{X}$:

Invariant 7.22 *For all $p \in P$,*

$$\langle p.expertise\_level, p.updates, p.safe\_to\_update \rangle = \mathcal{X}(p, p.view.id).$$

Proof 7.22:

Basis: Straightforward.

Inductive Step: The critical actions are $\texttt{gprcv}(\langle *, u \rangle)_{p',p}$, $\texttt{safe}(\langle *, u \rangle)_{p',p}$, $\texttt{gprcv}(x)_{p',p}$, and $\texttt{safe}(x)_{p',p}$.

$\underline{\texttt{gprcv}(\langle *, u \rangle)_{p',p}}$: Only $p.updates$ is affected. Invariant 7.16 part 1 implies that $p.view.id$ is in $established\_viewids(p)$. The validity of the proposition in the poststate immediately follows from the inductive hypothesis.

$\underline{\texttt{safe}(\langle *, u \rangle)_{p',p}}$: Only $p.safe\_to\_update$ is affected. Invariants 7.5 and 7.16 imply that $p.view.id$ is primary and normal. The step itself is just as above.

$\underline{\texttt{gprcv}(x)_{p',p}}$: The only interesting action is when p establishes its view, i.e., when it receives the last expertise message. Use Invariant 7.21 directly. Then, show that a) $queue_x[p.view.id][1..(next[p, p.view.id] - 1)]$ is the entire $queue_x[p.view.id]$ (Invariant 7.8); b) the view is established but not normal; c) there are no update messages on $queue[p.view.id]$.

$\underline{\texttt{safe}(x)_{p',p}}$: The only interesting action is when p switches to normal mode. Only $safe\_to\_update$ is affected. Show that a) the view is established; and b) if the view is primary then it is normal.

■

7.4.3 Derived Expertise-level vs Real View

The following invariant reveals the relationship between derived expertise level of a process p in a view g and the view g itself.

Invariant 7.23 *For all p and all $g \in created\_views(p)$, $\mathcal{X}(p, g).xl \leq g$. More strongly, if $g \notin primary\_views$ or $g \notin established\_viewids(p)$ then $\mathcal{X}(p, g).xl < g$.*

Proof 7.23:

Basis: In the initial state, g_0 is the only view created. For all p, $g_0 \in created\_views(p)$. The value of $\mathcal{X}(p, g_0).xl$ is equal to g_0, as g_0 is both, established at p and primary.

Inductive Step: A critical action always deals with a single p and a single g. If in the poststate, $g \notin established\_viewids(p)$, then $\mathcal{X}(p, g).xl = \mathcal{X}(p, prev\_view(p, g)).xl$. By the inductive hypothesis, $\mathcal{X}(p, prev\_view(p, g)).xl \leq prev\_view(p, g)$, which, in turn, is less than g. (Notice that, since $g \notin established\_viewids(p)$, $g > g_0$ and $prev\_view$ is defined.) Otherwise, if $g \in established\_viewids(p)$ in the poststate, we should consider whether or not g is a primary view. a) If g is not a primary view, then $\mathcal{X}(p, g).xl = \max_{\mathcal{X}}(queue_x[g]).xl = x.xl$ for some $\langle x, p' \rangle \in queue_x[g]$. By Invariant 7.25, $x.xl = \mathcal{X}(p', prev\_view(p', g)).xl$. By inductive hypothesis, this is less than $prev\_view(p', g)$, which, in turn, is less than g. b) Otherwise, if g is a primary view, then $\mathcal{X}(p, g).xl = g$. ■

The following invariant reveals monotonicity of each server's expertise level.

Invariant 7.24 *For all p and all g_1 and g_2, if $g_1 \leq g_2$, $g_1 \in created\_viewids(p)$ and $g_2 \in created\_viewids(p)$, then $\mathcal{X}(p, g_1).xl \leq \mathcal{X}(p, g_2).xl$.*

Proof 7.24: The case of $g_1 = g_2$ is trivial. Let's consider $g_1 < g_2$.

Basis: In the initial state, g_0 is the only created view. It is established for all p and is primary. Therefore, $\mathcal{X}(p, g_1).xl = \mathcal{X}(p, g_2).xl$.

Inductive Step: The critical actions are those that involve p and g_2. If, in the poststate, $g_2 \notin established\_viewids(p)$, then $\mathcal{X}(p, g_2).xl = \mathcal{X}(p, prev\_view(p, g_2)).xl$, and the inductive hypothesis applies, since $g_1 \leq prev\_view(p, g_2)$. Otherwise, if $g_2 \in established\_viewids(p)$ in the poststate, then by Invariant 7.15 there exists $\langle x, p \rangle \in queue_x[g_2]$. By Invariant 7.25, $x.xl = \mathcal{X}(p, prev\_view(p, g_2)).xl$, which, according to the inductive hypothesis, is greater than $\mathcal{X}(p, g_1).xl$. We are done if we can show that $\mathcal{X}(p, g_2).xl \geq x.xl$.

To show that $\mathcal{X}(p, g_2).xl \geq x.xl$, we consider the following two cases: a) If g_2 is not a primary view, then $\mathcal{X}(p, g_2).xl = \max_{\mathcal{X}}(queue_x[g_2]).xl \geq x.xl$. b) if g_2 is a primary view, then $x.xl \leq \max_{\mathcal{X}}(queue_x[g_2]).xl = \mathcal{X}(p', prev\_view(p', g_2)).xl$. By Invariant 7.23, this is less than or equal to $prev\_view(p', g_2)$, which is less than g_2, the value of $\mathcal{X}(p, g_2).xl$. ■

7.4.4 Recursive Nature of $\mathcal{X}$

The following invariant expresses the fact that an expertise message x sent by p in a view g is the maximum expertise possessed by p during its previous view.

Invariant 7.25 $(x \in pending[p, g] \vee \langle x, p\rangle \in queue[g]) \Rightarrow x = \mathcal{X}(p, prev\_view(p, g))$

Proof 7.25:

Basis: In the initial state, $pending[p, g]$ and $queue[g]$ are empty for all p and g. Thus the proposition is vacuously true.

Inductive Step: The critical actions are $\texttt{gpsnd}(x)_p$, and $\texttt{vs\_order}(x, p, g)$.

$\underline{\texttt{gpsnd}(x)_p}$: This action appends x to $pending[p, current\_viewid[p]]$. By a precondition on this action, x is $\langle p.expertise\_level, p.updates, p.safe\_to\_update\rangle$, which, according to Invariant 7.22, is equal to $\mathcal{X}(p, p.view.id)$. Another precondition ensures that process p is in *expertise_broadcast* mode. Invariant 7.17 implies that $p.view.id \notin established\_viewids(p)$. Thus, according to the definition of $\mathcal{X}$, x is $\mathcal{X}(p, prev\_view(p, g))$.

$\underline{\texttt{vs\_order}(x, p, g)}$: Follows immediately from the inductive hypothesis.

■

7.4.5 Consistency of Derived Expertise

In this section, we prove key invariants that express relationship between derived *updates* and *safe* sequences of different servers in different views depending on their derived *expertise_level*. All together, we show the following three results: First, the *updates* sequences of different servers are consistent if their expertise levels are the same. Second, the *safe* sequence of one server is always a prefix of the *updates* sequence of another server with the same or higher expertise level. Finally, the *safe* sequence of a server in a normal primary view contains as a prefix the *safe* sequence of any server with a strictly smaller expertise level.

