Enriched View Synchrony: A Programming Paradigm for Partitionable Asynchronous Distributed Systems

Özalp Babaoglu, Alberto Bartoli, and Gianluca Dini

Abstract—Distributed systems constructed using off-the-shelf communication infrastructures are becoming common vehicles for doing business in many important application domains. Large geographic extent due to increased globalization, increased probability of failures, and highly dynamic loads all contribute toward a *partitionable* and *asynchronous* characterization for these systems. In this paper, we consider the problem of developing reliable applications to be deployed in partitionable asynchronous distributed systems. What makes this task difficult is guaranteeing the consistency of shared state despite asynchrony, failures, and recoveries, including the formation and merging of partitions. While view synchrony within process groups is a powerful paradigm that can significantly simplify reasoning about asynchrony and failures, it is insufficient for coping with recoveries and merging of partitions after repairs. We first give an abstract characterization for shared state management in partitionable asynchronous distributed systems and then show how views can be enriched to convey structural and historical information relevant to the group's activity. The resulting paradigm, called *enriched view synchrony*, can be implemented efficiently and leads to a simple programming methodology for solving shared state management in the presence of partitions.

Index Terms—Large-scale distributed systems, group communication, fault tolerance, reliable network applications, shared state management.

1 Introduction

Distributed computing is rapidly becoming the principal paradigm for providing critical services in everyday life and the deployment of future networking technologies will only accelerate this trend. Large geographic extent due to increased globalization and unpredictability of loads imposed by users contribute towards an *asynchronous* characterization for these systems in the sense that communication delays and relative computing speeds cannot be bounded with certainty. Banking, finance, commerce, medical systems, telecommunications, industrial process control, and collaborative work are just some of the many sectors that will increasingly rely on large-scale asynchronous distributed systems as their computing infrastructure. Distributed applications to be deployed in such systems are difficult to reason about and to develop. The principal difficulty stems from the fact that in asynchronous distributed systems subject to failures, inability to communicate cannot be attributed to its real cause—the destination may have crashed, it may be overloaded and thus slow, the communication path may have been disconnected, or it may be experiencing long delays [1].

An abstraction that can simplify both reasoning about and implementation of distributed applications is *view synchrony*[1] in the context of process groups [3], [4], [5], [6]. Two aspects of view synchrony enable it to hide most of the complexities due to failures and asynchrony. On the one hand, it cleanly transforms failures into group membership changes through *views* that are agreed upon by all connected members of the group. On the other hand, view synchrony provides guarantees about the set of messages delivered globally as a function of the view changes that a process observes locally. As such, it permits components of a group to reason globally based solely on local information.

Partitions that may result from communication failures are an insidious characteristic of large-scale distributed systems. Furthermore, inability to bound delays due to asynchrony may lead to the formation of virtual partitions that are indistinguishable from real ones [7]. Partitions, whether real or virtual, tend to become more frequent and last longer as the geographic extent of the system grows. Informally, we define a *partitionable system* as one admitting multiple views of the same group to exist concurrently. In such systems, membership of a group may change dynamically, not only due to individual process failures and recoveries, but also due to subsets of correct processes becoming disconnected and later reconnecting. Each collection of mutually-communicating processes may install their own view of the group without waiting for the failures that caused the partition to be repaired. This is in contrast to the *primary-partition group membership* model where there can be at most one view of the group active at any time [2], [8].

• Ö. Babaoglu is with the Department of Computer Science, University of Bologna, Mura Anteo Zamboni 7, 40127 Bologna, Italy.
E-mail: ozalp@CS.UniBO.IT.
• A. Bartoli and G. Dini are with the Dipartimento di Ingegneria dell'Informazione, University of Pisa, Via Diotisalvi 2, 56126 Pisa, Italy.
E-mail: {alberto, gianluca}@iet.unipi.it.

For information on obtaining reprints of this article, please send e-mail to: transcom@computer.org, and reference IEEECS Log Number 104649.0.

1. The abstraction was first introduced in the Isis system where it is known as *virtual synchrony* [2]. We prefer not to use this term since it is associated with the primary-partition model of group membership that excludes the possibility of progress in multiple concurrent partitions.

0018-9340/97/$10.00 © 1997 IEEE

For certain application classes, including mobile computing, loose-consistency data sharing, collaborative work, and scientific computing, progress may indeed be possible in multiple concurrent partitions. To minimize latency in such applications, a *partitionable group membership* service installs views (perhaps concurrent ones) at all correct processes without undue delays and lets the application itself decide if it can make progress.

In this paper, we consider programming reliable applications in partitionable asynchronous distributed systems based on process groups and view synchrony. Group members have to maintain state information that is distributed and/or replicated among them. Although view synchrony can be a great aid towards guaranteeing the consistency of this information, many technical problems remain that need to be solved by the application programmer. We first give a characterization of these *shared state* problems in terms of system events provoking them: Processes joining a group give rise to the *state transfer* problem; recovery from total failures leads to the *state creation* problem; unification of two or more concurrent partitions after repairs has to solve the *state merging* problem. We show that view synchrony alone is not sufficient in coping with these problems in that scenarios provoking them usually do not permit global reasoning with local information alone. Thus, most of the burden in solving shared state problems falls on the application programmer and detracts from the simplicity and elegance of view synchrony. We then propose an extension to the basic model by including structural and historical information within views in the form of *subviews* and *subview sets* that are manipulated by processes to reflect the application state and are preserved automatically across view changes by the system. Our extension is called *enriched view synchrony* and offers a simple programming methodology for solving shared state problems that are encountered when programming reliable services in partitionable asynchronous systems. We illustrate this methodology through detailed examples. Finally, we sketch how enriched view synchrony can be implemented efficiently through simple extensions to a typical view synchrony service.

2 System Model and View Synchrony

The system is a collection of processes executing at potentially remote sites that communicate through a network. As a result of failures, processes may crash and the communication network may partition. Crashes cause processes to halt prematurely. Crashed processes may rejoin the computation after recovery and partitions may merge after repairs. We consider an *asynchronous* system in that it is not possible to place bounds on communication delays or relative speeds of processes. This is a realistic way of taking into account delays due to transient failures, unknown scheduling strategies, and dynamic load on the computing and communication resources of most practical distributed systems.

In any distributed system, whether a remote process has crashed or not can be inferred by a local *failure detector* only indirectly, through messages received from that process. In an asynchronous system, information provided by any failure detection mechanism has to be considered as *hints* since unbounded delays may lead to false suspicions. Despite this fact, failure detectors have proven to be powerful abstractions for classifying asynchronous systems with respect to the consensus problem [9]. If the system is partitionable in addition to being asynchronous, then failure detection has to be based on the notion of *reachability* in order to establish if a remote process is not only up but that effective communication with it is possible. We assume that the system being considered is such that it admits a *reachability detector* with weak properties that have been shown to be sufficient to solve view synchrony [6]. We further assume that despite process and communication failures, recoveries are such that pairs of processes do not remain disconnected indefinitely.

View synchrony implements the notion of a *process group* and provides *reliable multicast* as the basic communication primitive [3], [4], [5], [6]. Processes that want to participate in a common computation *join* a named group. They terminate their participation by *leaving* the group. While a member of the group, processes communicate with each other through reliable multicasts. For the multicast primitive to be terminating in an asynchronous system despite failures, view synchrony includes a membership service that provides consistent information in the form of *views* regarding the components of the group that are currently up and that can mutually communicate. View synchrony abstracts away process and communication failures, both real and due to false suspicions, by transforming reachability detector outputs into *view change* events that are collectively agreed upon.

With the events $mcast(m)$, $dlvr(m)$, and $vchg(v)$, we denote the multicast of message m, delivery of message m, and view change to v, respectively. At each process, view synchrony installs new views through $vchg(v)$ events that define a totally-ordered sequence. The last view to be installed in this sequence at a process is called the *current view* of the process. Events are said to *occur in the view* that happens to be current at the time. Views v and w are called *consecutive* if there exists some process common to both views for which w is the next view to be installed after v. View w is called a *successor* of v if there exists a sequence of views leading from v to w such that each adjacent pair of the sequence are consecutive views. It is possible for two views installed at two different processes to be incomparable with respect to the successor relation, in which case they are called *concurrent*. Concurrent views allow us to model diverging views of the group membership due to partitions.

