ACADEMIC PRESS

Available at www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

J. Parallel Distrib. Comput. 63 (2003) 410–433

Journal of Parallel and Distributed Computing

http://www.elsevier.com/locate/jpdc

Application-based dynamic primary views in asynchronous distributed systems

Alberto Bartoli[a,*] and Ozalp Babaoglu[b]

[a] *Dipartimento di Elettrotecnica, Elettronica, Informatica, University of Trieste, Via Valerio 10, 34100 Trieste, Italy*
[b] *Department of Computer Science, University of Bologna, Mura Anteo Zemboni 7, 40127 Bologna, Italy*

Received 24 January 2001; revised 4 December 2001

Abstract

We consider programming network applications that are based on the process group paradigm. When such applications are deployed in unreliable networks, they may partition into several disconnected clusters causing multiple views of the group's current composition to exist concurrently. In this paper we propose a mechanism for efficiently deciding when a view constitutes the "primary partition" for the group. Our solution is highly flexible and has the following features: possibility to modify selection rules at run-time without having to halt and restart the application; support for dynamic groups whose membership may change not only due to failures/recoveries but also due to processes voluntarily joining and leaving; ability to re-establish a primary partition even after a "total failure" scenario where all group members crash. These features facilitate the development of partition-aware applications that are capable of "adapting" themselves to their operating environment by establishing selection rules based on observed execution characteristics.

Keywords: Replication; Group communication; Virtual synchrony; Dynamic voting; Partitionable membership; Quorum system; Adaptability; Environment awareness

1. Introduction

Group communication has proven to be a useful technology for developing network applications that are to be deployed in asynchronous distributed systems [11]. Groups naturally capture the distribution that is inherent to many applications and the fact that a wide range of fault tolerance strategies can be realized through middleware using off-the-shelf components makes the technology particularly attractive. Informally, a *group* is a named set of processes that cooperate towards some common goal or share some global state. The composition of the group is dynamic due to processes that voluntarily *join* and *leave* the computation, those that need to be excluded due to failures and those that need to be integrated after repairs. A group membership service tracks these changes and transforms them into *views* that are agreed upon as defining the group's current composition. Group members communicate through reliable *multicasts* whose semantics are formalized as *view synchrony* that defines global ordering guarantees on message deliveries as a function of view changes [7].

Failures that partition the communication network are a fact of life in most practical distributed systems and they occur more frequently as the geographic extent of the system grows or its connectivity weakens due to the presence of wireless links [52,58]. What distinguishes a partitioning event from an ordinary communication failure is that it disrupts communication between clusters of sites—*partitions*—and the usual system layers cannot hide this fact from applications. To do so would require special communication layers that buffer messages at their origin throughout a partitioning event and retransmit them upon reconnection [39,44]. Even if all partitioning events are eventually repaired, this approach may be impractical for several reasons. First, the number of messages that need to be buffered for retransmission during extended periods where communication is interrupted may grow arbitrarily

*Corresponding author.
E-mail addresses: bartolia@univ.trieste.it (A. Bartoli), babaoglu@cs.unibo.it (O. Babaoglu).

0743-7315/03/$ - see front matter © 2003 Elsevier Science (USA). All rights reserved.
doi:10.1016/S0743-7315(03)00016-9

large. Second, communication state information has to survive across site failures or power cycles, and thus has to be maintained in stable storage [47,55]. But more importantly, application-level actions taken in one partition may "conflict" with actions taken in other partitions, thereby leading to inconsistencies in the application state.

Group membership services can be classified in two categories based on their handling of partitioning events. A *primary-partition group membership service* maintains a single, agreed upon view of the group at any given time [40,54]. To achieve this requirement despite partitions, a primary-partition group membership service has to limit group membership changes to a single partition and block processes in other partitions by either not delivering them any views [56] or by pretending that they have crashed so as to force rejoins after recovery [54]. In summary, a primary-partition group membership service promotes the development of what could be characterized as *partition-ignorant* applications since the presence of multiple partitions is hidden from them. A *partitionable group membership service* allows multiple views of the group to co-exist and evolve concurrently [7,20]. Each view corresponds to a different partition. A partitionable group membership service reflects the presence of partitions up to applications rather than trying to hide them, thereby enabling the development of *partition-aware* network applications [8]. Even when progress is possible in multiple partitions, many applications have stringent consistency requirements forcing them to restrict the full set of their operations to one partition alone: the *primary partition*. Only a subset of their operations may be available in non-primary partitions. In the limit, it may be the case that only the primary partition is allowed to make progress and non-primary partitions cannot serve operations at all and (temporarily) block.

In this paper we propose a methodology for enabling applications to establish their own notion of "primary partition" on top of a partitionable group membership service. Each group member determines whether it belongs to the primary partition or not by applying a local *selection rule* to its current view. The contribution of this paper rests in the development of algorithms with the following features for establishing primary partitions in partitionable systems:

- possibility to modify selection rules at run-time without having to halt and restart the application,
- support for dynamic groups whose membership may change not only due to failures/recoveries but also due to processes voluntarily joining and leaving,
- ability to re-establish a primary partition even after a "total failure" scenario where all group members crash,
- possibility to define selection rules on a per-application basis. Applications using different instances of a group infrastructure are not bound to use the same selection rule. The same applies to applications that share a common group infrastructure for efficiency [30].
- ability to base selection rules on local information alone without requiring any additional communication.

These features may enable the development of partition-aware applications that are capable of "adapting" themselves to their operating environment by establishing selection rules based on observed execution characteristics. For example, an application could arrange for the primary view to include more reliable and more heavily used sites in order to improve its availability and performance. The above features may also simplify the administration of network applications. For example, hardware/software upgrade of server instances could be accomplished without introducing service outages; load balancing and capacity tuning operations could be performed by activating, deactivating or migrating servers based on changes in demand patterns.

To make the presentation concrete, we present our methodology in the form of an Application Programmer Interface (API) that exposes to the application the interface of group communication, augmented with the notion of a selection rule as discussed above. We present the details of the API implementation in a separate Appendix.

Our solution makes use of Enriched View Synchrony (EnrVS): an extension to traditional view synchrony that facilitates the maintenance of application-defined shared state within groups [5]. The algorithms for preserving the consistency of this state across failures, recoveries, partitions and other relevant events are dependant on application semantics and cannot be obtained automatically by applying the methodology promoted by EnrVS. Yet, use of EnrVS greatly simplifies the task with respect to traditional view synchrony.

2. Background and related work

2.1. Background

In this section we provide a simplified overview of our proposal, as a background for the following example and discussion of related work. Full details will be given later. We consider a system consisting of a collection of processes that communicate through a network. Processes may crash and the communication network may partition. The system is asynchronous in that no bounds

are assumed on communication delays or relative speeds of processes. Each process has access to a stable storage [47] and a software layer implementing *group communication* (cf. Section 3). Group communication notifies each process regarding the composition of the partition to which it belongs. Processes in the same partition can communicate through multicast messages that are delivered by the group communication layer with powerful reliability guarantees.

We associate with each process a *weight*. A partition qualifies as primary when its members hold a strict majority of the total weight assigned to the group [29]. The weight distribution is defined in a *weight table* that is a logically shared object among the group. The weight table is implemented through a thin software layer called the WT-*layer*, placed between the group communication and the application layers. The WT-layer exports the same interface as the group communication layer, but augmented with: (i) a flag indicating whether the partition that a process belongs to qualifies as primary; and (ii) the operation changeWeights(wtNew) that a process may invoke for assigning wtNew as the new value for the weight table. The operation must be executed in a partition whose members define a majority of the weights according to both the current and new values of the weight table. Otherwise the weight table is left unchanged.

A key feature of our proposal is that the application layer may change the weight distribution *dynamically* and *on-line*—without having to halt and restart the application. For instance, the application layer could increase the weights associated with processes that turn out to be more available or that are more heavily accessed. The group composition may be dynamic in that processes may join or leave the application at will.

2.2. Example

Let us consider a replicated database application built over a group communication layer. Each site runs an instance of the database management system and maintains a copy of the database. A transaction is a sequence of read and write operations on objects. We use 1-copy-serializability as the correctness criteria: the concurrent execution of transactions on replicated data is equivalent to a serial execution on non-replicated data [18]. Transaction processing occurs as follows. Each site is a group member. A transaction T is submitted at some site $S(T)$. This site constructs a transaction message $m(T)$ that fully describes T and sends $m(T)$ using a totally ordered multicast primitive. Upon delivery of $m(T)$, each site checks whether it belongs to the primary partition. If so, $m(T)$ is queued for processing, otherwise $m(T)$ is ignored. Each site processes transaction messages serially, i.e., without any temporal overlap between one transaction and the next.

A site S whose partition is not primary may only perform read-only transactions. That is, after terminating execution of the transaction possibly in-progress at the time S leaves the primary partition, S starts discarding transaction messages involving write operations. Transactions executed while not in the primary partition could operate on uncommitted data or on data committed by transactions that will have to be undone upon re-joining the primary partition (see also the end of the next section).

The database replica of a site that joins the primary partition must be brought up-to-date to reflect the transactions committed without the site's involvement. The details of this *state transfer* operation can be found in [43]. The cited paper shows that transaction processing need not be suspended during state transfer.

Note. (i) use of group communication does not impose any specific requirements to the database management system, which is used as a black box [3]; (ii) neither distributed locking nor two-phase commit is required. This application is meant only to illustrate our methodology and should not be judged for its performance, for example, since transactions are executed in sequence. Some degree of temporal overlapping between the executions of different transactions may be allowed by implementing a form of concurrency control outside of the database management system [53]. Alternatively, one may integrate a protocol similar to the above within the database management system [41,42]. Although more complex, such an approach may exploit concurrency to a much higher degree.

Now let us consider the following execution:

1. Initially, the database is replicated at three sites, p_1, p_2, p_3 and each replica is assigned weight 1 such that a partition must contain at least two of them to be designated "primary". The initial value for the weight table will then be $\mathrm{wt}_0 = [(p_1, 1)(p_2, 1)(p_3, 1)]$.
2. Some time later, two additional replicas are deployed at sites q_1 and q_2, perhaps to increase throughput. These sites are also assigned a weight of 1. One of the sites will execute changeWeights(wt_1) with $\mathrm{wt}_1 = [(p_1, 1)(p_2, 1)(p_3, 1)(q_1, 1)(q_2, 1)]$. The operation has to be executed in a partition that is primary according to *both* wt_0 and wt_1, e.g., composed of p_1, p_2, p_3 or p_1, p_2, q_1, among others.
3. The weights of p_1 and p_2 are then increased so as to require that a primary partition must include at least one of them. Perhaps because these two sites happen to service a significant fraction of the write workload and increasing their weights may increase overall throughput in the presence of network failures. This sort of reasoning, as well as the decision to modify the weights, could be done either by a system administrator or automatically, through a procedure

that embeds the relevant policies. A site will execute changeWeights(wt_2), for example, with $wt_2 = [(p_1,4)(p_2,3)(p_3,2)(q_1,1)(q_2,1)]$. The operation has to be executed in a partition collecting a majority of the weights according to both weight tables wt_1 and wt_2. In other words, members of the partition must collect a weight of at least 3 according to wt_1 and at least 6 according to wt_2.

4. Finally, site p_2 is to be permanently withdrawn from the service. Perhaps, because the remaining sites are deemed sufficient to support the workload and p_2 is destined for another uses. Or, because p_2 is located at a geographical location that is to be removed from the system. A site will execute changeWeights(wt_3) specifying, for example, $wt_3 = [(p_1,4)(p_3,1)(q_1,1)(q_2,1)]$ (the new value assigns a zero weight to p_2 while preserving the policy that any primary partition must include p_1; of course, a different policy would require a different weight table). The operation has to be executed in a partition collecting a majority of the weights according to both weight tables wt_2 and wt_3. In other words, members of the partition must collect a weight of at least 6 according to the former and at least 4 according to the latter.

2.3. Observations

Of course, one could update the weight table by shutting the system down and executing a cold restart, but we are interested in solutions that work on-line. In this paper we are interested only in the *mechanisms*, not in the *policies* that one could build above them. In particular, we observe what follows:

- There is nothing in our proposal that forces applications to redistribute weights upon the occurrence of specified events (e.g., view changes). Weights are modified only if and when the application decides to. For example, the time between any two steps of the above execution could range from minutes to hours to days or weeks. During this time, an arbitrary number of view changes could occur (including *total failures*).
- Modifications similar to those at step 3 could be taken *repeatedly*, at regular intervals. This strategy would enable the application to "react" to changing conditions.
- Such modifications could be triggered by any combination of utilization data *and* availability data. For example, sites that turn out to be less available could be given smaller weights. Or, in case of intermittent connectivity between two geographically distant clusters, one could specify which of the two is to be considered the primary partition. The choice could be based either on administrative criteria (e.g., the headquarters of the company) or on utilization data (the one most heavily accessed by clients).

We do not specify what can and cannot be done in each partition, nor do we specify the protocol for executing operations within a partition. The nature of the specific application is irrelevant, as long as it requires a notion of primary partition.