Proof Outline

Each of these invariants deals with two servers, p_1 and p_2, and two views, g_1 and g_2, such that $g_1 \in created\_viewids(p_1)$ and $g_2 \in created\_viewids(p_2)$. We prove each of them by induction on the upper bound g on g_1 and g_2, rather then by induction on the length of the execution sequence. This type of induction is valid because view identifiers are totally ordered and have a minimum element g_0.

The proof of each of these invariants follows the same pattern: For the basis we show that an invariant is true when g_1 and g_2 are both the initial view. This part is straightforward. For the inductive hypothesis, we suppose that an invariant is true for all g_1 and g_2 strictly smaller than g. For the inductive step, we show that the invariant is true for all g_1 and g_2 smaller than or equal to g.

To show that the invariant is true for all g_1 and g_2 smaller than or equal to g, we study each of the following cases:

1. $g_1 < g$ and $g_2 < g$
2. $g_1 < g$ and $g_2 = g$
3. $g_1 = g$ and $g_2 < g$
4. $g_1 = g$ and $g_2 = g$

The first case is covered directly by the inductive hypothesis. To show that the invariant is true for each of the other three cases, we use the definition and invariants of $\mathcal{X}$ to relate the values of derived variables associated with the view g to those associated with a smaller view. For this purpose, we look at possible subcases, such as whether or not a view is established at a server, a view is normal at a server, and a view is primary. Most of the cases follow the same argument and are straightforward.

However, there are are also unique cases, the proof of which exploits fundamental assumptions about the algorithm, e.g., that any two primary views have common members.

Invariants

The following invariant states that the *updates* sequences of any two servers in any two views are consistent ($\leq\geq$) if their expertise levels are the same. (Recall that two sequences are said to be *consistent* if one is a prefix of another.)

Invariant 7.26 *For all* p_1 *and* p_2*, and all* g_1 *and* g_2 *such that* $g_1 \in created\_views(p_1)$ *and* $g_2 \in created\_views(p_2)$

$$\mathcal{X}(p_1, g_1).xl = \mathcal{X}(p_2, g_2).xl \Rightarrow \mathcal{X}(p_1, g_1).us \leq\geq \mathcal{X}(p_2, g_2).us$$

Proof 7.26: By induction on the upper bound g of g_1 and g_2.

Basis: If $g = g_0$, then $g_1 = g_2 = g_0$ and it is straightforward to show that the proposition is true.

Inductive Step: Assume that the proposition is true for all g_1 and g_2 such that $g_1 < g$ and $g_2 < g$. We want to show that the proposition is true for all g_1 and g_2 such that $g_1 \leq g$ and $g_2 \leq g$.

We consider the following four cases:

1. $g_1 < g$ and $g_2 < g$. The proposition is true by the inductive hypothesis.
2. $g_1 < g$ and $g_2 = g$. We consider the following three subcases:
   (a) $g_2 \notin established\_viewids(p_2)$
   By definition of $\mathcal{X}$,

$$\mathcal{X}(p_2, g_2).xl = \mathcal{X}(p_2, prev\_view(p_2, g_2)).xl$$
$$\mathcal{X}(p_2, g_2).us = \mathcal{X}(p_2, prev\_view(p_2, g_2)).us.$$

Since $prev\_view(p_2, g_2) < g_2 = g$, it follows that the proposition is true by the inductive hypothesis.

(b) $g_2 \in established\_viewids(p_2)$ and $g_2 \notin primary\_views$

$$\begin{aligned}\mathcal{X}(p_2, g_2).us &= \max_{\mathcal{X}}(queue_x[g_2]).us \text{ – definition of } \mathcal{X} \text{ and Invariant 7.5}\\ &= x.us,\end{aligned}$$

for some $\langle x,\ p'\rangle \in queue_x[g_2]$, such that $x.xl = \max_{\mathcal{X}}(queue_x[g_2]).xl$ and $x.us = \max_{\mathcal{X}}(queue_x[g_2]).us$.

$$\begin{aligned} &= \mathcal{X}(p', prev\_view(p', g_2)).us \text{ – Invariant 7.25.}\\ \mathcal{X}(p_2, g_2).xl &= \max_{\mathcal{X}}(queue_x[g_2]).xl \text{ – definition of } \mathcal{X}\\ &= x.xl\\ &= \mathcal{X}(p', prev\_view(p', g_2)).xl.\end{aligned}$$

Since $prev\_view(p', g_2) < g_2 = g$, it follows that the proposition is true by the inductive hypothesis.

(c) $g_2 \in established\_viewids(p_2)$ and $g_2 \in primary\_views$

In this subcase, the antecedent of the proposition cannot be true:

$$\begin{aligned}\mathcal{X}(p_2, g_2).xl &= g_2 \text{ – by definition of } \mathcal{X}\\ &> g_1 \text{ – case assumption}\\ &\geq \mathcal{X}(p_1, g_1).xl \text{ – Invariant 7.23.}\end{aligned}$$

Thus, the proposition is vacuously true.

3. $g_1 = g$ and $g_2 < g$ This case is symmetric to the previous one.

4. $g_1 = g$ and $g_2 = g$

The case when either $g \notin established\_viewids(p_1)$ or $g \notin established\_viewids(p_2)$ is straightforward as it brings us to one of the previous cases (Invariant 7.25). Therefore, we consider the case when

$$g \in established\_viewids(p_1) \ \wedge\ g \in established\_viewids(p_2).$$

There are two subcases depending on whether or not g is a primary view:

(a) $g \notin primary\_views$

By the definition of $\mathcal{X}$, the contrapositive of Invariant 7.5, and Invariant 7.25,

$$\mathcal{X}(p_1, g).us = \max_{\mathcal{X}}(queue_x[g]).us = x_1.us = \mathcal{X}(p'_1, prev\_view(p'_1, g)).us$$
$$\mathcal{X}(p_2, g).us = \max_{\mathcal{X}}(queue_x[g]).us = x_2.us = \mathcal{X}(p'_2, prev\_view(p'_2, g)).us,$$

for some $\langle x_1,\ p'_1\rangle \in queue_x[g]$ and $\langle x_2,\ p'_2\rangle \in queue_x[g]$ such that

$$x_1.xl = x_2.xl = \max_{\mathcal{X}}(queue_x[g]).xl$$
$$|x_1.us| = |x_2.us| = |\max_{\mathcal{X}}(queue_x[g]).us|.$$

Notice that we do not assume that $x_1.us = x_2.us$.