View synchrony can be specified formally as a set of properties on view installations and message deliveries [6]. For completeness, we give such a specification in Appendix 1. The essence of view synchrony, however, can be captured informally by the following property that states how the group membership and reliable multicast services interact:

> *All processes that survive from view v into the same consecutive view w must have delivered the same set of messages in view v.*

Note that view synchrony does not place requirements on the relative order in which messages are delivered between

two consecutive views. We will assume, however, that messages multicast by the same process are delivered, if at all, in the order in which they were sent. As it turns out, message ordering guarantees stronger than this FIFO property may only help in *solving* but not *preventing* shared state problems.

3 The Application Model

An application is a distributed computation performed by a group of processes that run on top of view synchrony. Without loss of generality, we consider applications that are structured as a single group. The involvement of a process in the application begins when it joins the corresponding group and ends when it leaves the group through the view synchrony primitives *join*() and *leave*(), respectively. Each process has a local state, part of which may be permanent and survive across crashes. Including a permanent component for the local state allows us to model applications that may recover after crashes.

We consider the class of applications that implement *group objects*. According to the object-oriented paradigm, a group object is an instance of an abstract data type, encapsulating some internal state and exporting to its clients an interface defined through a set of *external operations*. Informally, semantics of an abstract data type may be defined through invariants over the internal state. Thus, the implementation of a group object for a certain type can be seen as simulating the logical internal state through a global state distributed over the group members. This in turn requires correct and coordinated interaction among the processes in the group such that invariants remain valid over the global state. How one actually determines the invariants for an abstract data type and implements the group object operations that satisfy them are beyond the scope of this paper. We assume that these tasks have already been achieved for a group object with static membership. In other words, if the group implementing the object does not experience any view changes, then the external operations transform the global state such that the invariants continue to be satisfied. What complicates the programming task is the possibility of view changes during external operations due to events such as failures, recoveries, joins, and leaves. We concentrate on this aspect of programming correct group objects. Clearly, for the group object to remain correct despite view changes during its operations, the implementation has to restore the truth of invariants over the global state whenever they are violated. To achieve this, the application relies on a set of *internal operations* that are visible only to the group object implementor and are not part of the external interface.

In order to concentrate on the problems associated with shared state management using view synchrony within a group object, we abstract greatly the application computation as follows. At any time, a process of the group object can be in one of three modes: NORMAL, REDUCED, and SETTLING (*N-mode*, *R-mode*, and *S-mode*, for short). In *N-mode*, a process performs all of the external operations defined for the object; in *R-mode*, it performs only a (possibly empty) subset of the external operations; finally, in *S-mode*, it performs internal operations only. The possible modes and transitions between them are shown in Fig. 1. An application may be structured such that only a subset of these modes or transitions are relevant.

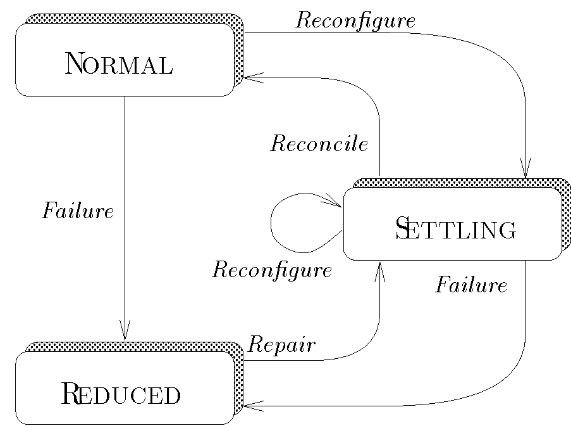


Fig. 1. Possible execution modes for a group object process. Each transition is labeled by its cause.

A transition from *N-mode* to *R-mode* is caused by any event that results in a new view that is not conducive to satisfying external operations without the risk of violating some invariant. We generically call this a *Failure* transition corresponding to a view change resulting from a process crash, communication failure, or false suspicion as discussed in Section 2. A view change that restores the conditions related to connectivity for performing all of the external operations results in a *Repair* transition. These typically correspond to recovery of crashed processes or the repair of communication failures that result in reestablishing connectivity between partitions. To return to *N-mode*, a process must first pass through *S-mode*. Given the possibility of concurrent views, while one process is in *R-mode*, others could remain in *N-mode* and continue to serve external operations, thus modifying the global state. *S-mode* models the state reconstruction that is required before a process in *R-mode* can return to *N-mode* through a *Reconcile* transition and resume serving external operations. Obviously, *Failure* transitions could occur while a process is in *S-mode*, causing it to return to *R-mode*. Finally, a process may switch to *S-mode* from *N-mode* through a *Reconfigure* transition without entering *R-mode*. This transition typically corresponds to events such as repairs or joins that cause an expansion of a process' view. *Reconfigure* transitions model the need to reconstruct the global state reflecting the new view composition before a process can resume serving external operations. For this reason, a *Reconfigure* transition may also occur while a process is already in *S-mode*. *Reconfigure* transitions from *S-mode* to *S-mode* may occur upon view changes delivered while an internal operation is being executed.

Two simple examples will serve to clarify most of the above discussion. First, consider a group object implementing a file with the two external operations *read* and *write*. For increased availability and reduced latency, the file is partially or fully replicated within the group. Informally, the correctness criteria for this object could be stated as

follows: With respect to write operations, the group object should behave exactly as if there were only one copy of the file; with respect to read operations, it is acceptable to return any available data, even though it may be stale (missing some of the more recent writes). One possible implementation of this group object is to associate with each replica of the file a vote and to define a quorum as a collection of votes that can be obtained in at most one concurrent view. For this example, it is possible to determine execution modes of processes based on the structure of the current view as follows. A process is in *N-mode* (and thus can serve both read and write operations) if the current view defines a quorum and all of the copies of the file held by processes in the view are up-to-date. If the current view is not a quorum, then all processes in the view that have a local copy of the file are in *R-mode* and they can service reads but not writes. Finally, a process is in *S-mode* if the current view defines a quorum but some processes in the view could have a copy that is not up-to-date. Processes in *S-mode* must first obtain an up-to-date copy of the file before returning to *N-mode*.

Next, consider a group object implementing a database with a single look-up query interface. For performance reasons, the database is fully replicated within the group and the query is performed in parallel by the group members, each being responsible for a portion of the database. The correctness criterion requires that look-ups against the replicated database return exactly the same results as the non-replicated case. In particular, the entire database must be searched before reporting that the value being looked up does not exist. For this example, the only external operation (look-up) can be performed in any view. Thus, *R-mode* does not exist. Any event causing a view change, however, results in a transition to *S-mode* in order to redefine the division of responsibility for portions of the database to be searched by members of the group. An inconsistency in this global state information could result either in some portion of the database being searched multiple times (reducing efficiency) or not being searched at all (compromising correctness).

We define the *history* of a process p, denoted by h_p, as a (possibly infinite) sequence of *dlvr* and *vchg* events. Let h_p^k denote an initial prefix of h_p containing the first k events. We assume that the first event of process p's history is a *vchg* event corresponding to p joining the group object. In general, the mode of a process can depend on an arbitrary number of past delivery events from the time it has joined the group. In other words, after k events, the mode of process p is defined by $\mathcal{M}(h_p^k)$, where $\mathcal{M}$ is called the *mode function*. A process determines its current mode by evaluating $\mathcal{M}$ each time view synchrony delivers it a new event. The actual mode function $\mathcal{M}$ associated with a group object depends on both the invariants of the application and on the implementation technique used to attain them. The problem of deriving $\mathcal{M}$ for a specific object group is beyond the scope of this paper. We assume that the mode function depends only on the current view composition and on the messages that are delivered during the current view. In other words, it is independent of the process' local history earlier than the last view change event. Furthermore, we assume that all processes in an object group share the same mode function.