In the example of the previous section, one may ensure that sites in a non-primary partition never read uncommitted data or data committed by a transaction that may have to be undone. To do so, it suffices that the group communication layer supports a form of *uniform* (also called *safe*) delivery of transaction messages [43,51]: if a site delivers a message in a partition, then each site in that partition also delivers the message or crashes. Whether this guarantee is provided or not is irrelevant to our proposal.

2.4. Related work

Use of weights to improve availability and performance of a replicated service was initially proposed in the context of a file service [29,59], then for a directory service [15,16] and for arbitrary data types [34,35]. The notion of a *quorum system* generalizes weighted majority [28]. A quorum system is a collection of sets of processes such that any two sets in the collection intersect. Each set in the collection is called a *quorum*. Weights provide a simple way to define a quorum system: any set of processes collecting a strict majority of the weights is a quorum. A quorum system may be immutable or *dynamic*. Dynamic quorum adjustments were proposed in various application domains: replicated file management [17], replicated directories [15,16], transaction processing [34,35], emulation of multi-writer/multi-reader registers [48], mutual exclusion [9,10].

Our proposal is a form of application-driven dynamic quorum adjustment: the current value of the weight table defines a quorum system, thus modifying the weight table modifies the quorum system. While the above proposals were tailored to specific problems, ours is inserted to an application-independent framework based on group communication. We are not aware of any similar effort (we will elaborate further on this claim below).

Some of the above works are based on a system model more restrictive than ours. The protocol in [17] assumes accurate and instantaneous detection of failures, as pointed out by [9]. The protocol in [10] does not tolerate partitions and assumes the existence of known bounds on relative processing speeds and message delivery times. Being based on group communication, our proposal assumes that failure detection may be inaccurate, the system is allowed to be asynchronous and the network may partition.

Our proposal uses quorums as a means for selecting the primary partition on the grounds that a broad range of applications deployed over group communication need a notion of primary partition. A common structuring for such applications is the following [5]: processes in the primary partition can execute all service operations; processes in a non-primary partition can execute only a (possibly empty) subset of the operations; processes that join a primary partition must be brought "up-to-date" by means of an application-specific *state transfer* before participating in the execution of the service operations. This structuring has some similarities with the quorum-based transaction processing protocols in [23,24], that are not based on group communication and in which the role of state transfer is played by a dedicated transaction that is run upon a merger and that spans all objects available in the newly formed partition.

A common approach for selecting the primary partition is *dynamic voting*, which can be informally summarized as follows. The first primary partition has a predefined composition. Then, a partition is primary only if it includes a majority of the previous primary partition. This approach may be embedded either into the application [38] or into a membership service [19,21,22,50,54,61]. The proposal of [19] tolerates total failures and is capable of accommodating on-line sets of processes that change dynamically. Our proposal also exhibits these properties. Of course, in both proposals, a process wishing to leave the computation forever may need to perform a prior negotiation with the other processes, otherwise forming a primary partition without that process might become impossible. An analogous requirement can be found in other fault-tolerant algorithms, e.g., [46,57]. Below we shall elaborate on dynamic voting within membership services because this option is closer to our proposal.

A membership service provides each process the composition of the partition it belongs to. This information takes the form of a *view*. Membership services may be classified as follows: (i) those that have no notion of primary view; (ii) those that attach a flag to each view for specifying whether the view is primary or not; (iii) those that report to processes only views qualifying as primary. Dynamic voting is one of the mechanisms that membership services belonging to (ii) and (iii) may use internally for selecting the primary partition. Another mechanism is majority-based selection for groups with static membership [27,36], but this mechanism cannot support dynamic adaptations like those of the previous example.

It is important to note the differences between dynamic voting and our proposal. In dynamic voting:

1. The minimum number of processes necessary to form a primary partition changes upon *every* view change (in our proposal changes to this quantity may be scheduled independently of view changes).
2. Such a change is performed by the system (in our proposal it is the application that decides if a change is required).
3. All processes are equivalent (in our proposal it is possible to differentiate processes through their weights).

By comparing these features to the example in the previous section, the difference between the two approaches is evident. While we provide mechanisms that each specific application can use to implement its own policy for selecting the primary partition, dynamic voting implements a pre-defined and immutable policy for dynamic quorum adjustments. With our approach, applications could be made to "react" dynamically to changing conditions or to adapt their behaviors according to significant statistics collected at run-time, as described in the previous section. This sort of adaptation would not be possible with dynamic voting. In the limit, an application could even use our mechanisms for implementing a policy analogous to dynamic voting: it would suffice to execute a suitable change of the weight table upon every view change (so that every majority of the new primary view would still be primary, similarly to [19]).

Evaluating the actual availability or performance improvements that could be obtained by our proposal is beyond the scope of this paper: such results would depend on the specific operating environment and policy used, whereas here we are interested only in mechanisms (much like [34,35], for example). Published analyses of dynamic voting (e.g., [37,38]) are not very useful for analyzing our proposal because: (i) they focus on a specific operating environment in which all processes and all communication links have the same failure probability; (ii) they take availability of the primary partition as the only performance index, neglecting the load actually submitted to each server. Our proposal is most useful when availability and/or utilization of processes are skewed, and perhaps change over time.

The simulation environment used in [37] includes a dynamic voting implementation called 1-*pending* that appears to be similar to the workings of our mechanism for implementing dynamic voting (i.e., update the weight table upon *every* view change, such that every majority of the primary view just installed will still be primary). In general, 1-*pending* forms a primary partition more often than a simple majority-based scheme but less often than the implementation in [19]. Thus, one may argue that in the operating environment and failure hypothesis considered in [37] the flexibility offered by our approach may have a cost. However, it is not surprising that an implementation tailored specifically for dynamic voting may be more efficient than a more

general one for which dynamic voting is just a special case. Moreover, in these environments for which dynamic voting is the most suitable policy, there may be no point in using a platform capable of supporting more general policies.

3. Group communication

Each process is equipped with a software layer implementing a form of group communication that we call *Enriched View Synchrony*[1] (*EnrVS* for short) (Section 3.3). EnrVS is an extension of *View Synchrony* (*VS*) aimed at simplifying the application-level handling of process recoveries after failures and merging of partitions after repairs, including the technical details of *state transfer* [4,5]. Typically, models for VS ignore such issues, but these models are often implemented with additional layers that simplify state transfer in a way *not* covered by the model [14,26,33,60,61]. Such layers enclose assumptions tailored to specific application domains and operating environments [13]. For example, they are suitable only for small amounts of state [32,61]. Other models incorporate the (application-dependent) state transfer actions into the view synchrony layer [22]. EnrVS, in contrast, provides a general and application-independent framework. In the next subsections we provide the necessary background for VS and EnrVS.

3.1. View synchrony

VS implements the notion of *process groups* and provides *reliable multicast* as the basic communication primitive. Processes may join or leave a named group. While a member of the group, processes communicate with each other through reliable multicasts. Reliable multicasts satisfy FIFO ordering (if a process multicasts m_1 before it multicasts m_2, then no process delivers m_2 unless it has first delivered m_1) and *total ordering* (if two processes deliver any two multicasts, then they deliver the two multicasts in the same order). Once a process has joined the group, it remains a group member until it leaves the group explicitly. For simplicity, we shall consider only processes that are group members.

View synchrony includes a group membership module that notifies each process about the set of processes that appear to be currently reachable. This information takes the form of *views*. A view consists of a set of process identifiers and a systemwide unique identifier. New views are communicated to processes in the form of *view change* events. A process that delivers a view change event $\mathrm{vchg}(v)$ is informed that the new view is v. In this case we say that the process has *delivered* or *installed* v. We say that an event e occurs in view v at a given process if and only if the last view delivered at that process before e is v. We denote by v.memb the set of processes composing the view v. Each view delivered to a process includes at least the process itself. We assume that a process that crashes and then recovers maintains the same identifier that it had before the crash. In other words, processes remain group members until they leave the group explicitly. To this end, each process stores its identifier in stable storage [2,47].

Multiple views of the group may exist concurrently. Views form a globally shared partial order, as follows. At each process, installed views form a totally ordered sequence. Given two views v and w, we say that w is an *immediate successor* of v (or v and w are *consecutive*) if and only if there is a process for which w is the next view to be installed after v. The transitive closure of the "immediate successor" relation is called *successor*. Two views are *concurrent* if and only if neither is a successor of the other. For example, consider Fig. 1 where rectangles indicate views, circles indicate processes and an arrow from view v to view w that w is an immediate successor of v. Processes are "colored" black or white so as to be able to trace their evolution. In the left figure, v_5 is a successor of v_1, v_6 is a successor or v_1, v_5, v_6 are concurrent. Intuitively, concurrent views are views that are installed at different processes and reflect different perceptions of the group membership, typically as a result of partitions. We extend the successor relation across failures of group members and across total failures as follows. Let v be the last view delivered by a process p before it crashes and let w be the first view delivered after its recovery. We shall say that w is an *immediate successor* of v and we shall assume that $\mathrm{w.memb} = p$.

An essential feature of view synchrony is that view changes are globally ordered with respect to message deliveries [7,12]: *Given any two consecutive views v and v′, any two processes that deliver both views must have*

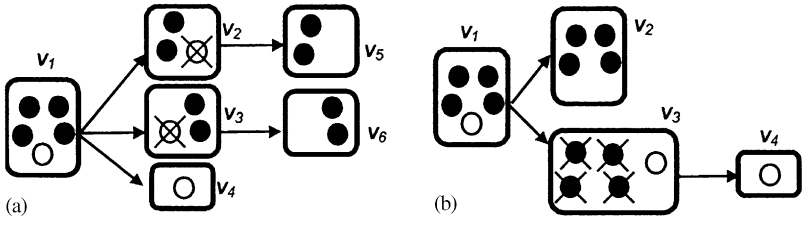

Fig. 1. Examples of view evolution.

delivered the same sequence of messages in view v. This property allows a process to reason globally on the basis of local information regarding the sequence of messages and view changes that have been delivered.

3.2. Overlapping concurrent views

Ideally, any two concurrent views should have empty intersection, to model "clean" network partitions. Unfortunately, it has been proved that any protocol for enforcing this property may be *blocking*: a failure (or a failure suspicion) occurring at an inopportune point of the membership protocol may delay a view installation until the failure recovers [6]. Existing platforms for group communication eliminate the possibility of blocking by allowing the existence of concurrent views that have overlapping composition [20,25,26]. That is, a process p that installed a view v could also be a member of one or more views concurrent to v. An example is in Fig. 1, where crossed out processes denote those that belong to but have not delivered the respective view. In practice, concurrent views that have overlapping composition might result from failures, either real or only suspected, during a view change protocol.

We say that a view whose members collect a strict majority of the weights, in the weight table is a *quorum view* (we use the term "primary view" with a different meaning, see below). With reference to our framework, the possibility of overlapping concurrent views implies that there might be multiple concurrent quorum views. For example, if each process in Fig. 1(a) was given weight 1, then v_2, v_3, v_4, would all collect a majority of weights (the same for v_2, v_3 in Fig. 1(b)). Applications may cope with the possibility of concurrent quorum views with application-specific techniques based on the following facts: (i) a view not installed by all of its members will disappear soon (the VS layer will detect that some members of the view are not responsive); and (ii) processes that are members of the same view v but actually delivered different views cannot communicate in v (e.g., in Fig. 1(a) the VS layer ensures that black processes in v_2 cannot communicate in this view with the white process; the same applies to v_3 at both sides of Fig. 1). Our proposal includes an application-independent solution to the problem of concurrent quorum subviews. While we do not see this additional feature as essential, it is indeed useful for the sake of completeness.

We say that a quorum view installed by all of its members is a *primary view*. In our proposal, a process may deliver in a quorum view a special message meaning "this is a primary view" (Section 4.1 for details). This special message is generated by the WT-layer autonomously. An application could be programmed such that, when a process installs a quorum view, the process does not take any irreversible action until the process is notified that the view is indeed primary (for example, in Fig. 1(a), such a notification may be delivered only in v_1, whereas in Fig. 1—right only v_1, and v_2).

3.3. Enriched View Synchrony

Enriched View Synchrony is an extension of VS that has been designed to simplify the application-level handling of state transfer [5]. EnrVS introduces the notion of *enriched view* (*e-view* for short), that augments the "flat" contents of a view by structural information in the form of *subviews* and subview sets, *sv-sets* for short. Processes in a view are grouped into non-overlapping subviews. Subviews in a view are grouped into non-overlapping sv-sets (see Fig. 2 and ignore ev_5 and ev_6 for the moment).

New e-views are communicated to processes in the form of e-view change events, that replace the traditional notion of view change. A process that delivers e-view change e-vchg(ev) is notified that the new e-view is ev. Such an event may notify either about a change in the *composition* of the e-view (the set of processes that can mutually communicate has changed; installing of ev_2, ev_3, and ev_4 in Fig. 2) or about a change in the *structure* of the e-view (the grouping in subviews or sv-sets of these processes has changed; installing of ev_5, ev_6). All processes that compose an e-view are notified of a change in structure, even those whose subview or sv-set composition has not changed.

