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

Roger Khazan[1], Alan Fekete[2], and Nancy Lynch[1]

[1] MIT LCS, 545 Technology Square, NE43-365, Cambridge, MA 02139, USA.
[2] Basser Dept. of CS, Madsen F09, University of Sydney, NSW 2006, Australia.

Abstract. We give a rigorous account of an algorithm that provides sequentially consistent replicated data on top of the view synchronous group communication service previously specified by Fekete, Lynch and Shvartsman. The algorithm performs updates at all members of a majority view, but rotates the work of queries among the members to equalize the load. The algorithm is presented and verified using I/O automata.

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. 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 recently 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* [14, 13]. In this paper, we use *VS* to support a second application: a *replicated data service* that *load balances* queries and guarantees *sequential consistency*.

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 involving omission, crashing or delay, but not Byzantine failures. The failures and recoveries may cause the network or its components to partition and merge. The biggest challenge for the service is to cope 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 everywhere. 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. 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 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, which 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 assigned queries, that is they are not blocked by missing update information.

Architecturally, 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.

Related Work

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 [25], Newtop [12], Relacs [3] and Horus [27]. Different services differ in the way they manage groups and in the specific ordering and delivery properties of their multicast primitives. Even though there is no consensus on what properties these services should provide, a typical requirement is to deliver messages in *total order* and *within a view*.

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, 8, 11, 15, 24, 26]. Fekete, Lynch, and Shvartsman recently presented the *VS* specification for a partitionable group communication service. Please refer to [14] for a detailed description and comparison of *VS* with other specifications.

Several papers have since extended the *VS* specification. Chockler, Huleihel, and Dolev [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 [9] have presented a specification for group communication service that provides a dynamic notion of primary view.

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 copies. The details of doing this in partitionable systems have been studied by Amir, Dolev, Friedman, Keidar, Melliar-Smith, Moser, and Vaysburd [18, 2, 1, 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 suggests 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 paper, we use a round-robin strategy originally suggested by Birman [4, p. 329]. According to this strategy, query requests are sent to the servers using totally-ordered multicast; the ith request delivered in a group of n servers is assigned to the server whose rank within this group is $(i \mod n)$. This strategy relies on the fact that all servers receive requests in the same order, and guarantees a uniform distribution of requests among the servers of each group. We extend this strategy with a *fail-over* policy that reissues requests when group membership changes.

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 concept of sequential consistency [21]. 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 of this paper provides an intermediate condition where the updates are atomic, but queries may see results that are not as up-to-date as those previously seen by other clients.

Contributions of this paper

This paper presents a new 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 algorithm is based on previous ideas (the load-balancing processing of queries is taken from [4] and the update processing relates to [18, 2, 1, 19]) we are unaware of a previously published account of a way to integrate these. In particular, we show how queries can be processed in minority partitions, and how to ensure that the servers always have sufficiently advanced states to process the queries.

Another important advance in this work is that it shows how a verification can use some of the stronger properties of VS. Previous work [14] verified TO, an application in which all nodes within a view process messages identically (in a sense, the TO application is anonymous, since a node uses its identity only to generate unique labels). The proof in [14] uses the property of agreed message sequence, but it does not pay attention to the identical view of membership at all recipients. In contrast, this paper's load-balancing algorithm (and thus the proof) uses the fact that different recipients have the same membership set when they decide which member will respond to a query.

The rest of the paper is organized as follows. Section 2 introduces basic terminology. Section 3 presents a formal specification for clients' view of the replicated service. Section 4 contains an intermediate specification for the service, the purpose of which is to simplify the proof of correctness. Section 5 presents an I/O automaton for the server's state-machine and outlines the proof of correctness.

2 Mathematical Foundations

We use standard and self-explanatory notation on sets, sequences, total functions ($\rightarrow$), and partial functions ($\hookrightarrow$). 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.