Since

$$\mathcal{X}(p_1, g).xl = \max_{\mathcal{X}}(queue_x[g]).xl = x_1.xl = \mathcal{X}(p'_1, prev\_view(p'_1, g)).xl$$
$$=$$
$$\mathcal{X}(p_2, g).xl = \max_{\mathcal{X}}(queue_x[g]).xl = x_2.xl = \mathcal{X}(p'_2, prev\_view(p'_2, g)).xl,$$

and both $prev\_view(p'_1, g)$ and $prev\_view(p'_2, g)$ are strictly smaller than g, it follows that the proposition is true by the inductive hypothesis.
Notice that, since $|x_1.us| = |x_2.us|$, in addition to consistency of *updates* sequences, we can actually conclude their equality:

Corollary 7.27 *For all p_1, p_2, and g such that $g \notin primary\_viewids$ and $g \in established\_viewids(p_1)$ and $g \in established\_viewids(p_2)$, it follows that*

$$\mathcal{X}(p_1, g).us = \mathcal{X}(p_2, g).us$$

(b)$g \in primary\_views$
By definition of $\mathcal{X}$, $\mathcal{X}(p_1, g).xl = \mathcal{X}(p_2, g).xl = g$ and

$$\begin{aligned}\mathcal{X}(p_1, g).us &= \max_{\mathcal{X}}(queue_x[g]).us + queue_u[g][1..(next[p_1, g] - 1)]\\ &= x_1.us + queue_u[g][1..(next[p_1, g] - 1)]\\ \mathcal{X}(p_2, g).us &= \max_{\mathcal{X}}(queue_x[g]).us + queue_u[g][1..(next[p_2, g] - 1)]\\ &= x_2.us + queue_u[g][1..(next[p_2, g] - 1)],\end{aligned}$$

for some $\langle x_1,\ p_1'\rangle \in queue_x[g]$ and $\langle x_2,\ p_2'\rangle \in queue_x[g]$ such that

$$x_1.xl = x_2.xl = \max_{\mathcal{X}}(queue_x[g]).xl$$
$$|x_1.us| = |x_2.us| = |\max_{\mathcal{X}}(queue_x[g]).us|.$$

As we did in subcase (a), we can show that $x_1.us = x_2.us$. Therefore, since $queue_u[g][1..(next[p_1, g] - 1)]$ and $queue_u[g][1..(next[p_2, g] - 1)]$ are consistent, it follows that the proposition is true.

■

We introduce the following notation for the safe prefix of derived expertise:

Definition 7.3 $\mathcal{X}(p, g).sp = \mathcal{X}(p, g).us[1..\mathcal{X}(p, g).su]$.

The following invariant states that the *safe* sequence of any server is as a prefix of the *updates* sequence of any other server with a higher or the same expertise level. (Recall the discussion in Chapter 5 on page 49 about *safe* sequences acting as a base for future views.) To prove this property, it is helpful to note that a server could have gotten its *safe* sequence directly by participating in a primary normal view, or indirectly by adopting it from another server during an expertise-exchange process. We express this fact as an additional part of this invariant.

Invariant 7.28 *The following two statements are true:*

1. *For all p_1 and g_1 such that $g_1 \in created\_viewids(p_1)$, there exist p_2 and g_2 such that the following four statements are true:*

   (a) $g_2 \in primary\_viewids$ (c) $\mathcal{X}(p_2, g_2).xl \leq \mathcal{X}(p_1, g_1).xl$

   (b) $g_2 \in normal\_viewids(p_2)$ (d) $\mathcal{X}(p_2, g_2).sp = \mathcal{X}(p_1, g_1).sp.$

2. *For all p_1 and p_2, and all g_1 and g_2 such that $g_1 \in created\_views(p_1)$ and $g_2 \in created\_views(p_2)$, it follows that*

$$\mathcal{X}(p_1, g_1).xl \leq \mathcal{X}(p_2, g_2).xl \Rightarrow \mathcal{X}(p_1, g_1).sp \leq \mathcal{X}(p_2, g_2).us.$$

Proof 7.28: By induction on the upper bound g of g_1 and g_2.

Basis: If $g = g_0$, then $g_1 = g_2 = g_0$ and it is straightforward to show that both propositions are true.

Inductive Step: Assume that both propositions are true for all g_1 and g_2 such that $g_1 < g$ and $g_2 < g$. We want to show that both propositions are true for all g_1 and g_2 such that $g_1 \leq g$ and $g_2 \leq g$. We consider each of the two propositions separately.

Part 1. If $g_1 < g$, then the truth of part 1 follows immediately from the inductive hypothesis. Otherwise, if $g_1 = g$, then we consider the following cases:

1. $g_1 \notin \mathit{established\_views}(p_1)$. Then the proposition is true because, according to the definition of $\mathcal{X}$, $\mathcal{X}(p_1, g_1) = \mathcal{X}(p_1, \mathit{prev\_view}(p_1, g_1))$, and the inductive hypothesis applies.

2. $g_1 \in \mathit{established\_views}(p_1)$ and $g_1 \notin \mathit{primary\_views}$.

$$\begin{aligned}\mathcal{X}(p_1, g_1).sp &= \mathcal{X}(p_1, g_1).us[1..\mathcal{X}(p_1, g_1).su] \\ &= x_u.us[1..x_s.su] \\ &= \mathcal{X}(p_u, \mathit{prev\_view}(p_u, g_1)).us[1..\mathcal{X}(p_s, \mathit{prev\_view}(p_s, g_1)).su],\end{aligned}$$

for some $\langle x_u,\ p_u\rangle \in \mathit{queue}_x[g_1]$ and $\langle x_s,\ p_s\rangle \in \mathit{queue}_x[g_1]$ such that

$$x_s.xl \leq x_u.xl = \max_{\mathcal{X}}(\mathit{queue}_x[g]).xl = \mathcal{X}(p_1, g_1).xl.$$

By the inductive hypothesis of part 2, $\mathcal{X}(p_s, \mathit{prev\_view}(p_s, g_1)).sp$ is a prefix of $\mathcal{X}(p_u, \mathit{prev\_view}(p_u, g_1)).us$, and thus,

$$\mathcal{X}(p_1, g_1).sp = \mathcal{X}(p_s, \mathit{prev\_view}(p_s, g_1)).sp$$

Therefore, the inductive hypothesis of part 1 applies, and the proposition is true.

3. $g_1 \in \mathit{established\_views}(p_1)$, $g_1 \in \mathit{primary\_views}$, but $g_1 \notin \mathit{normal\_viewids}(p_1)$. The proof that the proposition is true in this case is very similar to the previous one.

4. $g_1 \in \mathit{established\_views}(p_1)$, $g_1 \in \mathit{primary\_views}$, and $g_1 \in \mathit{normal\_viewids}(p_1)$. The proposition is true because we can take p_2 as p_1 and g_2 as g_1.

Part 2. We consider the following three cases:

1. $g_1 < g$ and $g_2 < g$. The proposition is true by the inductive hypothesis.

2. $g_1 < g$ and $g_2 = g$. We consider the following three subcases:

(a) $g_2 \notin$ *established_viewids*(p_2). Similarly to the corresponding case in the proof of Invariant 7.26.

(b) $g_2 \in$ *established_viewids*(p_2) and $g_2 \notin$ *primary_views*. Similarly to the corresponding case in the proof of Invariant 7.26.

(c) $g_2 \in$ *established_viewids*(p_2) and $g_2 \in$ *primary_views*.