4 The Shared State Problem

Whatever the reason for switching to *S-mode*, the activity of a process in this mode consists of checking the current global state and, if necessary, reconstructing a new one where the invariants are satisfied. We call the reconciliation that is necessary the *shared state problem*. A process makes the *Reconcile* transition into *N-mode* only upon the successful completion of the shared state problem. This transition distinguishes itself from the others since it is *synchronous* with respect to the computation. As discussed in the previous section, the other transitions in general are triggered by external events such as failures, recoveries, joins, leaves, network partitions, or partition mergers. These events, by their nature, are asynchronous with respect to the computation performed by an application. On the contrary, the *Reconcile* transition can take place only when the global state has been successfully reconstructed, which is application defined.

In the following, we shall present three conditions that give an abstract characterization for the shared state problem. These are necessary conditions and classify the shared state problem according to possible group reconfigurations resulting from failures, recoveries, joins, leaves, partitions, and repairs. Proving that a given condition actually provokes an instance of the corresponding shared state problem depends strongly on the application semantics. We believe, however, that this characterization is rather general and provides insight for building partition-aware applications.

Consider the event $vchg(v)$ delivering a new view v at process p. Moreover, let c_v be any consistent cut of the computation that includes the $vchg(v)$ events for each process p in v. When p delivers view v, it first evaluates $\mathcal{M}(v)$ to compute its next mode. Without loss of generality, we assume that the mode function evaluation is instantaneous. So, the evaluation of the new mode by all processes in v coincides with the cut c_v. Since we assume that the mode function depends only on the current view composition, all processes in v evaluate the same next mode along c_v. In other words, when a new view v is installed, every process in this view either eventually switches to the same mode or crashes.

Let us focus on the *Repair* and *Reconfigure* transitions that lead to *S-mode* and bring about the shared state problem. Consider the event $vchg(v)$ that causes a switch to *S-mode* along cut c_v. Processes in v may reach this mode through different histories: some of them might have been in *R-mode*, whereas others might have been in *N-mode* before switching to *S-mode*.[2] Therefore, we can split v in two disjoint subsets denoted $RS(v)$ and $NS(v)$ containing, respectively, those processes that were in *R-mode* and those processes that were in *N-mode* before switching to *S-mode*. Since the view synchrony model allows concurrent views, processes in $NS(v)$ could even have belonged to different views

2. We assume that a process joining the group for the first time was initially in *R-mode*.

when they were in *N-mode.*[3] Thus, we further decompose *NS*(*v*) into disjoint subsets called *clusters* such that processes in the same cluster belonged to the same view, whereas processes in different clusters belonged to different views when they were *N-mode.*

We concentrate on three incarnations of the shared state problem as described below. The common scenario for all of them is the occurrence of a *vchg*(*v*) event for which the new mode is *S-mode.*

State Transfer. This problem arises if the application is not able to tolerate processes that may join the computation at arbitrary times (which is a very common situation). We have a state transfer problem when processes that were in *R-mode* before switching into *S-mode* happen to merge together with processes that were instead in *N-mode.* Thus, a necessary condition for the state transfer problem is that neither *NS*(*v*) nor *RS*(*v*) are empty. In general, state transfer is handled by having each process in *RS*(*v*) compare its local state to the state of at least one process in *NS*(*v*) and possibly modify it as a consequence of this comparison.

State Creation. This problem arises whenever the global state must be reconstructed from scratch, for example, after a total failure scenario. A necessary condition for the state creation problem is that *NS*(*v*) is empty but *RS*(*v*) is not. State creation requires each process *p* in *v* to compare its local state with that of all other processes in *v* and possibly modify it as a result of this comparison. Identifying which local state is to be used for recreation of the others may require determining the last process to fail [10].

State Merging. This problem arises whenever processes in concurrent partitions may continue serving external operations independently. When the conditions leading to the partition are repaired, an application-specific decision has to be taken in defining a new global state that reconciles the divergence that may have taken place. The necessary condition for this situation is that *NS*(*v*) is not empty and is composed of at least two clusters.

The state merging problem does not exclude the possibility that the set *RS*(*v*) is also nonempty. In this case, the state merging and state transfer problems present themselves together. Moreover, in applications that are structured around the primary partition paradigm, state merging can never arise since primary partitions are totally ordered and, therefore, there can never be more than one cluster in *NS*(*v*).

At each view change, a process has to first determine if a shared state problem needs to be solved, and if so, which one. Occurrence of a shared state problem can be deduced locally by the mode function evaluating to *S-mode.* Classifying the problem, on the other hand, is more difficult. The only local information relevant towards classifying the shared state problem is the new view composition as provided by view synchrony. Unfortunately, this information alone is typically not sufficient for classification since views as defined by view synchrony are flat structures and do not contain information regarding *RS*(*v*), *NS*(*v*), and possible clusters.

Suppose, for example, that some process *p* makes the transition from *R-mode* to *S-mode* upon delivery of *vchg*(*v*). By reasoning on the composition of view *v*, the only conclusion *p* can draw is that *RS*(*v*) is not empty[4] but it is not able to distinguish between a state transfer or a state creation problem since it has no information about *NS*(*v*). The other aspect of this problem is that *p* is not able to determine the role that other processes in *v* will have with respect to the shared state problem. Processes can obtain this information only through additional protocols that are typically complex and costly [10].

Moreover, by their nature, *Repair* and *Reconfigure* transitions may occur asynchronously with respect to the execution of group object operations. This stems from the fact that an application has no control over the next event to be delivered by the view synchrony layer. So, an instance of a shared state problem may interrupt the execution of an external operation or overlap with another instance of the shared state problem (i.e., interrupt an internal operation). Clearly, this asynchrony is a source of significant complexity that may obscure the conceptual simplicity and elegance of view synchrony. Effectively attacking these problems depends mostly on the semantics of the application and programming skills [11].

5 Discussion

Analysis of view synchrony in terms of shared state problems allows us to better understand certain design decisions that have been made in various implementations of the abstraction. Isis, for example, provides a *state transfer tool* that permits a process joining the group to bring itself up-to-date automatically [2]. The programmer only has to define what constitutes the shared state (and thus needs to be transferred before the new process is allowed to participate actively in the computation) in terms of program variables. The actual details of the state transfer itself (e.g., from which process to obtain the state, handling view changes during transfer, etc.) are handled automatically by the system. In Isis, a state transfer is performed *before* installing a new view that includes the joining process. This is an important point since it guarantees that all processes in the current view have an up-to-date state, thus simplifying the structure of the *entire* application and not just that of the view change handlers. A consequence of this feature is the requirement for additional synchrony between the application and the external environment—a new view including the joining process cannot be delivered until the state transfer is complete, which is an application-specified action. This, in turn, requires a significantly more complex runtime support for Isis than what is needed for implementing the view synchrony model as described in this paper.

Another feature of Isis that is quite relevant with respect to shared state problems is the fact that two consecutive views of a group may expand by at most one member at a time. This seemingly minor detail has a substantial impact on the ability to reason globally with local information upon view changes. To illustrate the point, consider a process *p*

3. The same reasoning holds for processes in *RS*. However, this case is not meaningful for our discussion.

4. The set *RS*(*v*) contains at least process *p* itself.

that has just joined the group resulting in a new view *v*. Given the property just stated, *p* can immediately conclude that it is the *only* process in *RS*(*v*) and that *all* other processes in its view are in *NS*(*v*). Similarly, *p* may easily deduce whether there is an instance of the state transfer or state creation problems; the latter being the case if *p* is alone in the view. Implementation of the Isis state transfer tool has probably benefited greatly from this feature. Instead, systems such as Relacs [12], Horus [13], and Transis [5] adopt a model similar to ours, where two consecutive views may differ by an arbitrary number of members due to partitions or mergers. In these systems, global reasoning on behalf of shared state problems after view changes is much more complex.