The successor relation may be defined for e-views similarly to views. We say that e-vchg(ev) is a "view change" when this event notifies a change in composition. We say that two view changes e-vchg(ev) and

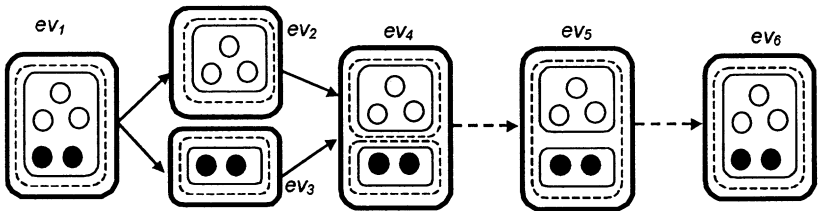

Fig. 2. Example of e-view evolution. Thick, dashed thin frames denote views, sv-sets and subviews, respectively. Arrows indicate view changes and dashed arrows indicate e-view changes that are not view changes. For simplicity, in the following figures sv-sets composed of a single subview are not traced out as dashed frames.

e-vchg(ev′) are *consecutive* to mean that there is a process for which the next view change to be delivered after e-vchg(ev) is e-vchg(ev′). We say that an event (delivery of a message or e-view change) occurs *in view* ev at process p, to mean that the last view change delivered by p when the event occurs is e-vchg(ev).

The system guarantees that delivery of e-view change events is totally ordered within each view: *Given any two consecutive view changes* e-vchg(ev) *and* $\mathrm{e}-\mathrm{vchg}(\mathrm{ev}')$, *any two processes that deliver both events have delivered the same sequence of e-view changes in view* ev. Moreover, e-view changes are globally ordered with respect to message deliveries: *Given any two consecutive e-views* ev *and* ev′, *any two processes that deliver both e-views have delivered the same sequence of messages in e-view* ev. Intuitively, one may think of an e-view change as a special message within the total order of messages delivered within a view.

Changes in the structure of an e-view may occur only upon a request from the application. The application may only *merge* subviews or sv-sets through dedicated primitives:

- svSetMerge(svSetList) creates a new sv-set that is the union of the sv-sets given in svSetList (e.g., ev_5 in Fig. 2 is installed as a result of a svsetMerge(); notice that this e-view differs from ev_4 in structure but not in composition).
- subviewMerge(svList) creates a new subview that is the union of the subviews given in svList (e.g., ev_6 in Fig. 2 is installed as a result of a subviewMerge()). If all the subviews in sv-list do not initially belong to the same sv-set, the call has no effect. The resulting subview belongs to the sv-set containing the input subviews.

The system preserves the structure of an e-view across view changes, as follows (*Structure Property*). Let e-vchg(ev) and e-vchg(ev′) be two consecutive views and let p, q be two processes that deliver both events. Let $\text{e-vchg}(\mathrm{ev}_L)$ be the last e-view change delivered by p and q in view ev (it might be $\text{e-vchg}(\mathrm{ev}_L) = (\text{evchg}(\mathrm{ev}))$. *If p and q are in the same subview (sv-set))in ev_L, then they remain in the same-subview (sv-set) also in ev′; moreover, if p and q are in different subviews (sv-set)in ev_L, they remain in different subviews (sv-set) also in ev′.* This property is illustrated in Fig. 2 (e-views $\mathrm{ev}_1, \mathrm{ev}_2, \mathrm{ev}_3, \mathrm{ev}_4$). Black and white processes enter the same e-view ev_4 as a result of a decision taken by the system, but remain in different subviews and sv-sets; they will enter the same sv-set and subview only when the application decides to, in ev_5 and ev_6, respectively.

The interface of the EnrVS layer is summarized in Table 1.

We omit the operations for inspecting the content of an e-view (processes, subviews, sv-sets) and for figuring out the type returned by getEvent() (message or e-view change). Identifiers of either subviews or sv-sets are relative to the e-view in which they are defined. If an input parameter of either subviewMerge() or svSetMerge() is an identifier no longer defined, then the call has no effect. Delivery of messages from a given process satisfies FIFO-ordering, irrespective of whether the message is sent through Send() or through Mcast().

Table 1
Interface of the EnrVS layer

join( )	Join the group
leave( )	Leave the group. Upon return, the process is no longer a group member
Mcast(msg, dstSpec)	Multicast msg within dstSpec (either mySV, mySVSet or myEview)
Send(msg, dst)	Send msg to the indicated e-view member
GetEvent( )	Returns either a message or an e-view change. Blocking operation
SubviewMerge(svList)	See the text
SvSetMerge(svSetList)	See the text

EnrVS is best exploited if applications are structured according to a certain simple methodology that we summarize below (see [5] for a more general presentation; the cited paper specifies EnrVS for a system that features only FIFO-ordered multicast, whereas here we assume totally ordered multicast).

- Processes in the same subview have the same application-defined state.
- Processes in a "primary subview" may execute all service operations (a subview is primary when its members collect a majority of the weights and the corresponding view has been installed by all of its members; a rigorous definition will be given in Section 4.1). The other processes may execute a (possibly empty) subset of these operations.
- If a view contains two or more subviews, these subviews are collected in an sv-set with a call to SVSetMerge(sv-set-list); a proper *reconciliation* procedure is then executed among processes in this sv-set in order to define a new common value for the application-defined state (i.e., a state transfer if one of the participating subviews is a primary subview); upon completion of this procedure, all processes in this sv-set merge into a single subview with a call to SubviewMerge(sv-set-list). For instance, in Fig. 2, black and white processes have a common shared state (ev_1) and a partitioning event occurs; after the merger (ev_4) these processes enter the same sv-set (ev_5) and run a proper reconciliation procedure in order to define a new common state; finally they merge again into the same subview (ev_6). Note, SVSetMerge() and SubviewMerge() are called by one member only but all members see the effect through new e-views.

The key property of the above methodology is that if a view change occurs during the execution of a service operation, the set of participants in the operation cannot expand: this set either remains unchanged or it shrinks (Structure Property applied to subviews). That is, whereas a primary *view* may expand at arbitrary times as a result of view changes scheduled by the system, a primary *subview* may expand only as a result of an explicit request from the application. A little thought will reveal that this framework greatly simplifies the programming details of service operations and, in particular, state transfer that require several rounds of message exchange. The same reasoning, with the Structure Property applied to sv-sets, may be applied to view changes occurring during a reconciliation procedure. Of course, the problem of a view change occurring during the execution of a service operation or a reconciliation procedure must be handled also in algorithms designed for VS. We note that all such algorithms that we are aware of, are designed to restart the algorithm upon a view change. This is an artifact of the limited expressiveness of VS rather than being imposed by the application requirements (see also Section 5.2).

4. Interface of the WT-layer

The WT-layer maintains a *weight table* for the group. Each entry of the table contains the identifier of a group member and the *weight* associated with it. Members not included in the table are implicitly granted zero weight. Conceptually, the weight table is a single object shared by all group members.

We denote by sv(ev) the subview sv defined in e-view ev and by sv(ev).memb the membership of the subview. We use notation $\mathrm{ev} \in h_p$ to indicate that process p installed e-view ev and $\mathrm{sv}_p(\mathrm{ev})$ to indicate the subview of process p in $\mathrm{ev} \in h_p$. We define predicate $Q(S)$ to be true if and only if the set of processes S collects a strict majority of the weights in the weight table (in this case we say that S *is a quorum*). The value of $Q(S)$ depends on the cut along which the predicate is evaluated, because the weight table may change during the execution. However, the value $Q(\mathrm{sv}(\mathrm{ev}).\mathrm{memb})$ is always the same: either members of sv(ev) collect a majority of the weights along any cut where sv(ev) is defined or they do not (see next section). Finally, we use the following definitions:

- *Quorum subview*: sv(ev) is a quorum subview if and only if $Q(\mathrm{sv}(\mathrm{ev}).\mathrm{memb})$ is true.
- *Primary subview*: Let e-vchg(ev) be delivered in view ew; sv(ev) is a primary subview if and only if it is a quorum subview and ew has been installed by all of its members.

Note, a process whose subview is primary is guaranteed to remain in a primary subview until the next view change (within a view subviews may only expand).

The key guarantee of the WT-layer is:

(WT-1.) Primary subviews form a totally ordered sequence.

4.1. Application Programmer Interface (API)

An application consists of a set of upcalls that are invoked by the WT-layer upon the occurrence of specified events:

1. upcallMessage(*m*: message) Invoked to deliver message *m*.
2. upcallChange(ev: e-view change) Invoked to deliver e-view change ev. An-e-view change conveys membership information augmented with the structuring in subviews and sv-sets (Section 3.3). Additionally, an e-view change includes a flag, called *quorum flag*, that tells whether the subview of the executing process is a quorum subview. We denote by $\mathrm{ev.quorum}_p$ the value of the quorum flag at process p in e-view ev. It is guaranteed that:

WT-2. Processes that installed the same subview have the same quorum flag:

$$\forall \mathrm{sv}(\mathrm{ev}), \mathrm{ev} \in h_p \text{ and } \mathrm{ev} \in h_q \text{ and } \mathrm{sv}_p(\mathrm{ev}) = \mathrm{sv}_q(\mathrm{ev}) \Rightarrow \mathrm{ev.quorum}_p = \mathrm{ev.quorum}_q.$$

WT-3. If the quorum flag is set, then the process' subview is indeed a quorum subview:

$$\forall \mathrm{ev}. \quad \mathrm{ev.quorum}_p \Rightarrow Q(\mathrm{sv}_p(\mathrm{ev})).$$

The above properties could be satisfied by an implementation that never sets the quorum flag, but our implementation does not exhibit this trivial behavior. Indeed, a quorum subview is always delivered with the quorum flag set with this exception: after certain failures occurring while updating the weight table, there might be transitory periods in which a quorum subview is undetected (Sections 4.3 and 5).

Message and e-view change deliveries provide the guarantees of EnrVS (Section 3.3).

The WT-layer may multicast spontaneously a Primary message within a quorum subview:

WT-4. If the quorum flag is set when a process delivers *a Primary* message, then the process' subview is a primary subview.

That is, a process is first notified that its subview is a quorum, then the subview is primary. It is up to the application to decide what can be done and what cannot be done while waiting for the Primary message. We do not insist in guaranteeing that a Primary message be delivered in *every* primary subview. However, if no view

Table 2
Interface of the WT-layer

mcast(msg, dstSpec)	Multicast msg within dstSpec (either mySV, mySVSet or myEview)
send(msg, dst)	Send msg to the indicated e-view member
leave()	Leave the group (see the text)
changeWeights(wtNew)	Request to assign the value wtNew to the weight table
readWeights()	Return the value of the weight table

changes occur for approximately two message round-trip times, then the Primary message is indeed delivered (Appendix B).

Within the above upcalls, a process may invoke the operations in Table 2.

Note, primitives for subview and sv-set merging are *not* available—they are invoked only from within the WT-layer (in Section 3.3, the term "application" was used in a broad sense to indicate the layer immediately above the EnrVS layer, whereas in this specific case this layer is the WT-layer). Operations mcast() and send() are analogous to those in Table 1. Operation leave() is for requesting to leave the group, as follows. The invoking process remains a group member and, if it does not crash, eventually it will deliver either a LeaveOK or a LeaveRefused message (messages from the WT-layer are tagged with a type field written in SmallCaps); the former notifies the process that it is no longer a group member, the latter notifies the process that it cannot leave the group: a process can leave the group only when the weight table grants zero weight to it. A process joins the group with the join() operation, that blocks the invoking thread and transfers control to the WT-layer until the process is no longer a group member, i.e., until completion of the upcallMessage() triggered by the delivery of a LeaveOK.

Operation ChangeWeights(wtNew) is a request to assign the value wtNew to the weight table. Eventually, a $\langle WTChange, \text{wtNew}, \text{outcome} \rangle$ message will be delivered at all members of the subview sv of the invoking process (unless this process crashes before the request has been processed by the WT-layer). The same delivery guarantees as for application-originated messages apply to WTChange messages. The outcome is an enumerated value selected by the WT-layer for notifying whether the weight table has been indeed updated, as follows (ev indicates the e-view in which the WTChange message is delivered and q denotes the process that delivers the message):

- Ok, meaning that the weight table has been updated to wtNew. In this case, ev.quorum_q = true.
- NotPossible, meaning that the weight table has been left unchanged.
- Unknown, meaning that an attempt to update the weight table has initiated but q cannot tell whether the value has become wtNew or has remained unchanged. In this case, ev.quorum_q = false.

In order to update the weight table all the following constraints must be satisfied (wtOld indicates the previous value of the weight table):

C1. sv is a quorum of both wtOld and of wtNew.
C2. Let switched denote the set of processes with zero weight in wtOld and non-zero weight in wtNew. Set switched is a subset of sv and is not a quorum of wtNew.
C3. There is at least one set of processes that is a quorum in wtNew and no disjoint sets of processes that are a quorum in wtNew.

Constraint C1 tells that the weight table can be updated only provided sv would be a quorum subview before and after the update. This constraint is necessary for coping with failures occurring during the update and appears in similar forms in virtually any algorithm for changing the notion of "primary partition" dynamically (e.g., [19,38,54]). This constraint is also the reason why the value of predicate $Q(\text{sv(ev)})$ does not change even though the weight table may change. Constraint C2 ensures that is always possible to form a quorum subview, even if processes leave the group (see Section 5.7). Constraint C3 is obvious.