By the inductive hypothesis of part 1, there exists a primary g_s ($g_s \leq \mathcal{X}(p_1, g_1).xl$) and a process p_s such that

$$g_s \in normal\_viewids(p_s) \text{ and } \mathcal{X}(p_1, g_1).sp = \mathcal{X}(p_s, g_s).sp.$$

Since g_s is normal for p_s, it follows from part 2 of Invariant 7.14 that g_s is totally established. From the definition of $\mathcal{X}$ and the inductive hypothesis of part 2, it follows that, for all $p \in GtoS(g_s)$,

$$\mathcal{X}(p_s, g_s).xl = \mathcal{X}(p, g_s).xl = g_s$$
$$\mathcal{X}(p_s, g_s).sp \leq \mathcal{X}(p, g_s).us$$

Since g_s and g_2 are primary views, there exists a process $p_\cap$ that is a member of both views (by the definition of primary views).

Since g_2 is established at p_2, Invariant 7.15 implies that there exists $\langle x_\cap, p_\cap \rangle \in queue_x[g_2]$. By Invariant 7.24, $\mathcal{X}(p_\cap, g_s).xl \leq x_\cap.xl$. Moreover, by definition of $\max_{\mathcal{X}}()$ it follows that $x_\cap.xl \leq \max_{\mathcal{X}}(queue_x[g_2]).xl$ and $x_\cap.us \leq \max_{\mathcal{X}}(queue_x[g_2]).us$.

By definition of $\mathcal{X}$, $\mathcal{X}(p_2, g_2).us$ contains as a prefix $\max_{\mathcal{X}}(queue_x[g_2]).us$, which equals to x_u for some $\langle x_u,\ p_u \rangle \in queue_x[g_2]$ such that $x_u.xl = \max_{\mathcal{X}}(queue_x[g_2]).xl$.

It follows that

$$\mathcal{X}(p_1, g_1).xl \leq x_\cap.xl \leq x_u.xl = \mathcal{X}(p_u, prev\_view(p_u, g_2)).xl,$$

and the inductive hypothesis applies:

$$\mathcal{X}(p_1, g_1).sp \leq \mathcal{X}(p_u, prev\_view(p_u, g_2)).us \leq \mathcal{X}(p_2, g_2).us$$

3. $g_1 = g$ and $g_2 \leq g$. We have to consider the following four subcases:

(a) $g_1 \notin$ *established_viewids*(p_1). Straightforward.

(b) $g_1 \in \mathit{established\_viewids}(p_1)$ and $g_1 \notin \mathit{primary\_views}$
Straightforward. Similarly to the corresponding case in part 1.

(c) $g_1 \in \mathit{established\_viewids}(p_1)$ and $g_1 \notin \mathit{normal\_views}(p_1)$.
Straightforward. Similarly to the corresponding case in part 1.

(d) $g_1 \in \mathit{established\_viewids}(p_1)$, $g_1 \in \mathit{primary\_views}$, and $g_1 \in \mathit{normal\_views}(p_1)$.
Straightforward. Directly from the definition of $\mathcal{X}$ and part 13 of Invariant 7.1.

■

Finally, the following invariant states that the *safe* sequence of any server in a primary and normal view contains as prefixes *safe* sequences of other servers with strictly smaller expertise levels.

Invariant 7.29 *For all p_1 and p_2, and all g_1 and g_2 such that $g_1 \in \mathit{created\_views}(p_1)$ and $g_2 \in \mathit{created\_views}(p_2)$,*

$$\left.\begin{array}{l} g_2 \in \mathit{primary\_viewids} \\ g_2 \in \mathit{normal\_viewids}(p_2) \\ \mathcal{X}(p_1, g_1).xl < \mathcal{X}(p_2, g_2).xl \end{array}\right\} \Rightarrow \mathcal{X}(p_1, g_1).sp \leq \mathcal{X}(p_2, g_2).sp$$

Proof 7.29: Straightforward. The proof is very similar to that of Invariant 7.28. ■

7.5 Consistency of *updates*, *safe* and *done* sequences

In this section, we extend Invariants 7.26, 7.28, and 7.29 of the previous section, which express consistency properties of the derived expertise, to the real expertise of servers. Then, we use this result to obtain a number of high-level consistency properties, among which there are Invariants 6.1 and 6.2 (Chapter 6, page 53).

Invariant 7.30 *For all processes p_1 and p_2,*

1. *If their expertise levels are the same, then their updates sequences are consistent.*

$$p_1.\mathit{expertise\_level} = p_2.\mathit{expertise\_level} \Rightarrow p_1.\mathit{updates} \leq\geq p_2.\mathit{updates}$$

2. If expertise level of p_1 is less than or equal to that of p_2, then $p_1.safe$ is a prefix of $p_2.updates$.

$$p_1.expertise\_level \leq p_2.expertise\_level \Rightarrow p_1.safe \leq p_2.updates$$

3. If expertise level of p_1 is less than or equal to that of p_2, and p_2 is in normal mode of a primary view, then $p_1.safe$ is a prefix of $p_2.safe$.

$$\left.\begin{array}{l}(p_1.expertise\_level < p_2.expertise\_level)\\(p_2.mode = normal)\\(p_2.view \in primary\_views)\end{array}\right\} \Rightarrow p_1.safe \leq p_2.safe$$

Proof 7.30: The proof follows immediately from Invariant 7.22, which states the correspondence between state variables of T and the derived expertise $\mathcal{X}$, and from Invariants 7.26, 7.28, and 7.29, which restate Invariant 7.30 in terms of the derived expertise $\mathcal{X}$. ■

Corollary 7.31 *For all processes p_1 and p_2, safe prefixes of their updates sequences are consistent.*

Proof 7.31: According to part 2 of Invariant 7.30, *safe* prefix of a node is a prefix of *updates* sequence of a node with a greater or equal *expertise_level*. Without loss of generality assume that $p_1.expertise\_level$ is less than or equal to $p_2.expertise\_level$. Then, $p_1.safe$ is a prefix of $p_2.updates$. Applying the same part of Invariant 7.30 just to p_2, we have $p_2.safe$ is a prefix of $p_2.updates$. Therefore, $p_1.safe$ and $p_2.safe$ are consistent. ■

Corollary 7.32 *For any reachable state t, if (t, π, t') is a transition of T, then for all $p \in P$, $t[p].safe \leq t'[p].safe$.*

Proof 7.32: The only interesting actions in the inductive proof are the last $\texttt{gprcv}(x)_{p',p}$ and $\texttt{safe}(x)_{p',p}$. The proof is straightforward. ■

For simplicity, we introduce the following notation:

Definition 7.4 *Let $p.done$ denote $p.updates[1..p.last\_update]$.*

Invariant 7.33 *For all $p \in P$, $p.done \leq p.safe$.*

Proof : Straightforward proof by induction on the length of the execution sequence. Relies directly on corollary 7.32. ■

Corollary 7.34 *For any reachable state t, if (t, π, t') is a transition of T, then for all $p \in P$, $t[p].done \leq t'[p].done$.*

Proof 7.34: Follows directly from Invariant 7.33 and Corollary 7.32. ■

Top-level Invariants 6.1 and 6.2 follow immediately from the corollaries above. Here are the restatements of these top-level invariants:

Corollary 7.35 $p.done \leq p.safe \leq p.updates$.