The modeling is done in the framework of the I/O automaton model of Lynch and Tuttle [23] (without fairness), also described in Chapter 8 of [22]. An I/O automaton is a simple state-machine in which the transitions are associated with named *actions*, which can be either *input*, *output*, or *internal*. The first two are *externally visible*, and the last two are *locally controlled*. I/O automata are *input-enabled*, i.e., they cannot control their input actions. An automaton is defined by its signature (input, output and internal actions), set of states, set of start states, and a state-transition relation (a cross-product between states, actions, and states). An *execution fragment* is an alternating sequence of states and actions consistent with the transition relation. An *execution* is an execution fragment that begins with a start state. The subsequence of an execution consisting of all the external actions is called a *trace*. The external behavior is captured by the set of traces generated by its executions. Execution fragments can be concatenated. Compatible I/O automata can be *composed* to yield a complex system from individual components. The composition identifies actions with the same name in different component automata. When any component automata performs a step involving action π, so do all component automata that have π in their signatures. The *hiding* operation reclassifies output actions of an automaton as internal.

Invariants of an automaton are properties that are true in all reachable states of that automaton. They are usually proved by induction on the length of the execution sequence. A *refinement mapping* is a single-valued simulation relation. To prove that one automaton implements another in the sense of trace inclusion, it is sufficient to present a refinement mapping from the first to the second. A function is proved to be a refinement mapping by carrying out a *simulation proof*, which usually relies on invariants (see Chapter 8 of [22]).

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

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 $t[p].view$ or as $p.view$ if t is clear from the discussion.

3 Service Specification S

In this section, we formally specify our replicated data service by giving a centralized I/O automaton S that defines its allowed behavior. The complete information on basic and derived types, along with a convention for variable usage, is given in Figure 1. The automaton S appears in Figure 2.

Fig. 1 Type information

Var	Type	Description
c	C	Finite set of client IDs. ($c.proc$ refers to the server of c).
db	DB	Database type with a distinguished initial value db_0.
a	$Answer$	Answer type for queries. Answers for updates are $\{ok\}$.
u	$Update : DB \rightarrow DB$	Updates are functions from database states to database states.
q	$Query : DB \rightarrow Answer$	Queries are functions from database states to answers.
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.

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 $\mathtt{request}(r)_c$, representing the submission of request r by a client c; S replies to its clients via actions of the form $\mathtt{reply}(o)_c$, representing the delivery of output value o to a client c.

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 apply update and query requests to the latest value of this variable. In the replicated system, this would imply that processing of query requests would have to be restricted to the primary components of the network.

In order to eliminate this restriction and thus increase the availability of the service, we give a slightly weaker specification, which does not require queries to be processed with respect to the latest value of *db*, only with respect to the value that is at least as advanced as the last one witnessed by the queries' client. For this purpose, S maintains a history *dbs* of database states and keeps an index $last(c)$ to the latest state seen by each client c.

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.

The assumption that clients block (i.e., do not submit any new requests until they get replies for their current ones) cannot be expressed within automaton S because, as an I/O automaton, it is input-enabled. To express this assumption formally, we *close* S by composing it with the automaton $Env = \prod_{c \in C}(C_c)$, where each C_c models a nondeterministic blocking client c (see Figure 3); Real blocking clients can be shown to implement this automaton. In the closed automaton $\overline{S}$, the `request` actions are forced to alternate with the `reply` actions,

Fig. 2 Specification S

Signature:

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

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

State:
$dbs \in SEQ0\ DB$, initially db_0. Sequence of database states. Indexing from 0 to $|dbs| - 1$.
$map \in C \hookrightarrow (Request + Output)$, initially $\perp$. Buffer for the clients' pending requests or replies.
$last \in C \rightarrow \mathcal{N}$, initially $\{* \rightarrow 0\}$. Index of the last db state witnessed by id.

Transitions:

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

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

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

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

which models the assumed behavior. In the rest of the paper, we consider the closed versions of the presented automata, denoting them with a bar (e.g., $\overline{S}$).