View changes could occur during an execution of ChangeWeights(). The outcome is Ok if constraints C1–C3 are satisfied during the entire execution; it is NotPossible if one of the constraints is not satisfied at the beginning of the execution; it is Unknown otherwise, i.e., when the constraints are satisfied at the beginning but a view change occurs and one of the constraints does not hold in the new view (clearly, only C1 or C2 could not be satisfied in the new view). Note that the WT-layer never changes the weight table spontaneously.

Finally, operation readWeights() is a request to receive a copy of the weight table. The operation returns immediately and the process will receive a $\langle WTValue, \text{wt} \rangle$ message containing the weight distribution wt. If ev.quorum_p = false when the message is received, then wt might no longer correspond to the actual weight distribution—the weight table might have been updated in an e-view concurrent to ev.

4.2. Automatic application-level state transfer

A basic problem of any application deployed over group communication is restoring the "consistency" of the application-level state when a process recovers or a partition heals [5]. Here we augment the interface of the previous section to incorporate the ability to perform *automatic state transfer*. We provide capabilites similar

to those found in existing platforms [1,14,27,49], except that our framework is based on subviews rather than on views: *when a set of processes merge into a quorum subview*, *all members of the resulting subview have the same application-defined state*. To this end, an application must implement the following upcalls (that cannot invoke any operation of the WT-layer):

1. upcallGetState(): ByteArray Invoked to fetch the application-level state of the process. This state must be returned in a serialized form suitable for transmission.
2. upcallSelectState(s: set of ByteArray): ByteArray Invoked to select the application-level state to be assumed by a set of processes that are about to merge into a single subview. The input argument is a set of ByteArrays, one for each process. Each element was returned by an invocation of upcallGetState() at the corresponding processes. Clearly, the choice criterion implemented by this upcall is application-dependent.
3. upcallPutState(s: ByteArray) Invoked to set the application-level state of the process to the value described in s, that is a ByteArray obtained through an invocation of upcallGetState() (typically at another process).

Note, input and output arguments to the above upcalls are fully opaque to the WT-layer.

It is useful to outline the implementation of state transfer (Sections 5.2 and Appendix B). When an e-view is composed of multiple subviews, the WT-layer takes the following steps: (i) create an sv-set encompassing all subviews; let S denote the set of processes in this sv-set; (ii) invoke upcallGetState() at each $p \in S$; (iii) invoke upcallSelectState() at a designated member of S, passing the set of states obtained at the previous step as input; (iv) invoke upcallPutState() at each $p \in S$, passing the state obtained at the previous step as input; (v) merge all members of S into a single subview. To make sure that steps (ii)–(iv) occur atomically the procedure is such that: delivery of application-originated messages is suspended; and, a view change provokes the procedure to restart from step (ii).

This implementation of automatic state transfer is analogous to those found in existing platforms, in particular concerning the following constraints: (i) the *entire* state is transmitted, without the possibility to negotiate the portions that actually need to be transferred; (ii) delivery of application-originated messages is suspended; (iii) a view change provokes the transfer to restart; and, (iv) the state is transferred through primitives of the group communication platform (rather than, e.g., TCP). Such details may or may not fit the need of a specific application. In fact, facilities for automatic state transfer tend to be used as a black box and applied even in settings for which they are not suitable [13]. Although EnrVS simplifies the implementation of application-specific state transfer policies without the above constraints, this topic is orthogonal to this paper. We have preferred to show in full detail one of the possible policies for the sake of concreteness. An important application domain in which the above implementation is clearly unsuitable is replicated databases, where the application-level state may be several GBytes (see [43] for details).

4.3. Liveness guarantees

The API as specified so far does not ensure that when an e-view is composed of multiple subviews the WT-layer merges them into a single one. This expected behavior is indeed guaranteed, unless a view change occurs before the merging:

WT-5. If two processes remain in the same view for a sufficiently long time, then they will be members of the same subview before installing another view.

This property implies that any e-view has a successor e-view including a quorum subview (unless view changes always occur "too quickly"). The problem is, *not* every quorum subview is detected as such (property WT-3 holds in only one direction). The following property ensures that quorum subviews cannot remain undetected forever:

WT-6. If every failure recovers and the number of view changes is finite, then each e-view with the quorum flag unset has a successor e-view with the quorum flag set:

The reason for the hypothesis in this property is evident. If a process fails and never recovers, it might become impossible to form a quorum again (depending on the weight distribution at the time of the failure). If view changes never stop, then it might become impossible to complete the WT-layer protocol that restores the consistency of the weight table after a view change (depending on "how long" each view lasts).

Note, properties WT-1–WT-6 are provided even across total failures where all processes crash or become completely isolated from each other, and in an environment where processes may join or leave the system (almost) at their will.

5. Implementation of the WT-layer

5.1. Main data structures

The weight table is replicated at each process. A process p maintains a description of its replica in a record wt_p composed of the following fields: (i) wt_p.curr: the value of the weight table; (ii) wt_p.currid: a

system-wide unique identifier of the execution of ChangeWeights() that assigned the curr field; (iii) wt_p.vn: a version number that counts the number of updates applied to wt_p.curr; (iv) wt_p.prop: during an execution of ChangeWeights() the value about to be assigned to wt_p.curr, otherwise a null value; (v) wt_p.propid: a system-wide unique identifier of the ongoing execution of ChangeWeights(), or a null value in case wt_p.prop is null. When p starts it restores wt_p from its stable storage [47]. A successful execution of ChangeWeights() involves two atomic writes of wt_p on the stable storage.

When the system first bootstraps, the weight table is initialized, say to WT_0. Each process p that is granted a non-zero, weight in WT_0 assigns $wt_p = \{WT_0, \text{null}, 0, \text{null}, \text{null}\}$. Each process q that either is granted a zero weight in WT_0, or that is not part of the initial membership and is added to the system later, initializes all fields of wt_q to null. A null value for wt_q.curr means that q has no information whatsoever as to the weight distribution. A process p may leave the group only if fields curr and prop of wt_p grant zero weight to it.

Each process maintains a quorum flag in volatile storage and updates it upon every view change, as follows (p and sv_p denote the executing process and its subview, respectively):

- wt_p.prop = null $\Rightarrow$ (quorum := true $\Leftrightarrow$ sv_p.memb is a quorum in wt_p.curr),
- wt_p.prop $\neq$ null $\Rightarrow$ (quorum := true $\Leftrightarrow$ sv_p.memb is a quorum in both wt_p.curr and wt_p.prop).

The quorum flag is also updated upon an e-view change that merges multiple subviews into a single one (see next section, end of the propagation algorithm). The rule is the same, except that in certain executions the flag is set to false irrespective of sv_p.memb.

5.2. Overview of the implementation

The WT-layer receives either e-view changes or messages from the underlying EnrVS layer and forwards them to the application layer. E-view change events are augmented to include the quorum flag. Messages generated by the WT-layer for management of the weight table are *not* forwarded. Management of the weight table is split in two main algorithms, that we call *update* and *propagation*. We provide a high-level description of these algorithms below and full details, including pseudo-code, in Appendix B.

The update algorithm is executed amongst members of a subview for updating the respective wt replica. The algorithm is run as a result of ChangeWeights(wtNew):

U1. The invoking process checks that constraints C1–C3 of Section 4.1 are satisfied; if any check fails, the process delivers a WTCHANGE message with a NotPossible outcome to the application and the algorithm terminates; otherwise, the process multicasts an UPDATEREQUEST message within its subview; the message includes wtNew and a system-wide unique update identifier.

U2. Upon delivering the UPDATEREQUEST, each process checks that constraints C1–C3 of Section 4.1 are satisfied; if any check fails, the process delivers a WTCHANGE message with an Unknown outcome to the application and the algorithm terminates; otherwise, each process assigns wt.prop and wt.propid to the relevant fields of the UPDATEREQUEST message and writes wt on the stable storage.

U3. Each process sends an UPDATEACK to a designated process in the subview, called update coordinator. This process is elected without any message exchange, by applying a deterministic function to the composition of the subview.

U4. When the update coordinator has received the UPDATEACK from all members of the subview, it multicasts an UPDATECOMMIT message within the subview.

U5. Upon delivery of the UPDATECOMMIT, each process assigns wt.curr := wt.prop, wt.currid := wt.propid, and increments wt.vn; then it clears wt.prop, wt.propid and, finally, writes wt on the stable storage.

The update algorithm has some similarities with two-phase commit protocols [31,45] and it may be useful to point out that there are substantial differences between them. In a two-phase commit protocol, all participants are required to vote and even a single "no" vote implies that the decision has to be abort. In our update algorithm, on the other hand, whether an update should succeed or not depends on the *set* of participants that applies the update. Furthermore, this set may change *dynamically* as it depends on both the current and new value of the *object being updated* (i.e., the weight table).

The propagation algorithm is initiated whenever an e-view includes multiple subviews and merges all participants into a single subview. Members of the resulting subview sv have the same value for the quorum flag. They also have the same wt replica and application-defined state, except in certain executions (Section 5.5 for details). In these executions members of sv may have differing wt replicas and application-defined state but the quorum flag is guaranteed to be false, thus execution cannot proceed beyond step U2. In the EnrVS framework, the propagation algorithm is an example of reconciliation procedure (Section 3.3).

P1. A designated process, called the propagation coordinator, constructs an sv-set encompassing all sv-sets composed of a single subview. Members of this sv-set are the participants in the algorithm execution. The coordinator is elected without any

message-exchange, by using a deterministic function.

P2. Upon delivery of the e-view notifying the construction of the sv-set, participants start buffering application-originated messages, received from the EnrVS layer rather than delivering them to the application.

P3. Each participant sends to the coordinator a PROPCOLLECT message containing the wt replica and the local application-level state (returned by upcallGetState()).

P4. When the coordinator has received the PROPCOLLECT message from each participant, it uses the received information for selecting the "most recent" wt replica and the new application-level state—the former is the wt replica with highest vn field (see Section 5.5 for details), the latter is returned by upcallSelectState(). Then, the coordinator multicasts within the sv-set a PROPEND message carrying the new values for the wt replica and application-defined state and, finally, merges all participants into a single subview.

P5. When the subview merging has occurred, each participant updates the local wt replica and application-defined state (through upcallPutState()). Then, the process delivers to the application the messages buffered since step P2 and stops buffering application-originated messages received from the EnrVS layer.

View changes are handled as follows (recall that a view change is an e-view change notifying a change in the *composition* of a process e-view):

- *Update algorithm* (*coordinator blocked at* U4, *the other participants at* U5): Each participant delivers the view change to the application and checks again C1–C3 of Section 4.1; if any check fails, each participant delivers a WTCHANGE message to the application with Unknown outcome and the algorithm terminates (the process cannot tell how many other processes have indeed updated the respective wt replica; this aspect illustrates another difference from two-phase commit protocols: an UPDATEACK message cannot be interpreted as a "yes" vote, as there may be executions in which all participants in the same subview vote "yes" yet the decision has to be abort). If the coordinator is no longer a member of the new e-view, each participant continues from step U3.
- Propagation algorithm (coordinator blocked at P4, the other participants at P5): Each participant delivers the application-originated messages buffered so far and then the view change. Then, each participant continues from step P3 (rolling back to P3 is necessary to make it appear that application-level state transfer occurs atomically (i.e., it is necessary that no participant executes upcallMessage() in between upcallGetState() and upcallPutState()).

We remark that execution of either algorithm may proceed across view changes. This feature is intrinsic to the algorithm design style made possible by EnrVS, where the set of participants in an algorithm execution may never expand due to a view change, it may only *shrink* (Section 3.3).

Multiple instances of either alogrithm may proceed concurrently within the same e-view, as each instance will be performed within a separate subview or sv-set (the next section shows an example). Clearly, in case of an update, at most one such instance may be able to proceed beyond step U2. At any given process, executions of the two algorithms are coordinated as follows:

- Update instances are executed serially and so do propagation instances.
- An update cannot begin while a propagation is in progress—i.e., a process that delivers an UPDATEREQUEST message (step U1) while executing the propagation algorithm queues the message and handles it upon termination of the propagation.
- A propagation can begin while an update in progress. However, it will not complete step P3 until the update instance has completed.

To clarify the last rule, suppose process p is executing an update instance and delivers a view change. Upon delivering this view change, another process q might start the propagation algorithm (p and q are in different subviews). However, neither p nor q can proceed beyond step P3 because p will not send its own PROPCOLLECT until completing the update.

5.3. *Example of* e-*view evolution*

Fig. 3 shows an example of e-view evolution. Let us suppose that it shows the very first e-views since the creation of the group. Black processes enter the same view in ev_3 and start an instance of the propagation algorithm (ev_4). While this instance is in progress, the view expands to include the gray process and the white process. These two processes start a further instance of the propagation algorithm *in parallel* to the one being executed between the black processes (ev_8). Once both instances have completed ((ev_{10}), the two resulting subviews execute an instance of the propagation algorithm among themselves (ev_{11}) and finally merge into a single subview (ev_{12}).