Corollary 7.36 *In any reachable state t of T, for any p_1 and p_2, $p_1.done$ and $p_2.done$ are consistent.*

7.6 Coherence of Local Buffers

Two of the top-level invariants (Invariants 6.3 and 6.4) in Chapter 6 express coherence properties of the server's *map* and *pending* buffers. One of the properties states that, if an update request appears as safe at some server but has not been yet executed by its native server, then it is still reflected in the native server's *map* and *pending* buffers. The other property states that a server can have at most one unexecuted native update request on its *updates* sequences. The non-triviality of the first property comes from the fact that servers can remove update requests from their *pending* buffers not only as a result of executing them, but also as a result of an expertise-exchange process that moves these update requests for reprocessing. The second property is straightforward, but one of its proof cases has to deal with the *updates* sequence being adopted from another server during expertise-exchange process.

As is, these properties do not immediately allow for proofs by induction because their critical actions are not grounded in inductive hypotheses. That is, for example, when an update request is delivered to a server, there is no direct way to argue what the prestate value of the *updates* sequence at that server is.

In order to prove these properties, we generalize them to include statements that allow us to track the state of the system throughout its execution: from the time that messages are submitted to the group communication layer, while they are in transit, and until they are finally delivered to the replication layer.

Invariant 7.37 *For all clients $c \in C$, let p stand for $c.proc$. Then, the following statements are true:*

1. There exists at most one element of the form $\langle c,\, u\rangle$ in the following sequences: $pending[p, g]$, $queue_u[g][next[p, g]..]$, and $p.updates[(p.last\_update + 1)..]$, where g ranges through all the views that are at least as large as the last normal primary view of p.

2. Moreover, if there is one such element, then $\langle c,\, u\rangle \in p.map$ and $c \in p.pending$.

Proof 7.37: By induction on the length of the execution sequence.

Basis: Straightforward since all buffers are empty in the initial state.

Inductive Step: We consider the following critical actions:

$\underline{\texttt{gpsnd}(c,u)_p\text{:}}$ This action adds element $\langle c,\ u\rangle$ to $pending[p,g]$. For part one, we have to show that, in the prestate, there are no such elements in the considered sequences. This immediately follows from the precondition $c \notin p.pending$ and the contrapositive of part two of the inductive hypothesis. For part two, we have to show that this element in reflected in its native *map* and *pending* buffers. This follows immediately from the precondition $\langle c,\ u\rangle \in p.map$ and the effect $p.pending \leftarrow p.pending\ \cup\ c$.

$\underline{\texttt{vs\_order}(\langle c,\ u\rangle, p, g)\text{:}}$ This action moves $\langle c,\ u\rangle$ from $pending[p,g]$ to $queue_u[g][next[p,g]..]$. The proposition follows from the inductive hypothesis.

$\underline{\texttt{gprcv}(c,u)_{p,p}\text{:}}$ This action moves $\langle c,\ u\rangle$ from $queue_u[g][next[p,g]..]$ to $p.updates[(p.last\_update+1)..]$. The proposition follows from the inductive hypothesis.

$\underline{\texttt{update}(c,u)_p\text{:}}$ This action removes $\langle c,\ u\rangle$ from $p.pending$, but it also advances $p.last\_update$ to cover this element. Therefore, the proposition is vacuously true.

$\underline{\texttt{gprcv}(x)_{p',p}\text{:}}$ The only action of interest is the last action of the expertise exchange process, when p adopts the *updates* sequence of an expert. We have to show that there can be at most one element of the type $\langle c,\ u\rangle$ on that sequence, and if there is one, that it is reflected in p's local buffers. The proposition follows from the inductive hypothesis, once we notice that any adopted sequence of updates contains as a prefix the safe prefix of p in its last normal primary view (Invariant 7.30); therefore, all the remaining elements of this sequence come after that view and are covered by the inductive hypothesis.

$\underline{\texttt{safe}(x)_{p',p}\text{:}}$ The only action of interest is the last action of the expertise exchange process in a primary view $p.view$ that removes $\langle c,\ u\rangle$ from $p.pending$. We want to show that there are no elements of the type $\langle c,\ u\rangle$ in the sequences listed in the proposition. This action makes $p.view$ be the last normal primary view of p (Invariant 7.18). Thus, there are only three sequences that we need to consider: $pending[p, p.view.id]$, $queue_u[p.view.id][next[p, p.view.id]..]$, and $p.updates[(p.last\_update+1)..]$. The proposition is true because p has not had a chance yet to send any update requests in its present view (by the contrapositive of part 2 of Invariant 7.16) and because $\langle c,\ u\rangle$ is removed from $p.pending$ only if it is not on the $p.updates$ sequence of p (according to the condition in the code).

■

Top-level Invariants 6.3 and 6.4 follow straightforwardly from Invariant 7.37.

7.7 Coherence of Local Database Replicas

In this section, we show that the state of the local database replica at any server corresponds to the sequence of executed update requests at that server. We use this fundamental fact in the next section when we prove coherence of query processing.

Invariant 7.38 *For all* $p \in P$, $p.db = compose(p.updates.u[1..p.last\_update])(db_0)$.

Proof 7.38:

Basis: In the initial state, $p.db = db_0$, $p.updates.u[1..p.last\_update] = [\,]$, and $compose([\,])(db_0) = db_0$. Therefore, the proposition holds.

Inductive Step: The critical actions are $\texttt{gprcv}(c, u)_{p',p}$, $\texttt{gprcv}(x)_{p',p}$, and $\texttt{update}(c, u)_p$.
$\underline{\texttt{gprcv}(c, u)_{p',p}}$: The left side of the proposition is unaffected. The right side is also unaffected because of Corollary 7.35. Therefore, the proposition is true.
$\underline{\texttt{gprcv}(x)_{p',p}}$: Only the last action of expertise-exchange process is of interest. The left side of the proposition is unaffected. The right side of the proposition is also unaffected because of Corollary 7.34 and the fact that $p.last\_update$ is unchanged as a result of this action.
$\underline{\texttt{update}(c, u)_p}$: Both sides change. The proposition follows immediately from the inductive hypothesis and the definition of *compose*. Indeed,

$t'[p].db = u(t[p].db)$, by an effect of $\texttt{update}(c, u)_p$;

$= t[p].updates[t[p].last\_update + 1].u(t[p].db)$, by precondition on $\texttt{update}(c, u)_p$;

$= t[p].updates[t[p].last\_update + 1].u(compose(t[p].updates.u[1..t[p].last\_update]))$, by the inductive hypothesis;

$= compose(t[p].updates.u[1..(t[p].last\_update + 1)])$, by definition of composition;

$= compose(t'[p].updates.u[1..(t'[p].last\_update)])$, by an effect of $\texttt{update}(c, u)_p$.

■

7.8 Coherence of Query Processing

We conclude this chapter by proving top-level Invariant 6.5 which expresses coherence properties of query processing. Parts (a) and (b) of this multipart invariant claim that, when query answers arrive within the same view of their submission, the corresponding query requests are still reflected in the *map* and *pending* buffers of their recipients; This is similar to what we established before for update requests, but is simpler because it concerns only single views. Parts (c), (d), and (e) claim that query answers are correct in terms of both, the sequences of update requests at their recipients and the last database states that their clients have witnessed.