Fig. 3 Client Specification C_c

Signature:

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

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

State: $busy \in Bool$, initially $false$. Status flag. Keeps 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: $busy \leftarrow false$

4 Intermediate Specification *D*

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 layer D (see Figure 4), in which these tasks are separated. D is formed by splitting

Fig. 4 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: $map(c) \leftarrow ok$
$last(c) \leftarrow delay(c)$
$delay(c) \leftarrow \perp$

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.

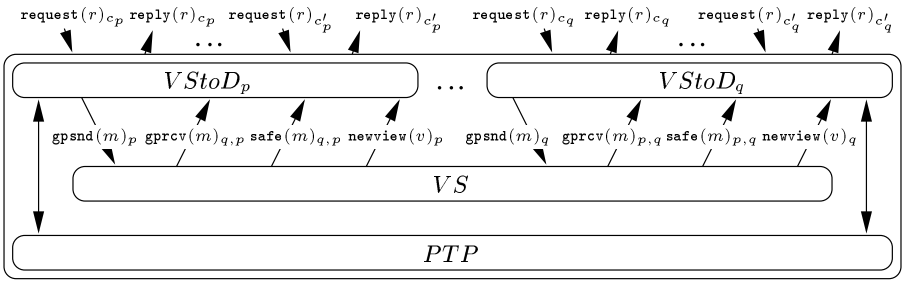
Lemma 1 *The following function $DS()$ is a* refinement *from $\overline{D}$ to $\overline{S}$ with respect to reachable states of $\overline{D}$ and $\overline{S}$.*[1]

$$
\begin{array}{lll}
DS(d:\overline{D}) \to \overline{S} & = & \\
\quad s.dbs & \leftarrow & d.dbs \\
\quad s.map & \leftarrow & overlay(d.map, \{\langle c,\ ok\rangle \mid c \in dom(d.delay)\}) \\
\quad s.last & \leftarrow & overlay(d.last, d.delay) \\
\quad s.busy_c & \leftarrow & d.busy_c \quad \textit{for all } c \in C
\end{array}
$$

Transitions of $\overline{D}$ simulate transitions of $\overline{S}$ with the same actions, except for those that involve `service`; these simulate empty transitions. Given this correspondence, the mapping and the proof are straightforward. The lemma implies that $\overline{D}$ implements $\overline{S}$ in the sense of trace inclusion. Later, we prove the same result about implementation $\overline{T}$ and specification $\overline{D}$, which by transitivity of the "implements" relation implies that $\overline{T}$ implements $\overline{S}$ in the sense of trace inclusion.

5 Implementation T

The figure 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 $c.proc = p$.



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 [14, see Appendix A] and a collection PTP of reliable reordering point-to-point channels between any pair of servers [22, pages 460-461], with all the output actions of this composition hidden, except for the servers' replies.

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

[1] Given $f, g : X \hookrightarrow Y$, $overlay(f, g)$ is as g over $dom(g)$ and as f elsewhere.

5.1 The Server's State-Machine $\boldsymbol{VStoD_p}$

The additional type and variable-name convention information appears in Figure 5. The I/O code for the $VStoD_p$ state machine is given in Figures 6 and 7.

Fig. 5 Additional Type Declaration

Var	Type	Description
	$\mathcal{Q} \subseteq \mathcal{P}(P)$	Fixed set of quorums. For any $Q \in \mathcal{Q}$ and $Q' \in \mathcal{Q}$, $Q \cap Q' \neq \emptyset$.
g	$\langle G, <_G, g_0 \rangle$	Totally-ordered set of view ids with the smallest element.
v	$V = G \times \mathcal{P}(P)$	An element of this set is called a *view*. Fields: *id* and *set*.
x	$X = G \times (C \times Update)^* \times \mathcal{N}$	Expertise information for exchange process. Fields: *xl*, *us*, *su*.
m	$M = C \times Update +$ $C \times Query \times \mathcal{N} + X$	Messages sent via VS: Either update requests, query requests, or expertise information for exchange process.
pkt	$Pkt = C \times Answer \times \mathcal{N} \times G$	Packets sent via PTP. ($\mathcal{N}$ is index of the witnessed *db* state.)