Based on the above observations, one might argue that limiting an expanding view to include exactly one more member than its predecessor is a desirable feature for a group communication system. Unfortunately, this is highly impractical in large-scale systems. Given that such systems may be prone to frequent partitions (real or virtual) and thus frequent mergers, the restriction that views grow one process at a time will result in an inordinate number of view change events. For example, consider two partitions of *n* members each that merge after repairs. This event will result in *n* view changes in each of the two partitions, admitting one new process at a time into the view when, in fact, a single view change is all that is really required. Furthermore, the limitation in question may lead to ambiguous semantics for the reliable multicast primitive under certain failure patterns [14].

Given that Isis implements the *primary partition* (or *linear membership*) model of group communication, concurrent views are not possible. In other words, for this system, the state merger problem does not exist by definition. The price to pay for this simplification is the inability to support partition-aware applications with weak consistency requirements that could make progress in multiple concurrent partitions.

It is highly desirable for systems implementing view synchrony to include support for solving shared state problems systematically rather than having the burden fall entirely on the application programmer. It is difficult, however, to provide a generic support layer (or a suite of layers) that is appropriate for all possible application classes. For example, if the application involved very large amounts of data, as might be the case for file systems or databases, the strategy of blocking view installations while state transfer is in progress might be infeasible. In such a situation, it will be desirable to split the state into two parts: A (large) piece transferred concurrently with application activity in the new view; a (small) piece that needs to be transferred while the servicing of external operations in the new view is suspended [11]. Moreover, one might want to avoid transferring the entire state "blindly" and might prefer a solution where the two parties—the joining process and those in *NS*—negotiate parts of the shared state to transfer, depending on the context of the join event. The search for a generic support layer becomes even more difficult when we consider state merger and state creation problems in addition to state transfer.

As another example, consider the Consistent Object Replication Layer (COReL) of the Transis system [15]. Transis implements view synchrony and allows concurrent views of the same group to exist [16]. COReL simplifies the development of applications based on replicated objects by providing primitives that (eventually) totally order multicast messages within a group. Any connected subset of processes defining a quorum can make progress, even after recovery from total failures. In particular, if some process *p* enters the quorum view after having been isolated for some time, COReL relays to *p* a copy of all messages exchanged within the quorum view during *p*'s absence. By processing these messages and applying the relevant updates to its local replica in sequence, *p* brings itself up-to-date. Although this functionality is clearly of great help for the programmer, the strategy may not be practical in large-scale systems where partitions may last for a long time. In such systems, it may be preferable to send directly the relevant portions of the up-to-date state rather than buffering, relaying, and processing all update messages in sequence. Moreover, there would be no point in relaying messages that do not alter the shared state.

Further confirmation of the difficulties presented by shared state problems in the context of view synchrony can be found in the group communication primitives of the Amoeba distributed operating system [17]. Informally, Amoeba guarantees that all processes in a given group see all events concerning that group (i.e., delivery of messages and view changes) in the same order. In particular, view changes are totally ordered with respect to message deliveries, which is the property that is essential for our entire discussion. Designers of Amoeba initially believed that developing group-based reliable applications would be a fairly simple task based solely on the Amoeba primitives. They later discovered that they had underestimated the technical problems related to state creation and state transfer. It turned out that state creation and state transfer required, in practice, a toolkit library built on top of the Amoeba services. Since Amoeba provides only totally-ordered multicasts, this example also corroborates our claim that problems of shared state maintenance are essentially independent of multicast ordering issues.

To summarize, programming real applications based on view synchrony, even when augmented with "toolkits," may prove to be too difficult. Rather than approaching the problem by constructing more toolkit layers on top of view synchrony, it might be worthwhile to question the suitability of the model itself. The *enriched view synchrony* extension we propose in the next section is an attempt in this direction.

6 Enriched View Synchrony

In this section, we present a novel extension to view synchrony that is aimed at simplifying reasoning about shared state problems. This extension, called *Enriched View Synchrony (EVS)*, requires minor modifications to the view synchrony run-time support and can be implemented efficiently. Appendix 2 contains a formal description of EVS and Appendix 3 contains a sketch of how it can be implemented.

6.1 Basic Properties of Enriched View Synchrony

Our proposed extension to view synchrony is based on the notions of *subviews* and *subview sets* (*sv-sets* for short). Just like views, *subviews* are sets of process names that exist within a given view. Each view is constructed out of at least one subview. Each process belongs to exactly one subview. In other words, subviews do not overlap and they do not span across view boundaries. Subviews in the same view can be grouped together as *sv-sets*. Each subview belongs to exactly one sv-set. Within a given view, subviews and sv-sets never split and they merge only under application control, as described below. Given two consecutive views *u* and *v*, processes that are common to *u* and *v* and that were in the same subview or sv-set in *u* remain in the same subview or sv-set also after the installation of *v*. The example depicted in Fig. 2 illustrates these properties. First, a partition causes view v_1 to split into two concurrent views v_2 and v_3. When the partition is repaired, the two concurrent views merge to form a single view v_4. Note that while the partition divides the black processes of v_1 between views v_2 and v_3, within each, black processes remain together in a single subview. The merged view v_4 maintains the structure of the two previous views with respect to subviews and sv-sets. Informally, subviews permit reasoning about which processes belonged to the same view before the installation of a new view. Subview sets, on the other hand, are used by applications to mark those processes involved in some global activity at the time of a view change and that should not be interrupted by new processes entering the view.

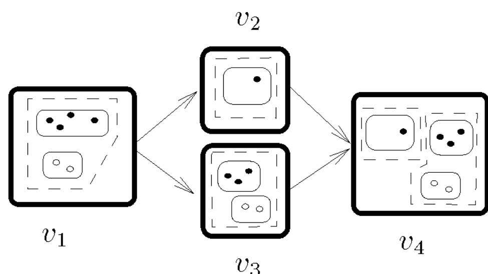


Fig. 2. Basic features of the enriched view synchrony model. Views, subviews, and sv-sets are indicated, respectively, through thick, thin, and dashed frames.

Sv-sets, subviews, and process names within a given view form a tree structure corresponding to properly nested sets: The view contains sv-sets, sv-sets contain subviews, and subviews contain process names. The case where there is a single sv-set containing a single subview containing all of the processes degenerates to the traditional view abstraction. The system attaches no meaning to subviews and sv-sets. It simply maintains the structuring information on behalf of applications.

What distinguishes subviews and sv-sets from views is the fact that their composition can grow only at the will of the application, and not at arbitrary times. For example, a process cannot simply appear in a subview after recovery or the merger of a partition. It will first have to appear in a subview by itself, and only when the application decides it may be admitted into an existing subview. As with views, failures may cause subview and sv-set compositions to shrink asynchronously with respect to the application at times of view changes. In Fig. 2, the partition causes the subview of black processes to shrink in each of views v_2 and v_3 with respect to v_1. After the merge, however, processes that were in different subviews or sv-sets in v_2 and v_3 continue to belong to different subviews or sv-sets also in v_4. This is because subviews and sv-sets may merge only in response to application-invoked primitives as described below. It is this aspect of EVS, where subviews and sv-sets expand synchronously with respect to the application, that distinguishes it from traditional view synchrony.

Our extended view synchrony service delivers processes, messages, and *enriched views* (*e-views* for short) that include the sv-set and subview structure within the view. Traditional view changes correspond to e-view changes where there is a change in the set of processes making up the view. Even when the view membership remains unaltered, e-view change events may be provoked by applications requesting mergers of subviews or sv-sets. When a process first joins a group, it appears within the new view in a new sv-set containing a new subview containing only the process itself. After their initial creation, subviews and sv-sets may be modified by the application through the following calls, which augment the usual view synchrony interface:

SV-SetMerge(*sv-set-list*). Create a new sv-set that is the union of the sv-sets given in *sv-set-list*. Any sv-set in *sv-set-list* that does not belong to the current view is ignored.