Whether any of the subviews in the example is primary depends on the weight table. For instance, let the initial the weight table WT_0 contain an entry for the black processes and the gray process, with weight 1

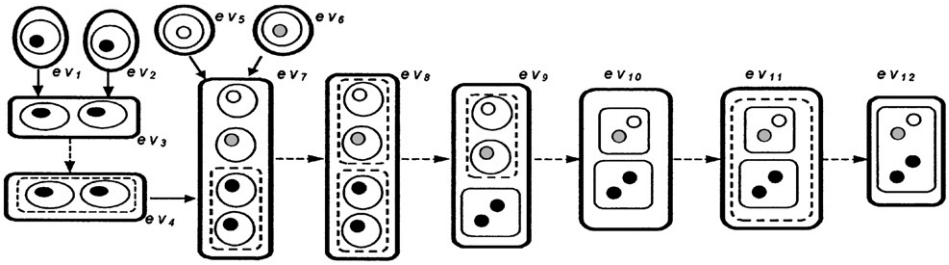

Fig. 3. Example of e-view evolution.

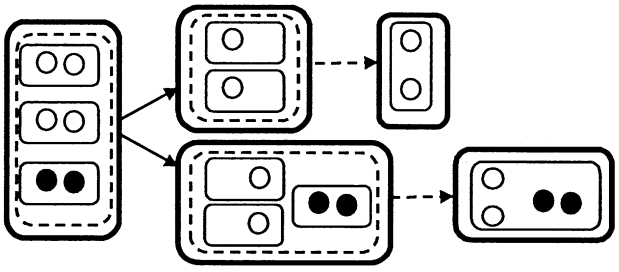

Fig. 4. Example of propagation algorithm execution that proceeds along concurrent e-views. The initial set of participants splits because of a view change.

each. Thus, the black processes and the gray process join the group with the respective wt set to $\{WT_0, \text{null}, 0, \text{null}, \text{null}\}$, whereas the white process joins with all fields of wt set to null. In this case, the first quorum subview is the one of black processes in ev_9.

Note, the propagation algorithm between black processes in ev_4 is not aborted by the view change corresponding to the installing of ev_7. Fig. 3 could correspond also to a recovery after a crash of all processes (the first e-view delivered to a process includes only the process itself). In this case there would be no difference with the previous discussion: each process would resume the respective wt from stable storage. Fig. 4 shows an execution in which the initial set of participants splits because of a view change and each resulting subset of participants continues the algorithm in the respective e-view.

5.4. Run-time costs

The run-time cost of the two algorithms can be summarized as follows.

- An execution of the update algorithm within a subview of k processes requires 2 totally ordered multicasts, k point-to-point messages and, at each process, 2 synchronous writes on the stable storage.
- An execution the propagation algorithm amongst n processes requires n point-to-point messages, 3 totally ordered multicasts (including those for sv-set and subview merging) and, at each process, 1 synchronous write on the stable storage. Note, this cost includes the application-level state transfer, that in practice has to be performed anyway.

The above costs could increase in case of view changes occurring before completion.

It may be useful to report the cost for an influential dynamic voting implementation, even though dynamic voting and our proposal are quite different (Section 2.4): an execution of the protocol in [19] within a view of n processes requires 2^*n multicasts and, at each process, 2 synchronous writes on the stable storage. This protocol must be run upon every view change and is independent of the application-level state transfer. Our update algorithm is run only when the application decides to and the propagation algorithm only upon a view expansion. Moreover, as just pointed out, the propagation algorithm includes actions that the application needs to perform anyway.

5.5. Selection of the "most recent" weight table

The coordinator selects the "most recent" wt replica at step P3 of the propagation algorithm, based on the wt replicas of all participants (Section 5.2). The selection procedure must be prepared to handle wt replicas reflecting uncompleted update attempts, because failures or view changes could cause a process to terminate the update algorithm prematurely. What makes this procedure quite complex is the fact that there might be wt replicas associated with the *same* version number but reflecting *different* uncompleted attempts. Such a scenario may result from:

- *Attempts initiated in concurrent quorum subviews*. For example, consider a quorum subview installed by all of its members and such that they all have the same wt replica. Suppose a view change occurs such that this subview splits in two or more concurrent quorum subviews (i.e., whose compositions overlap). Executions of ChangeWeight() may be initiated in all these subviews. Step U4 (collection of the UpdateAck from all members of the subview) may be completed in at most one such subview, but step U3 (update of

wt.prop and wt.updateid) may indeed be executed in all of them.

- *Attempts recorded only by processes that left the quorum subview*. Consider again a quorum subview installed by all of its members and such that they all have the same wt replica. Suppose a process p invokes ChangeWeight() but, because of a view change, p is the only process that delivers the associated UPDATEREQUEST message and thus executes step U3 (update of wt.prop and wt.updateid). Suppose another process q invokes ChangeWeight() in the new view and again, because of a view change, q is the only process that delivers the associated UPDATEREQUEST and executes step U3. The wt replicas at p and q describe different update attempts associated with the same version number.

The selection procedure executed by the coordinator, discussed in the next section, takes the PROPCOLLECT messages sent by all participants and returns a pair (decision, selectedWT) where the former is an enumerated and the latter is one of the wt replicas with the highest version number. The coordinator will multicast this pair to all participants with a PROPEND message (step P4) and then will merge all participants into a single subview (step P5). The actions of each participant p upon delivery of the e-view signalling completion of the algorithm are as follows:

- decision = Clear $\Rightarrow$ (*Clear all uncompleted attempts*) Assign $\mathrm{wt}_p :=$ selectedWT and determine the value of the quorum_p flag (Section 5.1). In this case, selectedWT.prop and selectedWT.propid are null.
- decision = Wait $\Rightarrow$ (*Leave uncompleted attempts unchanged*) Assign $\mathrm{wt}_p.\mathrm{curr}, \mathrm{wt}_p.\mathrm{currid}, \mathrm{wt}_p.\mathrm{vn}$ to the analogous fields of selectedWT. Leave wt_p.prop and wt_p.propid unchanged. Assign $\mathrm{quorum}_p :=$ false.
- decision = Propagate $\Rightarrow$ (*Complete one of the uncompleted attempts and clear the others*) Assign $\mathrm{wt}_p =$ selectedWT and determine the value of the quorum_p flag (Section 5.1). If $\mathrm{quorum}_p =$ true, then start an execution of the update algorithm from step U3 (Section 5.2). In this case, the values of selectedWT.prop and selectedWT.propid are those of one of the replicas with highest version number: they describe an uncompleted update attempt that has to be resumed.

5.6. Selection procedure

Given a weight table wt, predicate $Q(S, \mathrm{wt})$ is true if and only if the set of processes S collects a strict majority of the weights distribution in wt.curr. The selection procedure is given in Fig. 5 and discussed below. Significant examples are given in Section 5.7. The highest vn field is determined (lines 2–4; only fields prop and propid of the selectedWT might be changed in the following lines). The set of participants S is then partitioned in several subsets: S_{old}, with participants having a stale wt replica (line 5); S_0, with those having the highest vn and no uncompleted attempt; then the remaining participants are grouped so that processes in

```
Function selectNewWT (collected: set of PROPCOLLECT Messages): record
                                        { enum(Propagate, Clear, Wait), WeightTable };
1     S := < set of senders of elements in collected >;
2     tables := < set of weight tables in collected >;
3     selectedWT := any tx ∈ tables : ∀ty ∈ tables (ty≠tx) , ty.vn ≤ tx.vn;
4     highestVN := selectedWT.vn;
      // S is partitioned in k+2 disjoint sets, k being the number of uncompleted update attempts
      // collected[p] denotes the element of collected sent by process p
5     S_old :=    { p ∈ S : collected[p].wt.vn < highestVN };
6     S_0 :=      { p ∈ S : collected[p].wt.vn = highestVN and collected[p].wt.prop = null };
7     uncompletedUpdates := < set of wt.propId in collected >;
8     j := 1;
9     foreach upid ∈ uncompletedUpdates
        // Group processes with the same uncompleted attempt
10      S_j :=  { p ∈ S : collected[p].wt.vn = highestVN and collected[p].wt.update-id = upid };
11      j++;
12    end-foreach
13    multi-if
14      ♦ S = (S_old ∪ S_0) →                                  decision := Clear;                    //R1
15      ♦ S ≠ (S_old ∪ S_0) ∧ ¬Q(S, selectedWT) →              decision := Wait;                     //R2
16      ♦ S ≠ (S_old ∪ S_0) ∧ Q(S, selectedWT) ∧ ∀S_i, α_1(S_i) ∨ β_1(S_i) → decision := Clear;     //R3.1
17      ♦ S ≠ (S_old ∪ S_0) ∧ Q(S, selectedWT) ∧ ∃S_k : (β_3(S_k) ∧ ¬α_4(S_k)) ∧ ∀S_i ≠S_k, β_1(S_i) →
18                                                             decision := Propagate;                //R3.2
19                                                             p := any process ∈ S_k;
20                                                             selectedWT := collected[p].wt;
21      ♦ otherwise → decision := Wait;                                                              //R3.3
22    end-multi-if
23    if ( decision = Wait or decision = Clear ) then
24      { selectedWT.prop := null; selectedWT.wt.propId := null; }
25    return (decision, selectedWT);
```

Fig. 5. Selection procedure for step P3 of the propagation algorithm.

the same subset S_i have the highest vn and the same uncompleted attempt (8–12). The composition of the resulting subsets is analyzed and the return values are determined (lines 13–21). The branches of the multi-if statement are called *decision rules*:

- Rule R1 (decision = Clear) applies when there is no uncompleted attempt at the replicas with highest version number (line 14). Fields selectedWT.prop and selectedWT.update-id are null (23–25).
- Rule R2 (decision = Wait) applies when one of the replicas with highest version number has an uncompleted attempt and the set of participants does not define a quorum for the highest version number (line 15). That is, S is "too small" to decide safely whether the uncompleted attempt has to be cleared or completed.

The remaining decision rules are more complex. Predicates at lines 16 and 17 are defined in Table 3, which is based on the following definitions. Let $\mathrm{wt}(S_i)$ denote the replica wt_q at any process $q \in S_i$. Let OutS be the set of group members that are not in S, i.e., that are not participating in the algorithm execution. Given $p \in \mathrm{OutS}$, let $\mathrm{wt}_p(\mathrm{OutS})$ be the value of wt_p in an e-view concurrent to the e-view where the selection procedure is executed. Finally, let predicate $\mathrm{Installed}(S_i)$ be true iff $\exists p \in \mathrm{OutS} : \mathrm{wt}_p(\mathrm{OutS}).\mathrm{currid} = \mathrm{wt}(S_i).\mathrm{propid}$ or $wt_p(\mathrm{OutS}).\mathrm{vn} > \mathrm{wt}(S_i).\mathrm{vn}$.

- Rule R3.1 (decision = Clear) applies when the coordinator can tell that $\forall S_i \neg\mathrm{Installed}(S_i)$—i.e., all uncompleted attempts can be cleared safely.

  A simple reasoning by contradiction proves that $(\alpha_1(S_i) \cdot \vee \cdot \beta_1(S_i)) \Rightarrow \neg\mathrm{Installed}(S_i)$. If $\mathrm{Installed}(S_i)$ was true then a set of processes S_x defining a quorum in both selectedWT and $\mathrm{wt}(S_i)$ would have participated in the installing of $\mathrm{wt}(S_i)$: but if either $\alpha_1(S_i)$ or $\beta_1(S_i)$ holds then the existence of S_x can be excluded, thereby falling in a contradiction. Note, the coordinator may evaluate predicates $\alpha_1(S_i)$ and $\beta_1(S_i)$ as they do not require the knowledge of the *composition* of OutS (that is not available to the coordinator): it suffices to compute the weights collected in either $\mathrm{wt}(S_i)$ or selectedWT by processes that are not in S.
- Rule R3.2 (decision = Propagate) applies when the coordinator can tell that $\forall S_i \neq S_k \neg\mathrm{Installed}(S_i)$; and, in case $\mathrm{Installed}(S_k)$ was true, that no $\mathrm{wt}_p(\mathrm{OutS})$ may have a version number greater than highestVN (recall that $\mathrm{wt}(S_k).\mathrm{vn} = \mathrm{highestVN}$). In this case it is safe to clear all $\mathrm{wt}(S_i)$ and complete the installing of $\mathrm{wt}(S_k)$. Note that $\mathrm{wt}(S_k)$ exists *already* on a quorum of selectedWT because $\beta_3(S_i)$ holds.

  To prove that $\forall S_i \neq S_k \neg\mathrm{Installed}(S_i)$, the same argument as in R3.1 can be applied. To prove that no replica $\mathrm{wt}_p(\mathrm{OutS})$ may have a version number $\mathrm{vn}_x > \mathrm{highestVN}$ follows from another reasoning by contradiction: there should exist a set of processes $S_x \subseteq \mathrm{OutS}$ such that $\forall p \in S_x, \mathrm{wt.vn}_p \geqslant \mathrm{vn}_x$ and such that $Q(S_x, \mathrm{wt}(S_k))$ holds. But $\neg\alpha_4(S_k) \Rightarrow \neg Q(\mathrm{OutS}, \mathrm{wt}(S_k))$ thereby falling in a contradiction.
- Rule R3.3 (decision = Wait) applies when the coordinator cannot take any of the above decisions.