The activity of the server's state-machine can be either normal, marked by *mode* being *normal*, or recovery, marked by *mode* being either *expertise_broadcast* or *expertise_collection*. Normal activity is associated with the server's participation in already established view, while recovery activity — in a newly forming one. We also distinguish whether or not the server is a member of a *primary* view, which is defined as that whose members comprise a quorum ($view.set \in \mathcal{Q}$).

Fig. 6 Implementation ($VStoD_p$) : Signature and State Variables

Signature:

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

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

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 replica. Next state depends on current and action.
$map \in C\vert_{(c.proc=p)} \hookrightarrow Request + Output$, initially $\perp$.	Buffer that maps clients to their requests or replies.
$pending \in \mathcal{P}(C\vert_{(c.proc=p)})$, initially $\emptyset$.	Set of clients whose requests are being processed.
$last \in C\vert_{(c.proc=p)} \to \mathcal{N}$, initially $C\vert_{(c.proc=p)} \to 0$.	Index of the last *db* state seen by each client.
$updates \in (C \times Update)^*$, initially $[\,]$.	Sequence of updates. Indexing from 1. Fields: *c* and *u*.
$last\_update \in \mathcal{N}$, initially 0.	Index of the last executed element in *updates*.
$safe\_to\_update \in \mathcal{N}$, initially 0.	Index of the last "safe to update" element in *updates*.
$queries \in C \hookrightarrow (Query + Answer) \times \mathcal{N}$, initially $\perp$.	Query requests or answers, paired with their $last(c)$.
$query\_counter \in \mathcal{N}$, initially 0.	Number of queries received within current view.
$view \in V$, initially $V_0 = \langle g_0,\ P \rangle$.	Current view of *p*. Fields: *id* and *set*.
$mode \in \{normal,\ expertise\_broadcast,\ expertise\_collection\}$, initially *normal*.	Modes of operation. The last two are for recovery.
$expertise\_level \in G$, initially g_0.	The highest primary view id that *p* knows of.
$expertise\_max \in X$, initially $\langle g_0, [\,], 0 \rangle$.	Cumulative expertise collected during recovery.
$expert\_counter1 \in \mathcal{N}$, initially 0.	Number of expertise messages received so far.
$expert\_counter2 \in \mathcal{N}$, initially 0.	Number of expertise messages received so far as safe.

Processing of query requests is handled by actions of the type $\texttt{gpsnd}(c, q, l)_p$, $\texttt{gprcv}(c, q, l)_{p',p}$, $\texttt{query}(c, q, l)_p$, $\texttt{ptpsnd}(c, a, l, g)_{p,p'}$, and $\texttt{ptprcv}(c, a, l, g)_{p',p}$. The fact that servers of the same view receive query requests in the same order guarantees that the scheduling function of $\texttt{gprcv}(c, q, l)_{p',p}$ distributes query requests uniformly among the servers of one view.

Fig. 7 Implementation $VStoD_p$: Transitions

Transitions:

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

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

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

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

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

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

$\mathtt{newview}(v)_p$
Eff: $queries \leftarrow \bot$; $query\_counter \leftarrow 0$
$pending \leftarrow pending \ - \ \{c \mid (\exists\, q \,.\, \langle c,\ q\rangle \in map)\}$
$safe\_to\_update \leftarrow \max\big(safe\_to\_update,$
$\max\{last(c) \mid c \in C \ \wedge \ c.proc = p\}\big)$
$expertise\_max \leftarrow expertise\_max_0$
$expert\_counter1 \leftarrow 0$; $expert\_counter2 \leftarrow 0$
$mode \leftarrow expertise\_broadcast$
$view \leftarrow v$

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

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

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

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

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

$\mathtt{update}(c, u)$
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**
$pending \leftarrow pending \ - \ c$
$map(c) \leftarrow ok$
$last(c) \leftarrow last\_update$

$\mathtt{gprcv}(x)_{p',p}$
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**
$expertise\_level \leftarrow view.id$

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

Servicing of each query by a background thread $\mathtt{query}(c, q, l)_p$ is allowed only when the current state of the local database is at least as advanced as the last state witnessed by its client. This condition is captured by $last\_update \geq l$. The non-trivial part of this protocol is that the service actually guarantees that the servers always have the sufficiently advanced database states to be able to service the queries that are assigned to them.