SubviewMerge(*sv-list*). Create a new subview that is the union of the subviews given in *sv-list*. The resulting subview belongs to the sv-set of the invoking process. Any subview in *sv-list* that does not belong to the sv-set of the invoking process is ignored.

Fig. 3 illustrates a sequence of e-view changes provoked by the above calls. Dashed arrows indicate e-view changes that are not view changes (i.e., the composition of the view as a set of processes remains unchanged). The first e-view change is due to an *SV-SetMerge()* call merging three sv-sets, each containing a single subview consisting of white processes. The second e-view change is due to a *Subview-Merge()* call merging the top two subviews of the newly created sv-set. Note that the example depicts a scenario where no failures occur; thus, the composition of view *v* remains unchanged; only the structure of subviews and sv-sets within the view change in response to application invoked calls.

This extended service maintains the semantics of view synchrony regarding view changes and message deliveries,

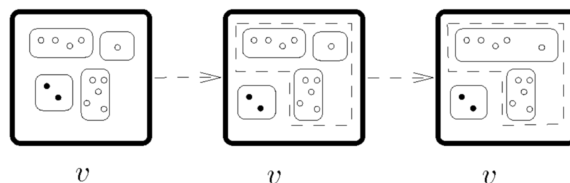


Fig. 3. The figure illustrates two e-view changes within a single view *v*. For simplicity, sv-sets that contain a single subview are not traced out as dashed frames.

exactly as described in Appendix 1. With respect to e-view changes, the following additional properties are guaranteed, which we state informally:

PROPERTY 6.1 (Total Order). *E-view change events within a given view (i.e., between two consecutive view change events) are totally ordered by all processes in the view.*

PROPERTY 6.2 (Causal Order). *E-view change events define consistent cuts of the computation. In other words, causality relations between message multicasts and e-view changes are preserved.*

PROPERTY 6.3 (Structure). *Subview and sv-set structures are preserved across view changes. In other words, processes that belong to the same subview (sv-set) in a given view remain in the same subview (sv-set) also in the successor view. Moreover, processes that do not belong to the same subview (sv-set) in a given view remain in different subviews (sv-sets) also in the successor view.*

6.2 Structuring Applications Based on Enriched View Synchrony

Our proposed extension to view synchrony presents an opportunity for systematic and simplified solutions to shared state problems. It enhances the global reasoning that can be achieved based on local information after view changes and simplifies handling of the asynchrony between view synchrony run-time support and the application.

In terms of the application model used in this paper, we structure an application according to the following methodology:

1) External operations are performed within a single subview and not across different subviews.
2) Internal operations are performed across subviews belonging to the same sv-set. Upon successful completion of the internal operation, all subviews within this sv-set are merged into a single one.

It follows that the existence of multiple sv-sets within a view signals the necessity for solving a certain instance of the shared state problem. Moreover, the existence of multiple subviews within a given sv-set signals that a shared state problem instance is in progress within this sv-set.

This methodology is illustrated in Fig. 4. Initially, some process in v_1 creates an sv-set containing all three subviews, signaling that some internal operation is in progress. The resulting e-view change is indicated with the dashed arrow between two instances of view v_1. After the partition merges and view v_3 is installed, the black processes that were in v_2 can conclude, based solely on local information, that all of the white processes were together in a partition (v_1) and were engaged in an internal operation before the merge, and thus should not be disturbed.

This methodology greatly simplifies reasoning about shared state problems using only information that is locally available to processes. Note that processes entering an expanding view are not permitted to participate in the computation that might be in progress at the time of the view change because they will appear in a different subview (or sv-set) than the one carrying out external or internal operations. Rather, they have to be "let in" explicitly by the other

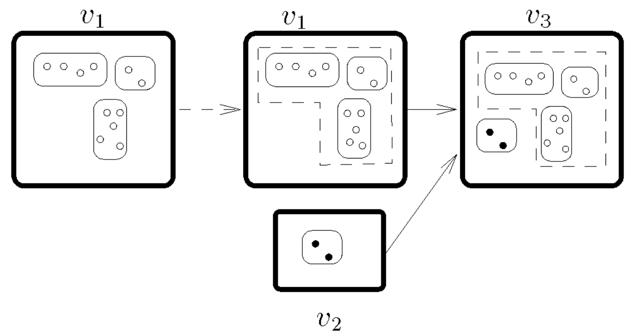


Fig. 4. Example of structuring of an application based on enriched view synchrony.

members in order that the appropriate subviews and sv-sets expand. The Structure Property guarantees that processes remain in the relevant subview (sv-set) across view changes, thus, all surviving processes will continue to participate in the computation and have the same notion of shared state.

As an example, consider the file object introduced in Section 3, and suppose that the implementation of the external operations involves the management of a mutually-exclusive write lock within a quorum view. The shared global state will thus include the identities of the lock manager and the current lock holder (if any). Suppose some process *p* installs a view *v* consecutive to *u* such that *v* defines a quorum whereas *u* does not (i.e., *p* switches from *R-mode* to *S-mode*). In traditional view synchrony, upon installing view *v*, the only conclusion *p* can draw based only on local information is the fact that *v* indeed defines a quorum. It cannot distinguish between the following scenarios:

1) A quorum already existed in one of the views prior to *v* (i.e., a state transfer problem exists since *NS* and *RS* are both nonempty);
2) The shared state was being reconstructed at the time *v* was installed (i.e., a creation problem exists since *NS* is empty but *RS* is not and an instance of the related internal operation has been interrupted);
3) A quorum is reborn after it had disappeared temporarily (i.e., a creation problem exists since *NS* is empty and *RS* is not).

With our proposed extensions, process *p* can draw several relevant conclusions through local reasoning on the view composition and structure. If the new view *v* contains a subview that defines a quorum, such a subview constitutes the set *NS* and thus contains processes whose notion of shared state is up-to-date. Notice that this is a major advantage since *v* may contain processes other than *p* that have just joined *v* and thus do not know how to obtain an up-to-date shared state. If, on the contrary, *v* does not contain any subview that by itself defines a quorum, then cases 2 and 3 can be distinguished by controlling if *v* contains an sv-set defining a quorum.

As for the asynchrony between application and run-time support, note that while an operation is being executed, the set of processes participating in it may only shrink—a new view may be delivered by view synchrony at arbitrary

times but the composition of subviews and sv-sets may grow only at the will of the application. Therefore, algorithms can be easily designed to run undisturbed across view changes. For instance, in case 2 above, process *p* can decide locally to wait for the processes running the creation protocol to complete their task before disturbing them for a copy.

As a further example, suppose that the availability of the replicated object file is further increased by allowing writes in any view. Informally, a write that takes place in a view defining a quorum has a permanent effect. The effects of a write performed in a non-quorum view are tentatively accepted but remain pending. They will become permanent if no write occurred in a concurrent view. Otherwise, they will be discarded. One can read up-to-date and permanent values only in a view defining a quorum.

One possible implementation of this scheme consists of letting a tentative write create a tentative copy, and associating with each tentative copy information reflecting the partial ordering among writes. Version vectors are an example of this information [18]. A tentative write becomes permanent by promoting the tentative copies to plain copies and disseminating them to a quorum of processes. A tentative write is rolled back by deleting the tentative copies it produced. When two or more views merge to form a single view, all version vectors in the resulting view are compared in order to detect concurrent writes. Any tentative write that is discovered to have been concurrent with respect to another write is rolled back. Then, the most recent write, if any, is propagated to all members of the view (and the version vectors updated accordingly). If the resulting view defines a quorum, this write is also made permanent.

In this example, the external operations *read* and *write* can be performed in any view, and it follows that processes in a view composed of a single subview are in *N-mode* while *R-mode* does not exist. Moreover, any view change that notifies the merging of two or more views produces a transition from *N-mode* to *S-mode*. Processes in *S-mode* compare their version vectors and propagate their copies, if necessary, before returning to *N-mode*. Since the only transitions that can occur are *Reconfigure* and *Reconcile*, processes have to confront only the state merging problem.