5.7. Examples of execution of the selection procedure

In this section we provide significant examples of execution of the selection procedure in the propagation algorithm (Section 5.5). The examples are such that the set of participants S defines a quorum in selectedWT and $S = S_0 \cup S_1$. It follows from Fig. 5 that the decision rule will be one of R3.1, R3.2, R3.3 (lines 13–22). More general examples can be constructed easily and do not provide any additional insight.

Consider e-view ev_1 in Fig. 6 and assume that: (i) ev_1 includes all group members; (ii) all group members have identical data structures and field wt.prop null; (iii) the weight table grants weight 1 to each group member, i.e., any set with 4 processes is a quorum. Suppose that an instance of the update algorithm is initiated in ev_1. The new desired value for the weight table, say wtNew, assigns weight 1 to the black process and weight 0 to all other processes. Finally, suppose a view change occurs while the algorithm is in progress, leading to the installing of ev_2, ev_3, ev_4. Let us analyze possible executions of the selection procedure depending on which processes assigned wt.prop := wtNew before the view change, that is, depending on the composition of S_1.

Table 3
Predicates for decision rules R3.1 and R3.2 in Fig. 5 (see also the appendix)

$\alpha_1(S_i) = \neg Q(\mathrm{OutS}, \mathrm{wt}(S_i))$	$\wedge$	$\neg Q(S_i, \mathrm{wt}(S_i))$	$\wedge$	$\neg Q(\mathrm{OutS} \cup S_i, \mathrm{wt}(S_i))$
$\alpha_2(S_i) = \neg Q(\mathrm{OutS}, \mathrm{wt}(S_i))$	$\wedge$	$\neg Q(S_i, \mathrm{wt}(S_i))$	$\wedge$	$\neg Q(\mathrm{OutS} \cup S_i, \mathrm{wt}(S_i))$
$\alpha_3(S_i) = \neg Q(\mathrm{OutS}, \mathrm{wt}(S_i))$	$\wedge$	$Q(S_i, \mathrm{wt}(S_i))$	$\wedge$	$Q(\mathrm{OutS} \cup S_i, \mathrm{wt}(S_i))$
$\alpha_4(S_i) = \neg Q(\mathrm{OutS}, \mathrm{wt}(S_i))$	$\wedge$	$\neg Q(S_i, \mathrm{wt}(S_i))$	$\wedge$	$\neg Q(\mathrm{OutS} \cup S_i, \mathrm{wt}(S_i))$
$\beta_1(S_i) = \neg Q(\mathrm{OutS}, \mathrm{selectedWT})$	$\wedge$	$\neg Q(S_i, \mathrm{selectedWT})$	$\wedge$	$\neg Q(\mathrm{OutS} \cup S_i, \mathrm{selectedWT})$
$\beta_2(S_i) = \neg Q(\mathrm{OutS}, \mathrm{selectedWT})$	$\wedge$	$\neg Q(S_i, \mathrm{selectedWT})$	$\wedge$	$\neg Q(\mathrm{OutS} \cup S_i, \mathrm{selectedWT})$
$\beta_3(S_i) = \neg Q(\mathrm{OutS}, \mathrm{selectedWT})$	$\wedge$	$Q(S_i, \mathrm{selectedWT})$	$\wedge$	$\neg Q(\mathrm{OutS} \cup S_i, \mathrm{selectedWT})$

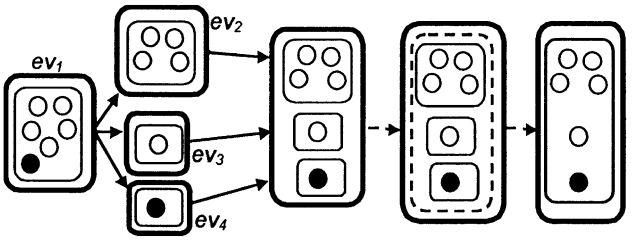

Fig. 6. Example of propagation algorithm (I).

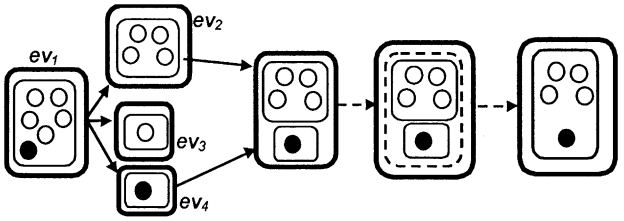

Fig. 7. Example of propagation algorithm (II). The initial scenario is as in Fig. 6 but the e-view evolution is different.

If S_1 does not include the black process, the decision will obviously be Clear (rule R3.1: $\neg Q(S_1,$ wtNew), thus $\alpha_1(S_1)$ holds). If S_1 does include the black process there are two cases: if S_1 includes the white processes in ev_2 the decision will be Propagate (rule R3.2 with $S_k = S_1 : Q(S_1,$ wtNew) and $Q(S_1,$ selectedWT), thus $\alpha_3(S_1)$ and $\beta_3(S_1)$ hold; recall that $S = S_0 \cup S_1$); otherwise, the decision will be Clear (rule R3.1: $\neg Q(S_1,$ selectedWT), thus $\beta_1(S_1)$ holds). Note, the quorum flag is false in ev_3 and ev_4; it is true in ev_2 only if its members did *not* record wtNew in the respective wt.prop, otherwise it is false.

Fig. 7 describes a different evolution in which the process in ev_3 does not participate in the propagation algorithm. If $S \neq S_1$, then the decision will be Clear (if S_1 includes only the black process then $\neg Q(S_1,$ selectedWT); if S_1 includes only the white processes then $\neg Q(S_1,$ newWT); thus, either $\alpha_1(S_1)$ or $\beta_1(S_1)$ holds). Note, the participants can conclude that clearing newWT is safe even though they do not know the state of the missing group member. If $S = S_1$, the decision is Propagate because S_1 is a quorum of *both* selectedWT and newWT.

Finally, Fig. 8 is similar to the previous case, except that here it is the black process that does not participate in the propagation. If S_1 does not include the processes in ev_2;, then the decision is Clear ($\neg Q(S_1,$ selectedWT) thus $\beta_1(S_1)$ holds). Otherwise, the decision is Wait: $Q(S_1,$ selectedWT), thus the participants can neither clear (they cannot exclude that the black process installed newWT) nor propagate (they do not define a quorum of newWT).

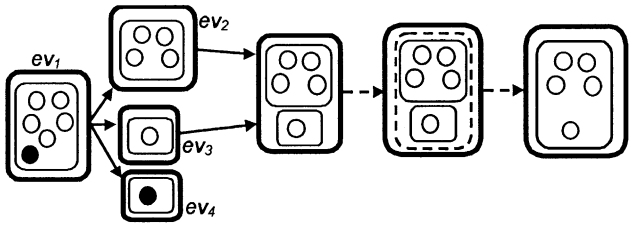

Fig. 8. Example of propagation algorithm (III). The Initial scenario is as in Fig. 6 but the e-view evolution is different.

5.8. *Creating a* quorum *subview*

We say that the propagation algorithm *creates* a quorum subview when the quorum flag is set in the resulting subview and it was not at each of the participating subviews. The latter may occur, for example, when the system recovers after experiencing a total failure, where all the processes crash. The quorum flag is set in the resulting subview when the set of participants S is a quorum in selectedWT and the selection procedure does not decide Wait (such a decision implies that the flag is forcibly set to false, see end of Section 5.5) The decision is not Wait when either:

- no uncompleted attempt is known at any member of S (Fig. 5, R1); or,
- clearing all uncompleted attempts is safe (R3.1); or,
- An uncompleted attempt $wt(S_k)$ exists at a subset $S_k \subseteq S$ such that S_k defines a quorum in both selectedWT and $wt(S_k)$ (R3.2).

The following theorem ensures that one of the above eventually occurs (see Appendix A for the proof):

Theorem. *If every failure recovers and the number of view changes is finite, then each e-view has a successor e-view* ev *such that* $p \in$ ev.memb *and* ev.quorum$_p$ *is set.*

6. Conclusions

We have presented a methodology and associated algorithms for establishing a primary partition in a partitionable distributed system. With our methodology, the membership service delivers views and the application applies the selection rule. In other words, it is not the membership service that decides whether a collection of (apparently) reachable group members is sufficient to make progress: such a decision is left up to the application. This clear layering makes it easy to support multiple applications with different and possibly conflicting notions of a primary partition. Our methodology supports groups with dynamic membership and is capable of re-establishing a primary partition automatically after recovering from a total failure, based on the last selection rule defined prior to the failure.

Each application may define its own selection rule and modify it at run-time according to its specific needs. This feature may simplify the implementation of "environment-aware" applications: the application collects relevant statistics during its execution in a particular environment, and based on these statistics and application-specific policies, the selection rule is modified as necessary. Adaptation may be performed on-line, without having to halt and restart the application. Such forms of application-driven adaptation are not possible with approaches where the primary partition is selected automatically by the system, based on a fixed and immutable policy (e.g., dynamic voting).

Acknowledgments

The authors are grateful to the anonymous referees for their detailed and constructive comments. This work has been partially supported by Microsoft Research (Cambridge, UK).

Appendix A. Proof of the Theorem in Section 5.7

The proof is based on the following lemmas. Let ev be the e-view where the selection procedure is executed and let $S = \text{ev.memb}$ (we use the same notation as in Fig. 5). Each lemma assumes that S includes all group members: a process $p \notin S$ either has never been a group member, or it left the group in an e-view ev′ predecessor of ev.

Lemma A.1. $\forall p \notin S, \text{selectedWT.curr}$ *grants zero weight to* p.

Proof. When p left the group, $\text{wt}_p\text{.curr}$ granted zero weight to p_1 by construction of the pseudo-code (see Appendix B). If $\text{wt}_p\text{.currid} = \text{selectedWT.currid}$ the lemma is proved. Otherwise, suppose selectedWT.curr grants non-zero weight to p. In this case, p must have participated in the installing of at least one of the weight tables with version number $\text{vn} \in (\text{wt}_p\text{.vn}, \text{selectedWT.vn}]$ (constraint C2). But p already left the group, thus we fall in a contradiction. □

Lemma A.2. $Q(S, \text{selectedWT})$ *holds*.

Proof. Follows trivially from Lemma A.1. □

Lemma A.3. $\forall S_i \neg Q(\text{OutS}, \text{prop}(S_i))$.

Proof. All members of OutS have zero weight in selectedWT (Lemma A.1). Any uncompleted attempt $\text{wt}(S_i)$ must satisfy constraint C2, thus it could assign non-zero weights to members of OutS but without making OutS a quorum of $\text{wt}(S_i)$. □

Lemma A.4. $\neg Q(S_K, \text{selectedWT}) \Rightarrow \neg Q(\text{OutS} \cup S_K, \text{selectedWT})$.

Proof. Follows trivially from Lemma A.1. □

The proof can now be constructed.

Theorem. *If every failure recovers and the number of view changes is finite, then each* e-*view has a successor* e-*view* ev *such that* $p \in \text{ev.memb}$ *and* ev.quorum_p *is set*.

Proof. Eventually, there will be an execution of the selection procedure including all group members, i.e., such that the above lemmas can be applied. If there are no uncompleted attempts ($S = (S_{\text{old}} \cup S_0)$), rule R1 will be taken. If there are uncompleted attempts ($S \neq (S_{\text{old}} \cup S_0)$), rule R2 will not be taken because of Lemma A.2. Thus, let us assume $S \neq (S_{\text{old}} \cup S_0)$ and $Q(S, \text{selectedWT})$.

- $\neg\exists \text{wt}(S_k) : Q(S_k, \text{selectedWT}) \Rightarrow \forall S_i$, either $\beta_1(S_i)$ or $\beta_2(S_i)$ holds. But $\beta_2(S_i)$ cannot hold because of Lemma A.4. Thus, R3.1 will be taken.
- $\exists \text{wt}(S_k) : Q(S_k, \text{selectedWT}) \Rightarrow \beta_3(S_k)$ holds (because of A.1). Moreover, $\forall S_i \neq S_k$, it is $\beta_1(S_i)$ ($\beta_3(S_i)$ does not hold because $S_i \cap S_k = \emptyset$ thus $\neg Q(S_i, \text{selectedWT})$; $\beta_2(S_i)$ cannot hold because of A.1). Finally, $\alpha_4(S_k)$ cannot hold because of Lemma A.3. □

Appendix B. Pseudo-code

We present full details about our algorithms in a pseudo-code with a Pascal-like syntax and a self-explaining notation. The API operations in Table 2 are implemented as follows (identifier myld denotes the name of the executing process).

- join() activates the WT-layer threads (see below) and suspends the invoking thread. The operation returns as soon as the invoking thread is awakened, which occurs when the WT-layer threads terminate.
- send() and mcast() invoke the analogous operations of the EnrVS layer, denoted EnrVS.send() and EnrVs.mcast().
- changeWeight(wtNew) issues EnrVS.send(⟨ REQUEST UPDATE, wtNew ⟩, myld).
- readWeight() issues EnrVS.send(⟨ REQUEST WEIGHT ⟩, myld).
- leave() issues EnrVS.send(⟨ REQUESTLEAVE ⟩, myld).
- changeWeight() checks constraints C1–C3 of Section 4.1. If all of them are satisfied then the operation issues EnrVS.mcast(⟨ REQUESTUPDATE, wtNew ⟩, mySV) otherwise it issues EnrVS.send (⟨ WTCHANGE, wtNew, NotPossible ⟩, myld).