When a 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 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 $\mathtt{update}(c, u)$. Each server maintains a sequence *updates* of update requests, the purpose of which is to enforce the order in which updates are applied to the local database replica. The sequence is extended each time an update request is delivered via a $\mathtt{gprcv}$ action. 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 are safe to execute and those that have already been executed. The *safe* prefix is extended to cover a certain update request on *updates* sequence when the server learns that the request has been delivered to all other members of that server's view.[2] The service guarantees that at all times *safe* and *done* prefixes of all servers are consistent (i.e., given any two, one is a prefix of another). Since *done* prefixes mark those update requests that have been applied to database replica, this property implies mutual consistency of database replicas.

When a server learns of its new view, it starts a recovery activity that is handled by actions of the type $\mathtt{newview}(v)_p$, $\mathtt{gpsnd}(x)_p$, $\mathtt{gprcv}(x)_{p',p}$, and $\mathtt{safe}(x)_{p',p}$. The query-related part of this activity was described above. For the update-related part, the server has to collaborate with others on ensuring that the states of all the servers of this view are consistent with their and other servers' past execution histories and are suitable for their subsequent normal activity.

For this purpose, each server has to be able to tell how advanced its state is compared to those of others. The most important criterion is the latest primary view of which the server knows. This knowledge may have come directly from personal participation in that view, or indirectly from another server. The server keeps track of this information in its state variable *expertise_level*. Two other criteria are the server's *updates* sequence and its *safe* prefix. The values of these three variables comprise the server's *expertise*.

Definition 1 *The cumulative expertise,* $\max_{\mathcal{X}}(X)$*, of a set or a sequence,* X*, of expertise elements is defined as the following triple*

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

[2] Some of the *optimistic* protocols, such as [16, 17], execute requests as soon as they are delivered by a total order multicast (ABCAST of *Horus*), but may result in inconsistent replicas, in which case they have to undo actions and roll the replicas' states back. On the other hand, *pessimistic* protocols, which implement *strict mutual consistency* among replicas, require additional information before they are able to execute a delivered request. The pessimistic version in [17] allows for a request to be executed only when a server collects a majority of acknowledgments, which have to be multicast by each server once it receives the request. Amir, Dolev, Melliar-Smith, and Moser in [1, 2] eliminate the need for end-to-end acknowledgments by using total order multicast with safe delivery, i.e., a message delivered to one member is guaranteed to be delivered to any other member of the same view provided it does not crash. As pointed out in [14, 13], "A simple 'coordinated attack' argument (as in Chapter 5 of [22]) shows that in a partitionable system, this notion of safe delivery is incompatible with having all recipients in exactly the same view as the sender." As a result, protocols based on this multicast primitive are more complicated than those based on *VS*, which separates message delivery and safe notification events.

As a first step, the server's collaboration with others during recovery activity aims at advancing everyone's expertise to the highest one known to them — their *cumulative* expertise (see Def. 1). Notice that adopting cumulative expertise of other servers can not cause inconsistency among replicas. The first step is completed with a delivery of the last expertise message via action $\mathtt{gprcv}(x)_{p',p}$.

Advancing the server's 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 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 messages have been delivered to all servers of this view, it extends its *safe* prefix to cover the entire *updates* sequence adopted during the 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.

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.