Suppose that a process *p* installs a view *v* consecutive to two or more views. As stated earlier, with traditional view synchrony, the only conclusion that process *p* can draw based on local information is whether *v* defines a quorum or not. With EVS, instead, *p* may also determine the grouping of *v* in clusters (e.g., subviews) and the clusters that are already involved in a state merging (e.g., sv-sets composed of multiple subviews).

7 Programming Example

In this section, we present details of the file object implementation described in Section 3 and in the previous section. The exercise is useful in that it will illustrate the programming methodology we introduced in previous Section 6.2.

7.1 Overview

Let a *quorum sv-set* and a *quorum subview* be, respectively, an sv-set and subview that include enough processes to define a quorum. Based on these definitions and the replicated file specification, the mode of a process can be determined as follows. A process is in

1) *N-mode* iff it belongs to a quorum view composed of only one subview;
2) *S-mode* iff it belongs to a quorum view composed of multiple subviews; and
3) *R-mode* iff it does not belong to the quorum view.

Furthermore, the shared state problems that may occur are the creation problem (when a quorum subview disappears for some time) and the state transfer problem (when one or more subviews appear in a view together with a quorum subview).

According to the programming methodology, external operations are executed within subviews. In particular, writes can be executed only in a quorum view composed of only one subview, whereas reads can be executed in any subview, thus returning possibly stale data. Reads are guaranteed to return the current contents of the file only if they are executed in the quorum subview. Internal operations for solving both creation and state transfer problem and reestablishing the consistency of the file contents after the joining of some processes are carried out within a quorum sv-set.

Developing the application based on our methodology requires the implementation of five components:

1) external operations;
2) internal operations;
3) computing the process mode;
4) detecting the shared state problem instances; and
5) deciding whether an internal operation has been interrupted.

We achieve this by splitting up each process into two components: the *low-level event manager* (*LLEM*) and the *high-level event manager* (*HLEM*). *LLEM* implements items 3–5 by analyzing e-views delivered to the process by the EVS run-time support. *LLEM* then passes the e-view event, enriched by the outcome of this analysis, to *HLEM*, that implements 1 and 2.

LLEM collects the results of its analysis into a data structure called *Analysis* that is a triple (*mode*, *problem*, *phase*) of enumerated types. The *mode* variable contains the current mode of the process and it may be any one of *N-mode*, *R-mode*, or *S-mode*. The *problem* variable describes which shared state problem needs to be solved and it may be any one of StateTransfer, Creation, or None. Finally, the *phase* variable describes if the process is involved in an internal operation or if an internal operation is necessary but has not started yet. The value InProgress specifies the former case, whereas the value Recorded the latter. If no shared state problem needs to be solved, then both *problem* and *phase* are set to None. A crucial point to observe is that *LLEM* constructs *Analysis* on the basis of local reasoning only. It is straightforward to deduce from our methodology that the *Analysis* produced by *LLEM* when analyzing a given *e-view* is identical at all processes that belong to the same subview in that *e-view*.

HLEM starts an internal operation when it receives an e-view event from *LLEM* augmented by a triple whose fields *mode* and *phase* are equal to *S-mode* and Recorded, respec-

tively (this point will be clarified further below). The value of field *problem* determines which internal operation has to be executed. In general, upon delivery of an e-view event, *HLEM* forwards this event to the in-progress operations and then, if necessary, starts an internal operation.

Internal operations begin by creating an sv-set that includes the relevant processes. Upon delivery of the corresponding e-view, the *phase* field of *Analysis* switches to INPROGRESS. Internal operations can proceed across view changes as long as the field *phase* continues to be INPROGRESS, that is, as long as the composition of the sv-set continues to define a quorum. Otherwise, the operation aborts.

Our algorithms are expressed in a simple pseudo programming language that supports multi-threaded processes. Indentation levels implicitly delimit blocks. The statement **wait-for**(*condition*) synchronizes a thread with the delivery of an event that renders the specified condition true. Upon delivery of an event, the thread executes an uninterruptible code segment called a *handler* that is specified through an **upon**(*event*) statement. Within a handler, we use the notation "Abort **wait-for**" as a shorthand for the forcible termination of the procedure containing the **wait-for**(*condition*) statement that synchronized the executing thread with the current event.

7.2 Implementation

Each process is composed of two initial threads, corresponding to *LLEM* and *HLEM*. In the following, we give the details for *LLEM* and the internal operations carried out by *HLEM* towards solving the state transfer and state creation problems. As will become clear later, these algorithms are quite general and are applicable to a large class of applications following the quorum model. *HLEM* starts an internal operation by spawning a new thread. For the sake of brevity, we omit the pseudo-code for external operations performed by *HLEM* since it does not contribute to this discussion. For the same reason, we omit details concerning interthread communication.

Several ancillary functions are defined. Function *SetOfSV-Set*() takes an e-view as argument and returns the set of sv-sets contained in that e-view. Function *SetOfSV*() takes either an e-view or an sv-set and returns the set of subviews contained in its argument. Function *comp*() takes either an e-view, an sv-set, or a subview as argument and returns the set of processes contained in its argument. Function *quorum*() also takes an e-view, an sv-set, or a subview as argument and returns the Boolean value TRUE iff the corresponding set of processes defines a quorum. *MySV*, *MySV-Set*, and *MyPid* denote the current subview, the sv-set, and the name of the invoking process. Finally, function *elect*() returns a process chosen deterministically from the set specified as its argument.

The pseudocode for *LLEM* is given in Fig. 5. Let *p* denote the executing process. The cases in which *p* is either *R-mode* or *N-mode* are straightforward (lines 4-8). If *p* is *S-mode*, its reasoning depends primarily on whether it belongs to a quorum subview (lines 11-16) or not (lines 17-26). In the former case, *p* reasons on the set of processes that belong to its sv-set but not to its subview (variable *in*, line 12). If this set is not empty, *p*'s sv-set contains multiple subviews. It follows that *p* is participating in the execution of a state transfer (line 14). Otherwise, the need for a state transfer is recorded (line 16). When *p* is not in a quorum subview, instead, it first determines whether there is a quorum sv-set (line 18). If there is no quorum sv-set, then a creation algorithm shall be started (line 26). Otherwise, *p*'s reasoning depends on whether it belongs to the quorum sv-set or not. The former implies that *p* is participating in the execution of an internal operation (lines 19 and 21-24). The latter implies that an internal operation is being executed but *p* is not participating in it (lines 19-20).

```
1  procedure LLEM()
2
3  upon vchg(ev)
4     if (not quorum(ev)) then
5        Analysis := (R-mode, NONE, NONE);
6     else
7        if (comp(MySV) = comp(ev)) then
8           Analysis := (N-mode, NONE, NONE);
9        else
10          % S-mode
11          if (quorum(MySV)) then
12             in := {p | p ∈ comp(MySV-Set) ∧ p ≠ comp(MySV)};
13             if (in ≠ ∅) then
14                Analysis := (S-mode, STATETRANSFER, INPROGRESS);
15             else
16                Analysis := (S-mode, STATETRANSFER, RECORDED);
17          else
18             if (∃ss ∈ SetOfSV-Set(ev) | quorum(ss)) then
19                if (MySV-Set ≠ ss) then
20                   Analysis := (S-mode, STATETRANSFER, RECORDED);
21                else
22                   if (∃sv ∈ SetOfSV(ev) | quorum(sv)) then
23                         Analysis := (S-mode, STATETRANSFER,
                                        INPROGRESS);
24                   else Analysis := (S-mode, CREATION, INPROGRESS);
25             else
26                Analysis := (S-mode, CREATION, RECORDED);
27    pass vchg(ev) event up to HLEM;
```

Fig. 5. Structure of low-level event management.