As is, Invariant 6.5 does not immediately allow for a proof by induction because its critical actions are not grounded in inductive hypotheses. In order to prove this invariant, we generalize it to express coherence of query processing in its intermediate stages (Invariants 7.40 and 7.42).

First, we state the following two auxiliary invariants:

The following invariant expresses a perhaps obvious fact that *query_counter* of any process is equal to the number of queries delivered to this process in its current view.

Invariant 7.39 $p.query\_counter = |queue_q[p.view.id][1..(next[p, p.view.id] - 1)]|$

Proof 7.39: Straightforward induction. ■

The following invariant expresses the fact that for each entry in the $p.queries$ buffer of $VStoD_p$ there is a corresponding entry in the $queue[p, p.view.id]$ buffer of VS.

Invariant 7.40 *For any p, let* $\langle g,\ S\rangle$ *denote* $p.view$*. Then the following statements are true:*

1. *If* $\langle c, q, l\rangle \in p.queries$ *then there exists* i *such that the following two statements are true:*

   (a) $\langle\langle c, q, l\rangle,\ c.proc\rangle = queue[g][i]$

   (b) $rank(p, S) = i \mod |S|$

2. *If* $\langle c, a, l\rangle \in p.queries$ *then there exists* q, l', *and* i *such that the following four statements are true:*

   (a) $\langle\langle c, q, l'\rangle,\ c.proc\rangle = queue[g][i]$ *and* $l' \leq l$

   (b) $rank(p, S) = i \mod |S|$

   (c) $a = q(compose(p.updates.u[1..l])(db_0))$

   (d) $l \leq p.last\_update$

Proof 7.40: Straightforward induction. Relies on Invariants 7.38 and 7.39. ■

Corollary 7.41 *In any reachable state of the system, for any* p *and* g, *if* $\langle c, a, l\rangle \in p.queries$ *or* $\langle c, a, l, g\rangle \in in\text{-}transit_{p,c.proc}$, *then* $l \leq \mathcal{X}(p, g).su$.

Finally, we are able to state and prove an invariant that generalizes Invariant 6.5 to intermediate stages of query processing.

Invariant 7.42 *For any client* $c \in C$, *let* p *denote* $c.proc$, g *denote* $p.view.id$, *and* l *denote* $p.last(c)$. *Then, the validity of one of the following two statements:*

1. $\langle c, q, l\rangle \in pending[p, g]$

2. $\langle\langle c, q, l\rangle,\ p\rangle = queue[g][i]$ *for some* $i \in \mathcal{N}$ *and either one of the following statements is true for* p' *such that* $rank(p', GtoS(g)) = i \mod |GtoS(g)|$*:*

   (a) $next[p', g] \leq i$

   (b) $\langle c, q, l\rangle \in p'.queries$

   (c) $\langle c, a, l'\rangle \in p'.queries$
   where $a = q(compose(p.updates.u[1..l'])(db_0))$ *and* $l \leq l' \leq p'.last\_update$

   (d) $\langle c, a, l', g\rangle \in in\text{-}transit_{p',p}$
   where $a = q(compose(p.updates.u[1..l'])(db_0))$ *and* $l \leq l' \leq p'.last\_update$

implies the validity of the following two statements: $\langle c, q\rangle \in p.map$ *and* $c \in p.pending$.

Proof 7.42: Straightforward induction. Relies on Invariants 7.38, 7.39, and 7.40 and Corollary 7.36. ■

The proof of top-level Invariant 6.5 follows straightforwardly from this invariant.

Chapter 8

Correctness of Load Balancing

In the previous two chapters, we proved that automaton $\overline{T}$ implements automaton $\overline{S}$ in the sense of trace inclusion. In this chapter, we accomplish the following three tasks:

First, we complete the proof of Lemma 5.1, which states that automaton $\overline{T'}$ implements automaton $\overline{T}$ in the sense of trace inclusion. This result implies that automaton $\overline{T'}$ also implements automaton $\overline{S}$ in the sense of trace inclusion (see Theorem 6.2).

Once we establish partial correctness of automaton T', we prove a liveness-related claim that its servers are always enabled to advance their database states sufficiently far to be able to process queries that are assigned to them.

Finally, we argue that, in any given view, each server is assigned the same (plus or minus one) number of query requests as any other server of that view.

8.1 Correctness of T'

This section completes the proof that automaton $\overline{T'}$ implements automaton $\overline{S}$ in the sense of trace inclusion. Recall from Chapter 5 pages 49–51 that what is left to be shown is that the identity mapping from the reachable states of $\overline{T'}$ to the reachable states of $\overline{T}$ holds when $\overline{T'}$ takes a `newview` action.

When $\overline{T'}$ executes a transition with a $\texttt{newview}(v)_p$ action, the corresponding execution sequence of $\overline{T}$ contains $p.\Delta$ actions of the form $\mathtt{safe}(c,u)_{p',p}$, possibly separated by actions of the form $\mathtt{safe}(c,q,l)_{p',p}$ (which have no effect on the algorithm), followed by a $\mathtt{newview}(v)_p$ action. The fact that the simulation holds follows immediately once we show that this execution sequence, call it α, is possible in $\overline{T}$.

In order to show that α is possible, we have to show that $p.\Delta$ actions of the form $\mathtt{safe}(c,u)_{p',p}$ are enabled in VS. In other words, if $\langle g,\ S\rangle$ denotes the current view of p prior to the newview action, then we have to show that for each $r \in S$ the sequence $queue[g][next\_safe[p,g]..(next[r,g]-1)]$ contains $p.\Delta$ update messages.

However, this complicated condition is true, if we can show that there exists a member $p' \in S$ such that the sequence $queue[g][next\_safe[p,g]..(next\_safe[p',g]-1)]$ contains $p.\Delta$ update messages. This is valid since part 13 of Invariant 7.1 states that the *next* index of any member in any given view is bounded from below by the *next_safe* indices of all the members in that view.

In order to show this result, we proceed with the following three steps: We first define a collection of history variables that keeps track of the largest database states witnessed by each process p in each view g. We then prove that in any reachable state t of $\overline{T}$ that corresponds to a reachable state of $\overline{T'}$, if $t[p].\Delta$ is greater than zero, then the largest database state known to the clients of p has been witnessed by p during its current view. Finally, we show that, if $t[p].\Delta$ is greater than zero, then there exists a process p' such that the sequence $queue[g][next\_safe[p,g]..(next\_safe[p',g]-1)]$ contains $p.\Delta$ update messages.

8.1.1 History Variable

Recall the definition of the derived variables $p.last\_max$ and $p.\Delta$ from Definition 5.2 on page 50. Variable $p.last\_max$ represents an index to the largest database state known to the clients of p. Variable $p.\Delta$ represents the difference between $p.last\_max$ and $p.safe\_to\_update$ if this difference is positive, and zero otherwise.