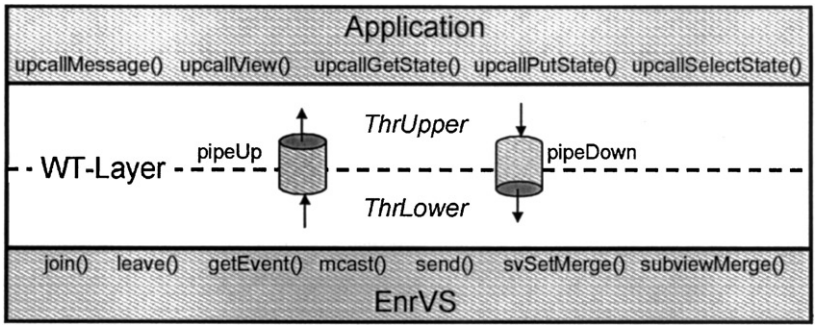

Fig. 9. Architecture of the WT-layer (EnrVS.send() and EnrVS.mcast() may be invoked also from within upcallMessage() and upcallView()).

```
Thread ThrUpper()
    repeat
        e := pipeUp.get();
        multi-if
            ♦ e is e-view change →      upcallChange(e);
            ♦ e is STATEGET →           pipeDown.put( upcallGetState() );
            ♦ e is STATESELECT →        pipeDown.put( upcallSelectState(e.statearray) );
            ♦ e is STATEPUT →           upcallPutState(e.state);
            ♦ e is LEAVEOK →            upcallMessage(e);
                                        signal application thread blocked on join();
                                        terminate;
            ♦ otherwise →               upcallMessage(e);
        end-multi-if
    forever
```

Fig. 10. Code for the ThrUpper thread.

```
quorumFlag:     Boolean initial false;
waitingFlag:    Boolean initial false;          // When true, quorumFlag must be kept false
ev:             EviewChange initial null;       // Last e-view change
updateReqQ:     Queue of Messages initial empty; // Pending update requests
pCoord:         ProcessId initial null;         // Identity of propagation coordinator
collected:      Set of Messages initial empty;  // Received PROPCOLLECT messages
// The actual wt replica.
// Writes on the stable storage must be performed as a synchronous atomic action.
wt: record of
    curr, prop :  WeightTable initial null;
    vn :          Integer initial null;
    update-id :   UpdateIdentifier initial null;
end-record;
```

Fig. 11. Variables for ThrLower() (visible to all procedures of this thread).

The WT-layer consists of two threads, called *ThrLower* and *ThrUpper* (Fig. 9). The former interacts with the EnrVS layer and implements the update and propagation algorithms. The latter receives events from ThrLower and executes upcalls to the application layer. The two threads interact in a producer–consumer way through two pipe objects, called pipeUp and pipeDown. ThrLower inserts a message into pipeUp through the put() operation, whereas ThrUpper extracts a message from pipeUp through the (blocking) get() operation. The pipeDown object is for the reverse path. Most exchanges occur through pipeUp—pipeDown is used only for returning the results of upcallGetState() and upcallSelectState() to ThrLower. These two pipes are the *only* objects shared amongst threads.

The implementation of ThrUpper is rather simple: see Fig. 10, that should be self-explanatory (tags beginning with STATE denote messages originated by ThrLower during the propagation algorithm, see below). The implementation of ThrLower is more complex. While the update and propagation algorithms themselves would not be difficult to describe, a single piece of code that integrates the two algorithms, including the interaction with ThrUpper and the handling of application-originated messages, involves many low-level details. On the other hand, the resulting description is quite close to an actual implementation.

ThrLower maintains the variables in Fig. 11 and executes the code in Fig. 12, discussed below. ThrLower executes an endless loop where each iteration begins by receiving an event from the EnrVS layer (line 5). We consider a few simple examples. First, consider an e-view composed of a single subview whose members have the same wt replica and quorumFlag set. Suppose a

```
Thread ThrLower()
1)  <Initialize global variables >;
2)  if not first bootstrap then wt.stableRead();          // restore wt from the stable storage
3)  EnrVs.join();
4)  repeat
5)      e := EnrVS.getEvent();
6)      multi-if
7)          ♦ e is e-view change →   ev := e; updateQuorum(); pipeUp.put(<e, quorumFlag>);
8)                                   if (#subviews in ev > 1) then propagation();
9)          ♦ otherwise →            handleApplicationMessage(e);        // Figure 13
10)     end-multi-if

11)     while (updateReqQ ≠ ∅)
12)         e := updateReqQ.get();
13)         completeUpdate(e);                                           // Figure 14
14)         if (#subviews in ev > 1) then completePropagation();         // Figure 15
15)     end-while
16) forever

Procedure propagation();
    repeat
        scheduleStepP1();
        // Wait for creation of the sv-set
        repeat
            e := EnrVS.getEvent();
            if e is e-view change then {ev := e; updateQuorum(); pipeUp.put(<e, quorumFlag>);}
            else   handleApplicationMessage(m);
        until (e is e-view change);
        if (#subviews in ev = 1) then return;          // No state exchange required
    until (#subviews in ev.svSet(myId) > 1)
    completePropagation();
```

Fig. 12. Code for ThrLower (I).

```
// Handle messages from the application layer
Procedure handleApplicationMessage(m)
    multi-if
        ♦ m is UPDATEREQUEST →      updateReqQ.put(m);
        ♦ m is LEAVEREQUEST →       if (< zero weight in wt.curr and wt.prop >) then
                                      { EnrVS.leave(); pipeUp.put(LEAVEOK); terminate thread;}
                                    else pipeUp.put(LEAVEREFUSED);
        ♦ otherwise →               pipeUp.put(m);        // Application-originated message
    end-multi-if

// Update quorum flag
Procedure updateQuorum()
    if waitingFlag then
        quorumFlag := false;
    else   quorumFlag := Q(memb(MySV), wt.curr);
        if (wt.prop ≠ null) then
            quorumFlag := quorumFlag and Q(memb(MySV), wt.prop);
    end-if

// Step P1 of the propagation algorithm. Can be invoked while the update algorithm is in progress.
Procedure scheduleStepP1();
    participants := { p | p ∈ ev.memb and (#subviews ∈ ev.svSet(p) = 1) };
    pCoord := elect(participants);
    if (myId = pCoord) then
        sv-sets := ∅; foreach p∈participants do sv-sets += ev.svSet(p);
        EnrVS.svSetMerge(sv-sets);
    end-if
    collected := ∅;
```

Fig. 13. Code for ThrLower: Ancillary procedures.

member of this e-view invokes ChangeWeight(). As described above, this invocation is performed by ThrUpper while executing an upcall and results in the multicasting of an UPDATEREQUEST message within the subview. This message is delivered to ThrLower (line 5). Each process inserts the message into the updateReqQ queue (line 9), then invokes procedure completeUpdate() (Fig. 13). This procedure is shown in Fig. 14 and implements the steps of the update algorithm discussed in Section 5.2. Understanding is simplified by assuming that no view expansions occur during execution of this procedure.

Then, consider a process p that is not part of the initial membership and bootstraps for the first time. The

```
// Implements the update algorithm from step U2
Procedure completeUpdate(msg: UPDATEREQUEST message);
    // Step U2
    if (waitingFlag or ( e.newWT does not satisfy one of C1, C2, C3 in Section 4.1 )) then
        { pipeUp.put( < WTCHANGE, e, Unknown >); return; }
    wt.prop := e.newWT; wt.update-id := e.upid;  wt.stableWrite();
    // Step U3
    uCoord := elect(MySV);
    Send(UPDATEACK, uCoord);
    // Beginning of U4 (pCoord) and U5 (other participants)
    acked := ∅; propagationInProgress := false;
    while (true)
        e := EnrVS.getEvent ();
        multi-if
            ♦ e is UPDATEACK message →                                  // Happens only at uCoord
                acked += e.sender;
                if (acked = mySV.memb) then EnrVS.mcast (UPDATECOMMITTED, MySV);   // Step U4
            ♦ e is UPDATECOMMITTED message →                                       // Step U5
                wt.curr := wt.prop; wt.vn := wt.vn + 1; wt.prop := null; wt.update-id := null;
                wt.stableWrite();
                pipeUp.put( < WTCHANGE, msg, Ok >);
                return true;
            ♦ e is e-view change →
                ev := e; updateQuorum(); pipeUp.put(< e, quorumFlag >);

                acked -= elements from senders ∉ ev.memb;
                multi-if
                    ♦ quorumFlag and acked = mySV.memb →
                                    EnrVS.mcast(UPDATECOMPLETED, MySV);        // Step U4
                    ♦ quorumFlag and uCoord ∉ ev.memb →                        // Restart U3
                                    uCoord := elect(mySV);
                                    EnrVS.Send(UPDATEACK, uCoord);
                    ♦ not quorumFlag→   pipeUp.put ( < WTCHANGE, e, Unknown >); return;
                end-multi-if

                // Handle possible overlap with step P1
                if (propagationInProgress) then
                    multi-if
                        ♦ #subviews ∈ ev = 1 →                  propagationInProgress := false;
                        ♦ #subviews ∈ ev.svSet(myId) > 1 →      no-op;                  // Step P1 completed
                        ♦ otherwise →                           scheduleStepP1();
                    end-multi-if
                elseif (#subviews ∈ ev > 1) then
                    { doStepP1(); propagationInProgress := true; }                  // Begin Step P1
            ♦ e is PROPCOLLECT message →    collected += e;          // Only if P1 completed and
                                                                     // this process is pCoord
            ♦ otherwise →                   handleApplicationMessage(e);   // Figure 12
        end-multi-if
    end-while
```

Fig. 14. Code for ThrLower (II): update algorithm.

first event that p receives from the EnrVS layer is an e-view that includes only p. The event is forwarded to the application layer (line 7) and then execution skips to the next iteration of the loop. If p invokes ChangeWeight(), execution proceeds as above except that completeUpdate() will return immediately as the very first check will fail (step U2).

Next, suppose p delivers an e-view including p and the quorum subview. There may be the following cases:

- *No execution of the update algorithm is in progress*. All processes will invoke propagation() (line 7). This procedure creates an sv-set encompassing all members of the e-view (understanding of this part is simplified by assuming that no view changes occur before the creation of the sv-set) and then invokes corePropagation() which implements the steps of the propagation algorithm discussed in Section 5.2. What makes this procedure somewhat complex is the management of view changes and application-generated messages. Assuming that these events do not occur simplifies the understanding. All processes will receive a copy of the new application-level state, returned by upcallSelectState() (in practice, this will be the state of one of the processes in the quorum subview); a copy of wt at any process in the quorum subview; Clear as decision value (decision rule R1).
- *An execution of the update algorithm is in progress*. Process p will invoke propagation() (Fig. 12, line 7) whereas the other processes will continue execution of completeUpdate(). If p happens to elect itself as pCoord, it will wait for PROPCOLLECT messages from the other processes that will send these messages only upon returning from completeUpdate() (Fig. 12,

```
// Implements the propagation algorithm from step P2
Procedure completePropagation();
    endMsg := null;
    while (#subviews in mySVSet > 1)
        delayedQueue := ∅;                    // Queue for delaying message delivery (step P2)
        // Step P3
        pipeUp.put(STATEGET);
        EnrVS.Send(<PROPCOLLECT, pipeDown.get(), wt>, pCoord);

        repeat
            e := EnrVS.getEvent();
            multi-if
                ♦ e is PROPCOLLECT message →    collected += e;
                ♦ e is PROPEND message →        endMsg := e;
                ♦ e is e-view change →          ev := e;
                        if (e is view change) then
                            // Restart from P2
                            while (delayedQueue ≠ ∅)
                                { e := delayedQueue.get(); pipeUp.put(e); }
                            updateQuorum(); pipeUp.put(< ev, quorum >);
                        else    collected -= elements from members ∉ ev.memb;
                            if (senders in collected = mySVSet.memb) then
                                // Step P4
                                { decision, newWT } := selectNewWT(collected);
                                pipeUp.put(<STATESELECT, collected.state >);
                                newState := pipeDown.get();
                                EnrVS.mcast( < PROPEND, newState,
                                                decision, newWT >, mySVSet);
                                EnrVS.subviewMerge(mySVSet.memb);
                            end-if
                        end-if
                ♦ otherwise →                   delayedQueue.put(e);
            end-multi-if
        until (#subviews in mySVSet = 1 or pCoord ∉ ev.memb or e is view change)
        if (pCoord ∉ ev.memb) then pCoord := elect(mySVSet.memb);
    end-while
    if (endMsg = null) then return;       // While loop exited before receiving PROPEND message

    // Step P5
    switch(endMsg.decision)

        case Propagate or Clear:
            waitingFlag := false;
            wt := endMsg.wt;
            pipeUp.put(<STATESELECT, endmsg.newState >);
        case Wait:
            waitingFlag := true;
            if (msg.wt.vn > wt.vn) then wt := { msg.wt.curr, msg.wt.vn; null, null }
    end-switch
    wt.stableWrite();
    updateQuorum(); pipeUp.put(< ev, quorum >);

    while (delayedQueue ≠ ∅)
        { e := delayedQueue.get(); pipeUp.put(e); }
    if (endMsg.decision = Propagate and wt.prop ≠ null) then
        completeUpdate();
```

Fig. 15. Code for ThrLower (III): propagation algorithm.

line 14). If p happens to elect one of the other processes as pCoord, the pCoord will simply store the PROPCOLLECT from p (near the end of completeUpdate()) and will process this message as above, upon returning from completeUpdate().