5.2 Refinement Mapping from $\overline{T}$ to $\overline{D}$

Automaton $\overline{D}$ has five types of actions. Actions of the types $\mathtt{request}(r)_c$ and $\mathtt{reply}(o)_c$ are simulated when $\overline{T}$ takes the corresponding actions. Actions of the type $\mathtt{query}(c)$ are simulated when $\overline{T}$ executes $\mathtt{ptprcv}(c,a,l,g)_{p',p}$ with $g = p.view.id$. The last two types, $\mathtt{update}(c)$ and $\mathtt{service}(c)$, are both simulated under certain conditions when $\overline{T}$ executes $\mathtt{update}(c,u)_p$. We define actions $\mathtt{update}(c,u)_p$ 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 $\mathtt{update}(c)$, that are just native simulate $\mathtt{service}(c)$, that are both leading and native simulate "$\mathtt{update}(c)$,$\mathtt{service}(c)$", and that are neither simulate empty transitions. Transitions of $\overline{T}$ with any other actions simulate empty transitions of $\overline{D}$.

Lemma 2 *The following function is a refinement from $\overline{T}$ to $\overline{D}$ with respect to reachable states of $\overline{T}$ and $\overline{D}$.*[3]

[3] If s is "$f_1, f_2, \ldots, f_n$" with each $f_i : A \to A$, and if $a \in A$, then $\mathrm{scan}(s) =$ $=$ "$f_1, (f_2 \circ f_1), \ldots, (f_n \circ \ldots \circ f_2 \circ f_1)$" and $\mathrm{map}(s,a) =$"$f_1(a), f_2(a), \ldots, f_n(a)$".

$$
\begin{array}{ll}
TD(t : \overline{T}) \rightarrow & \overline{D} = \\
\textit{let } t.done = & t[\wp].updates[1..t[\wp].last\_update], \textit{ where } \wp \in P \quad \textit{is any such that} \\
& t[\wp].last\_update = \max_{p \in P}\{t[p].last\_update\} \\
\; dbs & \leftarrow db_0 + map\,(scan(t.done), db_0) \\
\; map & \leftarrow \bigcup_{p \in P} \; t[p].map \\
\; last & \leftarrow \bigcup_{p \in P} \; t[p].last \\
\; delay & \leftarrow \{\langle t.done[i].c,\, i\rangle \mid 1 \leq i \leq |t.done| \; \wedge \; t[t.done[i].c.proc].last\_update < i\} \\
\; busy_c & \leftarrow t.busy_c \qquad \textit{for all} \quad c \in C
\end{array}
$$

An invariant will show that sequences of processed requests at different servers are consistent. In particular, all sequences which have maximum length are the same. $t.done$ is a derived variable that denotes the longest sequence of update requests processed in the system. This sequence corresponds to all modifications done 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 processed 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.

Fig. 8 Invariants used in the proof that $TD()$ is a refinement mapping (Lemma 2)

I 1 *For each server* $p \in P$, $p.last\_update \leq p.safe\_to\_update \leq |p.updates|$.

I 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:*

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

I 3 *Any update request that is safe somewhere but has not been executed at its native location is still reflected in its native map and pending buffers: If* $\langle c,\, u\rangle = p.updates[i]$ *and* $c.proc.last\_update < i \leq p.safe\_to\_update$, *then* $\langle c,\, u\rangle \in c.proc.map$ *and* $c \in c.proc.pending$.

I 4 *At most one unexecuted update request per each client can appear at that client's server: For any client* $c \in C$, *there exists at most one index* $i \in \mathcal{N}$ *such that* $i > c.proc.last\_update$ *and* $c = c.proc.updates[i].c$.