We define a history variable, $last\_max[p,g]$ for each p and g, to keep track of the largest database state witnessed by p during its participation in view g. The initial value of each of these variables is 0. The history variables are set in an obvious way at the places where variable *last* is modified:

$\texttt{update}(c, u)_p$
Pre: $last\_update < safe\_to\_update$
$\langle c,\ u\rangle = updates[last\_update + 1]$
Eff: $last\_update \leftarrow last\_update + 1$
$db \leftarrow u(db)$
if $(c.proc = p)$ **then**
$pending \leftarrow pending\ -\ c$
$map(c) \leftarrow ok$
$last(c) \leftarrow last\_update$
$\boxed{\textbf{if}\ \ (last\_update > last\_max[p, p.view.id])\ \ \textbf{then}\ \ last\_max[p, p.view.id] \leftarrow last\_update}$

$\texttt{ptprcv}(c, a, l, g)_{p',p}$
Eff: **if** $(g = view.id\ \wedge\ c.proc = p)$ **then**
$pending \leftarrow pending\ -\ c$
$map(c) \leftarrow a$
$last(c) \leftarrow l$
$\boxed{\textbf{if}\ \ (l > last\_max[p, p.view.id])\ \ \textbf{then}\ \ last\_max[p, p.view.id] \leftarrow l}$

8.1.2 Properties

We now present a number of properties involving the values of $last\_max[p, g]$.

First, we prove that, in any reachable state t of $\overline{T}$ that corresponds to a reachable state of $\overline{T'}$, $t[p].\Delta$ can be greater than zero only due to the database indices that p has witnessed during its current view. This result is stated in Corollary 8.2, which follows from Invariant 8.1 and Lemma 8.1 stated below.

The following invariant states that the value of $p.safe\_to\_update$ in a reachable state of $\overline{T'}$ is greater than the values of $last\_max[p, g]$ for all previous views g of p.

Invariant 8.1 *In any reachable state of $\overline{T'}$, for all p in P and all g in G,*

$$\text{if } g < p.view.id, \text{ then } last\_max[p, g] \le p.safe\_to\_update.$$

Proof : Straightforward induction. The only interesting action of $\overline{T'}$ is `newview`. Its effect makes sure that $p.safe\_to\_update$ is at least as large as $last\_max[p, g]$. ■

The following is an auxiliary lemma that states that throughout an execution of $\overline{T}$, the values of $last(c)$ are non-decreasing.

Lemma 8.1 *For any reachable state t of $\overline{T}$, if (t, π, t') is a transition of $\overline{T}$, then, for all p in P and all c in C such that $c.proc = p$, it follows that $t[p].last(c) \le t'[p].last(c)$.*

Proof 8.1: Straightforward induction. The critical actions are `update` and `ptprcv`. The inductive step for the former is straightforward. The inductive step for the latter follows immediately from part (d) of Invariant 6.5. ■

From Invariant 8.1 and Lemma 8.1, it follows that, in any reachable state t of $\overline{T}$ corresponding to a reachable state of $\overline{T'}$, if $t[p].\Delta > 0$, then $t[p].last\_max = t.last\_max[p, t[p].view.id]$. Therefore, the following corollary is true:

Corollary 8.2 *In any reachable state t of $\overline{T}$ corresponding to a reachable state of $\overline{T'}$,*

$$\text{if } t[p].\Delta > 0, \text{ then } t[p].\Delta = t.last\_max[p, t[p].view.id] - t[p].safe\_to\_update.$$

The following invariant states that $last\_max[p, g]$ can surpass $\mathcal{X}(p, g).su$ only when g is *primary* and *normal* at p. Moreover, there always exists a member p' of $GtoS(g)$ such that $last\_max[p, g]$ is bounded from above by $\mathcal{X}(p', g).su$.

Invariant 8.3 *In any reachable state of $\overline{T}$, for all p in P and all g in G, if $\mathcal{X}(p, g).su < last\_max[p, g]$, then*

1. $g \in normal\_viewids(p)$
2. $g \in primary\_viewids$
3. $\exists\, p' \in GtoS(g)\ .\ last\_max[p, g] \leq \mathcal{X}(p', g).su$

Proof : Straightforward induction. The only interesting action is `ptprcv`. The proof relies on Corollary 7.41 and Invariant 7.42. ■

From this invariant it is straightforward to show the following corollary:

Corollary 8.4 *In any reachable state of $\overline{T}$, for all p and all g, if $\mathcal{X}(p, g).su < last\_max[p, g]$, then there exists $p' \in GtoS(g)$ such that*

$$|queue_u[g][next\_safe[p, g]..(next\_safe[p', g] - 1)]| \geq (last\_max[p, g] - \mathcal{X}(p, g).su)$$

Proof : Straightforward. Relies on Invariant 8.3 ■

Finally, we are able to conclude that the identity refinement from $\overline{T'}$ to $\overline{T}$ holds when $\overline{T'}$ executes a transition with a `newview` action.

Lemma 8.2 *If t is a reachable state of $\overline{T}$ corresponding to a reachable state of $\overline{T'}$, then the corresponding execution sequence α is enabled and the mapping is preserved.*

This completes the proof of Lemma 5.1 that $\overline{T'}$ implements $\overline{T}$ in the sense of trace inclusion. Therefore, Theorem 6.2 holds, and the implementation automaton T' is partially correct with respect to the specification automaton S.

8.2 Properties of T'

Notice that all invariants of $\overline{T}$ are the invariants of $\overline{T'}$. This is true since $\overline{T'}$ implements $\overline{T}$ in the sense of trace inclusion and since the refinement mapping from $\overline{T'}$ to $\overline{T}$ is identity.

Using a very similar argument to the one used in the previous section, we can show that servers of $\overline{T'}$ are always enabled to advance their database states sufficiently far to be able to process the queries assigned to them.

Finally, we observe that the number of query requests assigned to each particular server during its participation in a certain view is the same (plus or minus one) as the number of query requests assigned to any other server during its participation in that view. This property follows immediately from the fact that each server sees a prefix of the total order on query requests delivered within the same view and from the properties of the $\mod$ function.

Chapter 9

Conclusions and Future Work

Group communication services provide powerful abstractions upon which it is possible to construct highly fault-tolerant applications, such as replication and load-balancing systems that tolerate partitioning and merging of the underlying network. Unfortunately, the lack of formalism at the level of group communication services had impeded the development of systematic and formal approaches for the design of applications that use these services.

In an effort to remedy the lack of good specifications for group communication services, Fekete, Lynch, and Shvartsman recently proposed a simple automaton specification for a view-synchronous group communication service [13].

In this thesis, we have used this specification as a building block to formally model an intricate and important application that integrates replication and load-balancing, guarantees sequentially consistent behavior, and tolerates network partitioning.

Using the I/O automaton model of Lynch and Tuttle, we have presented a specification and an implementation automata for this service, and have given a hierarchical proof that the latter implements the former in the sense of trace inclusion.

The specification automaton defines a sequentially consistent data service, in which update requests are performed with respect to the latest data states. Query requests, on the other hand, are performed with respect to the data states that are not necessarily the latest ones, but that are at least as advanced as the last states witnessed by the queries' clients.

The implementation automaton is composed of a collection of identical automata specifying a state machine of each server, the *VS* specification automaton for a group communication service, and a collection of automata specifying reliable reordering channels between any two servers. The implementation automaton models a replicated service in which update requests are processed in the same order at all servers, thus guaranteeing mutual consistency of data replicas, and in which query requests are processed at single servers determined by a load-balancing strategy which equalizes the number of queries assigned to each member of the same group.