The pseudocode for the part of *HLEM* that implements creation is given in Fig. 6. *HLEM* spawns a thread for executing procedure Creation() upon receiving the triple (*S-mode*, CREATION, RECORDED) from *LLEM*. In summary, state creation is performed as follows. Processes in the quorum view elect a coordinator that:

1) creates an sv-set encompassing the entire view;
2) collects local states from all processes in the sv-set;
3) decides on a new state and multicasts it within the sv-set;
4) merges the entire sv-set into a single (quorum) subview.

The correspondence between these steps and the pseudocode in Fig. 6 is straightforward. In particular, note that the primitives for subview and sv-set merging are invoked by the coordinator (lines 4-5 and 11-16). Changes in the view composition during execution of the algorithm are handled simply (lines 22-29). In particular, the algorithm is aborted only if the relevant sv-set does not constitute a quorum any more or if the coordinator leaves the quorum view before creating the sv-set (lines 24-25). In the latter case, another instance of the creation algorithm will be spawned by

```
1  procedure Creation()
2     s := comp(ev);
3     coord := elect(s);
4     if (MyPid = coord) then
5        SV-SetMerge(SetOfSV-Set(ev));
6     wait-for (vchg(ev) | Analysis.phase = InProgress);
7     core-Creation();
8
9  procedure core-Creation()
10    Transfer local state to coord;
11    if (MyPid = coord) then
12       wait-for (receipt of local state from all in s);
13       Select new state among received local states;
14       Transfer new state to all processes in s;
15       wait-for (ack from every process ∈ s, except for myself);
16       SubviewMerge(SetOfSV(MySV-Set));
17    else
18       wait-for (new state from coord);
19       Send ack to coord;
20    wait-for (vchg(ev) | comp(MySV) = comp(MySV-Set));
21
22 upon vchg(ev)
23    if (Analysis.mode ≠ N-mode) then
24       if (Analysis.mode = R-mode or
                        Analysis.phase ≠ InProgress) then
25          Abort thread;
26       s := s ∩ comp(ev);
27       if (coord ∉ s) then
28          coord := elect(s);
29          Abort wait-for and call core-Creation();
```

Fig. 6. State creation algorithm.

```
1  procedure State-Transfer-Active()
2     out := {sv| sv ∈ SetOfSV(ev) ∧ sv ∉ SetOfSV(MySV-Set)};
3     coord := elect(comp(MySV));
4     if (MyPid = coord) then
5        SV-SetMerge(MySV, out);
6     wait-for (vchg(ev) | Analysis.phase = InProgress);
7     core-Active();
8
9  procedure core-Active()
10    if (MyPid = coord) then
11       Transfer state to all processes in out;
12       wait-for (ack from every process in out);
13       SubviewMerge(SetOfSV(MySV-Set));
14    wait-for (vchg(ev) | comp(MySV) = comp(MySV-Set));
15
16 upon vchg(ev)
17    if (Analysis.mode ≠ N-mode) then
18       if (Analysis.mode = R-mode or
                        Analysis.phase ≠ InProgress) then
19          abort thread;
20       if (Analysis.problem = StateTransfer) then
21          out := out ∩ SetOfSV(MySV-Set);
22          if (coord ∉ comp(MySV)) then
23             coord := elect(comp(MySV));
24             abort wait-for and call core-Active()
25       else abort wait-for and call Creation();
26
27 procedure State-Transfer-Passive()
28    wait-for (vchg(ev) | Analysis.phase = InProgress);
29    Receive state;
30    Send ack;
31    wait-for (vchg(ev) | Analysis.mode = N-mode);
32
33 upon vchg(ev)
34    if (Analysis.mode ≠ N-mode) then
35       if (Analysis.mode = R-mode or
                        Analysis.phase ≠ InProgress) then
36          abort thread;
37       if (Analysis.problem ≠ StateTransfer) then
38          abort wait-for and call Creation();
```

Fig. 7. State transfer algorithm. The upper part is executed by processes in the quorum subview, the lower part by processes not in the quorum subview.

HLEM. If, instead, the coordinator leaves the quorum view after creating the sv-set, it is taken over by another process (lines 26-29). It can be shown that if the number of view changes is finite and the view continues to define a quorum, then the quorum subview will eventually be created.

We make the following observations. Let *ev* and *ev*′ be the e-views corresponding, respectively, to the formation of the sv-set and its merging into a single subview. The Causal Property of EVS guarantees that a process in the quorum subview will not be delivered a message pertinent to external operations before *vchg*(*ev*′). Similarly, for instance, the coordinator will not be delivered local states before the delivery of *vchg*(*ev*).

The pseudocode for the part of *HLEM* that implements state transfer is given in Fig. 7. This operation is started upon receiving the triple (*S-mode*, STATETRANSFER, RECORDED) from *LLEM*. In particular, processes in the quorum subview execute procedure *State-Transfer-Active*() whereas the others execute *State-Transfer-Passive*(). Processes in the quorum subview elect a coordinator that:

1) creates an sv-set encompassing the entire view;
2) transfers state to processes that are in the sv-set but not in the quorum subview;
3) merges the sv-set into a single subview.

Processes that are not in the quorum subview simply wait for the up-to-date state and for their admission in the quorum subview. Observe that processes joining the quorum view while an internal operation is in progress (either creation or state transfer) will simply wait for the up-to-date state and for their admission in the quorum subview (line 28 of Fig. 7 and lines 18-20 of Fig. 5).

The algorithm exhibits many similarities with the creation algorithm. Primitives for subviews and sv-set merging are invoked by the coordinator (lines 4-5 and 10-13); state transfer is aborted only if the coordinator leaves before enlarging the quorum sv-set or if the quorum sv-set disappears (lines 18-19 and 35-36)[5]; and the coordinator's leaving of the quorum view is managed by electing a new one (lines 22-24). Let *p* be a process executing the state transfer algorithm from outside the quorum subview. It can be shown that if the number of view changes is finite, then *p* will eventually belong to the quorum subview or its view will not define a quorum. Moreover, let *ev* denote the e-view corresponding to the end of state transfer. The delivery of *ev* lets processes in the quorum view switch to *N-mode* and thus resume servicing external operations. The Causal Property of EVS guarantees that no messages related to new external operations may be delivered before *ev*.

So far we have assumed that a process' local state may be sent to others as a single multicast message. This may not be realistic if the local state is very large, for instance, containing an entire file system volume or a database. In such cases, the transfer of the new state could require a long sequence of messages. Moreover, it might be useful if state

5. Lines 25 and 37-38 handle the case in which the quorum subview disappears but the sv-set in which state transfer was being executed still defines a quorum.

transfer were preceded by a phase in which the two parties negotiated the part of the state that actually needs to be transferred. We do not show the required changes to the algorithm for sake of brevity. We do observe, however, that these aspects make the possibility of running algorithms across view changes even more valuable, which is facilitated by EVS.

We have stated above that *HLEM* starts a state transfer when it receives the triple (*S-mode*, STATETRANSFER, RECORDED) from *LLEM*. We discuss this point in more detail in the sequel. Consider a process that joins the quorum view while state transfer is in progress and remains in that view until completion. Processes in the quorum subview will produce, at the end of state transfer, a triple (*S-mode*, STATETRANSFER, RECORDED), which causes *HLEM* to start a further instance of this internal operation. It is easy to see that an inopportune sequence of failures and repairs may indefinitely prevent the resuming of external operations. In practice, *HLEM* may be structured so that, after a predefined number of "consecutive" state transfers, processes in the quorum subview resume servicing external operations "for a while."[6]

Moreover, let the current value of *Analysis* be (*N-mode*, NONE, NONE) and let its value switch to (*S-mode*, STATETRANSFER, RECORDED) as a result of a view expansion notified by event *vchg*(*ev*). Upon delivery of such an event, an external operation might have been in progress. What actually happens in this case is application dependent and depends on the policy encoded in *HLEM*. A reasonable policy might consist of servicing the state transfer immediately, after aborting in-progress external operations. The approach of aborting in-progress operations is also commonly adopted [19], [20] probably because it is by far the simplest to implement in programming paradigms other than EVS. We believe that such a policy is motivated more by the lack of expressiveness of the programming model than by real application needs. Moreover, it may result in poor performance particularly in large-scale systems where communication between remote sites may be very unreliable.