The above examples should suffice to understand the overall structuring.

The pseudo-code does not show: management of FULLYINSTALLED messages (Section 3.2); support for the operation readWeight(). We discuss these aspects below as including them in the pseudo-code would have introduced further unnecessary details. All the actions below are performed by ThrLower, unless stated otherwise.

Upon delivery of a view change, a designated process p in the new view ev is selected deterministically and each view member sends to p an INSTALLED message carrying the identifier of ev, denoted ev.id (any VS implementation attaches a system-wide unique identifier to each view it delivers). When p delivers an INSTALLED message not carrying the identifier to its current view,

p ignores the message. When p has delivered an INSTALLED message from each member of its current view, p multicasts a FULLYINSTALLED message including ev.id. A process that delivers a FULLYINSTALLED message compares the ev.id in the message with the identifier of its current view. If the identifiers match and quorumFlag is set, then the process forwards the FULLYINSTALLED message to ThrUpper, otherwise it ignores the message.

Finally, when ThrUpper invokes readWeight() (from within an upcall), a message with a dedicated tag is sent to ThrLower at the same process. Upon delivery, a message $\langle \text{WTVALUE}, \text{wt.curr} \rangle$ will be delivered to ThrUpper (see Fig. 15).

References

[1] Y. Amir, G. Chokler, D. Dolev, R. Vitenberg, Efficient state transfer in partitionable environments, Proceedings of the European Research Seminar in Advanced Distributed Systems (ERSADS97), March 1997.

[2] Y. Amir, D. Dolev, S. Kramer, D. Malki, Transis: a communication sub-system for high availability, Proceedings of the 22nd Symposium on Fault-Tolerant Computing, July 1992, pp. 76–84.

[3] Y. Amir, C. Tutu, From total order to database replication, Proceedings of the IEEE International Conference on Distributed Computing System, 2002, to appear.

[4] Ö. Babaoglu, A. Bartoli, G. Dini, On programming with view synchrony, Proceedings of the 16th IEEE International Conference on Distributed Computing Systems, May 1996, pp. 3–10.

[5] Ö. Babaoglu, A. Bartoli, G. Dini, Enriched view synchrony: a programming paradigm for partitionable asynchronous systems, IEEE Trans. Comput. 46 (6) (June 1997) 642–658.

[6] Ö. Babaoglu, R Davoli, L. Giachini, P. Sabattini, The inherent cost of strong-partial view synchronous communication, in: Distributed Algorithms (WDAG9), Lecture Notes in Computer Science, Vol. 972, Springer, Berlin, October 1995, pp. 72–86.

[7] Ö. Babaoglu, R. Davoli, A. Montresor, Group communication in partitionable systems: specification and algorithms, IEEE Trans. Software Eng. 27 (4) (April 2001) 308–336.

[8] Ö. Babaoglu, R Davoli, A. Montresor, R. Segala, System support for partition-aware network applications, Proceedings of the 18th IEEE International Conference on Distributed Computing Systems, Amsterdam, May 1998.

[9] D. Barbara, H. Garcia-Molina, A. Spauster, Increasing availability under mutual exclusion constraints, ACM Trans. Comput. Systems. 7 (4) (November 1989) 392–426.

[10] M. Bearden, R. Bianchini, A fault-tolerant algorithm for decentralized on-line quorum adaptation, Proceedings of the 28th International Symposium on Fault-Tolerant Computing, June 1998, pp. 262–271.

[11] K. Birman, The process group approach to reliable distributed computing, Comm. ACM 36 (12) (December 1993) 37–53.

[12] K. Birman, Virtual synchrony model, in: Reliable Distributed Computing with the Isis Toolkit, IEEE CS Press, Silver Spring, MD, 1994.

[13] K. Birman, A review of experiences with reliable multicast, Software—Practice Exp. 29 (9) (July 1999) 741–774.

[14] K. Birman, R. Cooper, T. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, M. Wood, The Isis System Manual, Version 2.1, Dept. of Computer Science, Cornell University, September 1993.

[15] J. Bloch, D. Daniels, A. Spector, Weighted voting for directories: a comprehensive study, Technical Report CMU-CS-84-114, Carnegie-Mellon University, April 1984.

[16] J. Bloch, D. Daniels, A. Spector, A weighted voting algorithm for replicated directories, J. ACM 34 (4) (October 1987) 859–909.

[17] D. Davcev, W.A. Burkhard, Consistency and recovery control for replicated files, Proceedings of the 10th ACM Symposium on Operating Systems Principles, December 1985, pp. 87–96.

[18] S. Davidson, H. Garcia-Molina, D. Skeen, Consistency in partitioned networks, ACM Comput. Surveys 17 (3) (1985) 341–370.

[19] D. Dolev, I. Keidar, E. Lotem, Dynamic voting for consistent primary components, Proceedings of the 16th ACM Symposium on Principles of Distributed Computing, August 1997.

[20] D. Dolev, D. Malki, R. Strong, A framework for partitionable membership service, Technical Report CS95-4, Institute Of Computer Science, The Hebrew University of Jerusalem, 1995.

[21] R. De Prisco, A. Fekete, N. Lynch, A. Shvartsman, A dynamic view-oriented group communication service, Proceedings of the 17th ACM Symposium on Principles of Distributed Computing, June 1998, pp. 227–236.

[22] R. De Prisco, A. Fekete, N. Lynch, A. Shvartsman, A dynamic primary configuration group communication service, Proceedings of the 13th International Symposium on Distributed Computing (DISC), Lecture Notes in Computer Science, Vol. 1693, 1999, pp. 64–78.

[23] A. El Abbadi, D. Skeen, F. Cristian, An efficient fault-tolerant protocol for replicated data management, Proceedings of ACM Symposium on Principles of Database Systems, April 1985, pp. 215–229.

[24] A. El Abbadi, S. Toueg, Maintaining availability in partitioned replicated databases, ACM Trans. Database Systems 14 (2) (June 1989) 264–290.

[25] A. Fekete, N. Lynch, A. Shvartsman, Specifying and using a partitionable group communication service, Proceedings of the 16th ACM Symposium on Principles of Distributed Computing, August 1997, pp. 53–62.

[26] R. Friedman, R. Van Renesse, Strong and weak virtual synchrony in horus, Technical Report TR95-1537, Dept. of Computer Science, Cornell University, March 1995.

[27] R. Friedman, A. Vaysburd, Fast replicated state machines over partitionable networks, Proceedings of the 16th IEEE Symposium on Reliable Distributed Systems, Durham, North Carolina, October 1997, pp. 130–137.

[28] H. Garcia-Molina, D. Barbara, How to assign votes in a distributed system, J. ACM 32 (4) (1985) 841–860.

[29] D. Gifford, Weighted voting for replicated data, Proceedings of the Seventh Symposium on Operating Systems Principles, ACM, 1979, pp. 150–62.

[30] B. Glade, K. Birman, R. Cooper, R. van Renesse, Light-weight process groups in the Isis system, Distrib. Systems Eng. (July 1993).

[31] J. Gray, Notes on database operating systems, in: Operating Systems—An Advanced Course, Lecture Notes on Computer Science, Vol. 66, Springer, Berlin, 1978, pp. 393–481.

[32] Group Services Programming Guide and Reference (Version 2, Release 2), IBM Parallel Systems Support Programs for AIX, Document Number SC28-1675-00, November 1996.

[33] M. Hayden, The ensemble system, Ph.D. Thesis, Department of Computer Science, Cornell University, 1998.

[34] M. Herlihy, A quorum-consensus replication method for abstract data types, ACM Trans. Comput. Systems 4 (1) (February 1986) 32–53.

[35] M. Herlihy, Dynamic quorum adjustment for partitioned data, ACM Trans. Databases 12 (2) (June 1987) 170–194.

[36] M. Hiltunen, R. Schlichting, A configurable membership service, IEEE Trans. Comput. 47 (5) (May 1998).

[37] K. Ingols, I. Keidar, Availability study of dynamic voting algorithms, Proceedings of the 21st International Conference on Distributed Computing Systems, April 2001, pp. 247–254.
[38] S. Jajodia, D. Mutchler, Dynamic voting algorithms for maintaining the consistency of a replicated database, ACM Trans. Database Systems 15 (2) (June 1990) 230–280.
[39] A. Joseph, A. deLespinasse, J. Tauber, D. Gifford, M. Kaashoek, Rover: a toolkit for mobile information access, Proceedings of the 15th ACM Symposium on Operating Systems Principles, Copper Mountain, Colorado, December 1995, pp. 156–171.
[40] F. Kaashoek, A. Tanenbaum, Group communication in the Amoeba distributed operating system, Proceedings of the 12th IEEE International Conference on Distributed Computing Systems, May 1991, pp. 222–230.
[41] B. Kemme, G. Alonso, A new approach to developing and implementing eager database replication protocols, ACM Trans. Database Systems 25 (3) (September 2000) 335–379.
[42] B. Kemme, G. Alonso, Don't be lazy, be consistent: Postgres-R, a new way to implement Database Replication, Proceedings of the 26th International Conference on Very Large Databases (VLDB), September 2000.
[43] B. Kemme, A. Bartoli, Ö. Babaoglu, On-line reconfiguration in replicated databases based on group communication, Proceedings of the IEEE/IFIP Dependable Systems and Networks, 2001, pp. 117–126.
[44] J. Kistler, M. Satyanarayanan, Disconnected operation in the Coda file system, Proceedings of the 13th ACM Symposium on Operating Systems Principles, October 1991, pp. 213–225.
[45] B. Lampson, Atomic transactions, in: Distributed Systems—Architecture and Implementation, Lecture Notes on Computer Science, Vol. 105, Springer, Berlin, 1981, pp. 246–265.
[46] L. Lamport, The part-time parliament, ACM Trans. Comput. Systems 16 (2) (May 1998) 133–169.
[47] B. Lampson, H. Sturgis, Crash recovery in a distributed data storage system, Technical Report, Xerox Palo Alto Research Center, Palo Alto, California, April 1979.
[48] N. Lynch, A. Shvartsman, Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts, Proceedings of the 27th International Symposium on Fault-Tolerant Computing, June, 1997.
[49] C. Malloth, Conception and implementation of a toolkit for building fault-tolerant distributed applications in large scale networks, Ph.D. Thesis, EPFL, Lausanne 1996.
[50] C. Malloth, A. Schiper, View synchronous communication in large-scale networks, Proceedings of the 15th ACM Symposium on Operating Systems Principles, December 1995, pp. 143–155.
[51] L.E. Moser, Y. Amir, P.M. Melliar-Smith, D.A. Agarwal, Extended virtual synchrony, Proceedings of the 14th International Conference on Distributed Computing Systems, June 1994, pp. 56–65.
[52] L. Mummert, M. Ebling, M. Satyanarayanan, Exploiting weak connectivity for mobile file access, Proceedings of the 2nd Open Workshop of the ESPRIT Project Broadcast, July 1995.
[53] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, G. Alonso, Scalable replication in database cluster, Proceedings of DISC, 2000.
[54] A. Ricciardi, K. Birman, Consistent process membership in asynchronous environments, Reliable Distributed Computing with the Isis Toolkit, IEEE CS Press, Silver Spring, MD, 1994.
[55] M. Satyanarayanan, H. Mashburn, P. Kumar, D. Steere, J. Kistler, Lightweight recoverable virtual memory, ACM Trans. Comput. Systems 12 (1) (February 1994) 33–57.
[56] A. Schiper, A. Sandoz, Primary partition "virtually-synchronous communication" harder than consensus, in: Distributed Algorithms (WDAG8), Lecture Notes in Computer Science, Vol. 857, Springer, Berlin, October 1994, pp. 39–52.
[57] D. Skeen, Determining the last process to fail, ACM Trans. Comput. Systems 3 (1) (February 1985) 15–30.
[58] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, C. Hauser, Managing update conflicts in Bayou, a weakly connected replicated storage system, Proceedings of the 15th ACM Symposium on Operating Systems Principles, Copper Mountain, Colorado, December 1995, pp. 172–183.
[59] R. Thomas, A majority consensus approach for multiple copy databases, ACM Trans. Database Systems 4 (2) (1979) 180–209.
[60] R.van Renesse, K. Birman, R. Cooper, B. Glade, P. Stephenson, The Horus system, Reliable Distributed Computing with the Isis Toolkit, IEEE CS Press, Silver Spring, MD, 1994.
[61] W. Vogels, D. Dumitriu, K. Birman, R. Gamache, R. Short, J. Vert, M. Massa, J. Barrera, J. Gray, The design and architecture of the microsoft cluster service—a practical approach to high availability and scalability, Proceedings of the 28th Symposium on Fault-Tolerant Computing, June 1998.