The hierarchical proof of correctness establishes that all traces of the implementation automaton are valid traces of the specification automaton. This proof relies on a number of high-level invariants, which we have proved assertionally. The proof of these invariants is based on an interesting approach: we have invented a derived function $\mathcal{X}$ that expresses recursively the highest state reached by each server in each group. In a sense, this function presents a law according to which the replication part of the algorithm operates. As seen in Section 7.4, the recursive nature of this function makes proofs by induction easy: proving an inductive step simply involves unwinding a recursive step of the derived function $\mathcal{X}$.

We have also proved a liveness-related claim that the load-balancing part of the algorithm is *uniform* and *non-blocking*. For uniformity, we have shown that each member of a group is assigned the same number of query requests as any other member of that group. For non-blockage, we have shown that the servers are always able to sufficiently advance the state of their replicas in order to process the queries assigned to them.

In addition to presenting a novel algorithm that integrates replication and load-balancing, this work has the following two important implications:

First, it demonstrates that *VS* specifies a powerful service capable of supporting important applications. Moreover, it demonstrates that the style of the *VS* specification is formal enough to support rigorous modeling of applications and is simple enough to provide intuitive understanding of the group communication service. We note that the *VS* specification can be easily extended to include other potentially useful properties. Based on our experience with the load-balancing part of the presented algorithm, we identify the following two useful extensions to *VS*: a version of multicast without safe notifications, and a version of within-view unicast.

Second, our work exhibits a number of general approaches that can be used to formally model other replication and load-balancing algorithms based on formally specified group communication services.

In order to keep the discussion and the correctness proof tractable, we have chosen to omit secondary functionality from our algorithm, such as support for non-blocking clients and for updates that return data values (instead of ok). However, the algorithm and the proof can be straightforwardly extended to accommodate this functionality.

Another straightforward extension to the algorithm would be to implement *propagation by eventual path* [2, 1], a strategy in which servers of non-primary views share their update requests with the other members of their views. As a result of this sharing, update requests can reach primary groups and be executed faster than if they remained known only to their original servers. This strategy makes more sense when it is less important to notify clients that their requests have been performed than to actually perform them. In particular, this strategy makes less sense when clients block, which is why we did not implement it in our algorithm.

Regarding future work, an important direction would be to investigate liveness of the presented algorithm. While this thesis has dealt solely with the safety properties of the algorithm, it is important to consider its performance and fault-tolerance properties, which are stated conditionally to hold in periods of good behavior of the underlying network. In particular, it would be interesting to analyze conditions under which the load-balancing part of the algorithm performs adequately compared to other load-balancing schemes.

Another important direction for future research would be to investigate the suitability of multicast group communication systems for other, possibly adaptive, load-balancing schemes. These schemes could take advantage of powerful multicast primitives provided by the underlying group communication service to yield good scheduling strategies. To offset the extra computation and communication costs associated with multicast, these schemes could, for example, pack several tasks on one message.

Finally, it would be interesting to adopt the algorithm presented in this thesis to a dynamically evolving set of processes by using a variant of a dynamic view-oriented group communication service presented in [25], instead of VS. This adaptation would be similar to the way a static totally-ordered broadcast application of [13] has been adopted to its dynamic version in [25].

Bibliography

[1] Y. Amir, D. Dolev, P. Melliar-Smith, and L. Moser. Robust and efficient replication using group communication. Technical Report 94-20, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994.

[2] Yair Amir. *Replication using Group Communication over a Partitioned Network*. PhD thesis, The Hebrew University of Jerusalem, Jerusalem, Israel, 1995.

[3] O. Babaoglu, R. Davoli, L. Giachini, and P. Sabattini. The inherent cost of strong-partial view-synchronous communication. *Lecture Notes in Computer Science*, 972:72–86, 1995.

[4] Kenneth P. Birman. *Building Secure and Reliable Network Applications*. Manning Publications Co., Greenwich, CT, 1996.

[5] Kenneth P. Birman and Robbert van Renesse, editors. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.

[6] Tushar Deepak Chandra, Vassos Hadzilacos, Sam Toueg, and Bernadette Charron-Bost. On the impossibility of group membership. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*, pages 322–330, New York, USA, May 1996. ACM.

[7] G. V. Chockler, N. Huleihel, and D. Dolev. An adaptive totally ordered multicast protocol that tolerates partitions. In *Proceedings of the 17h Annual ACM Symposium on Principles of Distributed Computing (PODC '98)*. ACM, 1998. to appear.

[8] Gregory V. Chockler. An adaptive totally ordered multicast protocol that tolerates partitions. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1997.

[9] F. Cristian. Group, majority, and strict agreement in timed asynchronous distributed systems. In *Proceedings of the Twenty-Sixth International Symposium on Fault-Tolerant Computing*, pages 178–189, Washington, June25–27 1996. IEEE.

[10] Danny Dolev and Dalia Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, April 1996.

[11] Danny Dolev, Dalia Malki, and Ray Strong. A framework for partitionable membership service. Technical Report TR94-6, Department of Computer Science, Hebrew University, 1994.

[12] P. D. Ezhilchelvan, R. A. Macêdo, and S. K. Shrivastava. Newtop: A fault-tolerant group communication protocol. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, pages 296–306, Los Alamitos, CA, USA, May30 June–2 1995. IEEE Computer Society Press.

[13] Alan Fekete, Nancy Lynch, and Alex Shvartsman. Specifying and using a partionable group communication service. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 53–62, Santa Barbara, California, 21–24 August 1997.

[14] Alan Fekete, Nancy Lynch, and Alex Shvartsman. Specifying and using a partionable group communication service. extended version, available at http://theory.lcs.mit.edu/tds, 21–24 August 1997.

[15] Roy Friedman and Robbert van Renesse. Strong and weak virtual synchrony in horus. Technical Report TR95-1537, Cornell University, Computer Science Department, August 24, 1995.

[16] Roy Friedman and Alexey Vaysburd. Implementing replicated state machines over partitionable networks. Technical Report TR96-1581, Cornell University, Computer Science, April 17, 1996.

[17] Roy Friedman and Alexey Vaysburd. High-performance replicated distributed objects in partitionable environments. Technical Report TR97-1639, Cornell University, Computer Science, July 16, 1997.

[18] Idit Keidar. A highly available paradigm for consistent object replication. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Israel, 1994.

[19] Idit Keidar and Danny Dolev. Efficient message ordering in dynamic networks. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*, pages 68–76, New York, USA, May 1996. ACM.

[20] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[21] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989. Also available as MIT Technical Memo MIT/LCS/TM-373.

[22] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann series in data management systems. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1996. Prepared with LaTeX.

[23] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 56–65, Los Alamitos, CA, USA, June 1994. IEEE Computer Society Press.

[24] 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):54–63, April 1996.

[25] R. De Prisco, A. Fekete, N. Lynch, and A. Shvartsman. A dynamic view-oriented group communication service. In *Proceedings of the 17h Annual ACM Symposium on Principles of Distributed Computing (PODC '98)*. ACM, 1998. to appear.

[26] Aleta M. Ricciardi, Andre Schiper, and Kenneth P. Birman. Understanding partitions and the "no partition" assumption. Technical Report TR93-1355, Cornell University, Computer Science Department, June 1993.

[27] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.