I 5 *For all* PTP *packets* $\langle c, a, l, g\rangle$ *on a* $in\text{-}transit_{p',p}$ *channel, 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) $l \leq \max_{\wp}\{\wp.last\_update\}$

The proof of Lemma 2 is straightforward given the five top-level invariants in Figure 8. To prove these invariants assertionally we have developed an interesting approach [20]: One of the fundamental invariants states that *safe* prefixes of *updates* sequences at all servers are consistent. To prove this fact, it is not enough to have properties only about *safe* prefixes — we need invariants that deal also with unsafe portions of *updates* sequences (because the latter become the former during an execution). Invariants that relate *safe* prefixes and *updates* sequences of different servers depend on the servers' *expertise_level*, which may have come to a server directly from the participation in a primary view, or indirectly from someone else. In our proof, we have invented a derived function $\mathcal{X}$ that expresses recursively the highest expertise achieved by each server in each view in terms of servers' expertise in earlier views. In a sense, it presents the law according to which the replication part of the algorithm operates. The recursive nature of this function makes proofs by induction easy: proving an inductive step involves unwinding only one recursive step of the derived function $\mathcal{X}$.

6 Future Work

This paper has dealt with safety properties; future work will consider performance and fault-tolerance properties, stated conditionally to hold in periods of good behavior of the underlying network. In particular, we are planning to compare the response time of this algorithm with others which share query load differently, for example based on recent run-time load reports which are disseminated by multicast.

Other possible extensions to this work involve determining primary views dynamically, using a service such as the one in [9], and integrating the unicast message communication into the group communication layer.

References

1. Y. Amir. *Replication using Group Communication over a Partitioned Network.* PhD thesis, The Hebrew University of Jerusalem, Israel, 1995.
2. 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, Israel, 1994.
3. O. Babaoglu, R. Davoli, L. Giachini, and P. Sabattini. The inherent cost of strong-partial view-synchronous communication. *LNCS*, 972:72–86, 1995.
4. K. P. Birman. *Building Secure and Reliable Network Applications.* Manning Publications Co., Greenwich, CT, 1996.
5. K. P. Birman and R. van Renesse, editors. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.
6. T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 322–330, New York, USA, May 1996.
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*, pages 237–246, 1998.
8. 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, June 25–27, 1996. IEEE.
9. 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*, pages 227–236, 1998.
10. D. Dolev and D. Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, Apr. 1996.
11. D. Dolev, D. Malki, and R. 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, May 30 –June 2, 1995. IEEE Computer Society Press.
13. A. Fekete, N. Lynch, and A. Shvartsman. Specifying and using a partionable group communication service. Extended version, http://theory.lcs.mit.edu/tds.
14. A. Fekete, N. Lynch, and A. 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, Aug. 21–24, 1997.
15. R. Friedman and R. van Renesse. Strong and weak virtual synchrony in Horus. Technical Report TR95-1537, Cornell University, Computer Science Department, Aug. 24, 1995.
16. R. Friedman and A. Vaysburd. Implementing replicated state machines over partitionable networks. Technical Report TR96-1581, Cornell University, Computer Science, Apr. 17, 1996.
17. R. Friedman and A. Vaysburd. High-performance replicated distributed objects in partitionable environments. Technical Report TR97-1639, Cornell University, Computer Science, July 16, 1997.
18. I. Keidar. A highly available paradigm for consistent object replication. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Israel, 1994.
19. I. Keidar and D. Dolev. Efficient message ordering in dynamic networks. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 68–76, New York, USA, May 1996.
20. R. I. Khazan. Group communication as a base for a load-balancing replicated data service. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, May 1998.
21. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.
22. N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann series in data management systems. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1996.
23. 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.
24. 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.
25. 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, Apr. 1996.
26. A. M. Ricciardi, A. Schiper, and K. P. Birman. Understanding partitions and the "no partition" assumption. Technical Report TR93-1355, Cornell University, Computer Science Department, June 1993.
27. R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, Apr. 1996.

A The *VS* Specification

The *VS* specification of [14, 13] is reprinted in Figure 9. M denotes a message alphabet and $\langle G, <_G, g_0 \rangle$ is 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.

Fig. 9 *VS-machine*

Signature:

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

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

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 finite sequence of $M \times P$, initially empty

for each $p \in P$, $g \in G$:
  $pending[p, g]$, a finite 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: $next\_safe[q, g] \leftarrow next\_safe[q, g] + 1$

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 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 them 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.