A different policy consists of completing in-progress external operations and then servicing the state transfer. We have (implicitly) structured our example this way, which is easy in EVS. Moreover, our proposed methodology allows implementing widely differing policies. For instance, one could easily implement *overlapping* state transfers with the servicing of external operations, which is highly desirable when the state to be transferred is very large [11]. After creating the sv-set including the quorum subview and the just-joined processes, state transfer would occur in two phases. During the first phase, processes in the quorum subview keep on servicing external operations, and in background, transfer the large amount of state to the others. Then, in the second phase, the servicing of external operations is suspended while a small amount of state is transferred (for propagating the updates that might have occurred meanwhile) and, finally, the new quorum subview is created.

6. In this case, one shall assume that a process in the quorum subview may execute external operations also when *LLEM* delivers a triple with the *mode* field set to *S-mode*.

```
1  procedure LLEM()
2
3  upon vchg(ev)
4    if (comp(MySV) = comp(ev)) then
5      Analysis := (N-mode, NONE, NONE);
6    else
7      if (comp(MySV) ≠ comp(MySV-Set)) then
8        Analysis := (S-mode, STATEMERGING, INPROGRESS);
9      else
10       Analysis := (S-mode, STATEMERGING, RECORDED);
11   pass vchg(ev) event up to HLEM;
12
13 procedure Merging()
14   s := {p | p ∈ comp(ev) ∧ comp(SV(p)) = comp(SV-Set(p))};
15   coord := elect(s);
16   if (MyPid = coord) then
17     s' := {sv | p ∈ s ∧ sv = SV(p)};
18     SV-SetMerge(s');
19   wait-for (vchg(ev) | Analysis.phase = INPROGRESS);
20   core-Merging();
```

Fig. 8. Example of low-level event management (top part) and state merging (bottom part) when processes in concurrent views may be *N-mode*. *SV*(*p*) and *SV-Set*(*p*) denote, respectively, the subview and sv-set of process *p*. The remaining parts of the merging algorithm (procedure core-Merging() and handler of *vchg*(*ev*)) are identical to those of Fig. 6.

To complete the discussion, we present an example in which *concurrent* views may be in *N-mode*. In particular, we shall assume that external operations may be executed in any view composed of a single subview. It follows from this assumption that *R-mode* mode does not exist and that merging is the only internal operation required.

The pseudocode for *LLEM* is given in Fig. 8 (top) and is self-explanatory.[7] Portion of *HLEM* that implements merging may be obtained, for instance, by modifying the creation code in Fig. 6 as shown in the bottom part of Fig. 8. The only essential difference is the initialization of variable *s*, that contains the set of processes participating in the execution of the internal operation. This variable is now initialized to contain all processes in the current view that are not already executing the operation. Observations that were made for the creation algorithm can be easily reformulated for this example.

It is important to point out that merging operations may be executed in *parallel*, that is, there may be multiple merging operations in progress within the same view. Consider, for example, the delivery of a view containing three sv-sets *ss*1, *ss*2, *ss*3, where *ss*1 is composed of multiple subviews while *ss*2 and *ss*3 are composed of a single subview each. Processes in *ss*2 and *ss*3 start an internal operation among themselves without waiting for the completion of the state merging that is in progress within *ss*1. The subview resulting from the merging of *ss*2 and *ss*3 will participate in a further instance of the merging algorithm with the subview resulting from the merging in *ss*1. This scenario cannot happen in the creation algorithm because creation requires a quorum of participants.

In traditional view synchrony, programming parallel state merging operations within the same view is very complex. In contrast, it is practically straightforward in

7. The *mode* variable of *Analysis* is now allowed to assume the value STATEMERGING.

EVS: The set of processes that participates in the algorithm is built based on simple local reasoning and, once built, it never expands even in the presence of view changes. Apart from the potential performance improvements (for instance, when the state to be compared could require a long sequence of messages), what needs to be emphasized is the significant simplification of the programming task.

8 Related Work

View synchrony was originally proposed for local-area distributed systems where partitions were considered unlikely [4]. In this so-called *primary-partition* model, only a single view of a group may be active. While this model is limited in its expressiveness, the lack of concurrent views simplifies the implementation issues. For example, a process that is about to leave the view may be delivered messages in an order that is inconsistent with respect to other processes that remain in the view. This apparent inconsistency is not a problem since each process that leaves the primary view is declared as having crashed and it may reenter the view only after recovery as a new process.

The original primary-partition model has later been extended to permit multiple views to exist concurrently [3], [5], [21], [16], [6]. However, none of the proposed extensions to view synchrony explicitly address the problems related to shared state maintenance, which are instead central to the utility of the abstraction. What distinguishes our work from the numerous other proposals is the fact that EVS permits a methodology for solving the problem over a large class of applications. This is in contrast with solutions that are customized for particular problem domains or environments.

Cristian considers a group of replicated servers that export updates and queries as external operations [22]. This work introduces three different specifications of replica consistency. Very informally, the first of these specifications allows updates to occur in concurrent views and requires the execution of proper reconciliation procedures when these views merge. These procedures, which resemble our internal operation for state merging, are assumed to be application-dependent and are not detailed. The second specification allows updates to occur only in a majority view. The third specification requires that each update involve all members of the group. Each update is propagated with a single atomic broadcast, and the cited paper presents atomic broadcast protocols that implement each of the three specifications.

The protocols of Cristian are such that processes in a view do not start sending broadcasts until the local state of each replica consistently reflects the history of updates performed in the group. This property of agreement on initial view state is achieved by means of a dedicated phase of the protocol: upon a view change, a process in the view collects the state of the other processes, decides on the new state and broadcasts its decision to the others. In other words, in our terminology, internal operations are "hard-wired" into the atomic broadcast protocol. In contrast, we build internal operations on top of the communication primitives and we provide a paradigm where reasoning about the low-level details of internal operations is simplified. We claim that our choice is more modular. Moreover, it removes complexity from the lowest layers of the system and, thus, it appears to be justified by an end-to-end argument [23]. As an aside, the fact that the protocol for atomic broadcast includes a dedicated phase for reconciling the local states, supports our claim that shared state problems are independent of multicast ordering issues and, in a sense, they are more fundamental.

The work of Cristian shares with us the idea that consistent membership information and total ordering of message deliveries with respect to membership changes are not sufficient for reasoning about agreement on initial view state. In particular, the membership service provides additional information (e.g., the composition of the previous majority view or the composition of the respective previous view) about the previous view of *each* process in a newly-installed view. In EVS, we explore a similar idea but in a different way. An enriched view that notifies a view change provides information about the respective previous views. For instance, processes in the same subview (sv-set) come from the same view. However, this information is presented in a form that is closer to the actual needs of applications and it may be controlled by applications themselves.

El Abbadi et al. present a replica control protocol for a replicated database using the notion of views [24]. Each site can independently decide which sites to include and when in its current view. In particular, it is the application that decides when a view shall expand. The correctness of the database does not depend on this choice, but data availability and operation costs may be affected. The notion of view is not provided by the communication layer but it is completely implemented by the database management system. When a site decides to change its current view, the replica control protocol invokes the system transaction "update" that contacts all replicas in the view to bring them up-to-date. This approach removes all problems related to operations interrupted by an asynchronous view expansion, at the expense of some additional complexity in the application.

A further example is reported by Howard and Katz and proposes a "reconcile" language construct for creating a consistent state among a set of processes that access only the respective local variables [25]. However, this work is more concerned with the semantics and use of the proposed construct than with how to actually support it. Our work is placed at a lower level of abstraction and it is cast in terms of view synchrony and partitionable membership. A similar comparison can be made with respect to the work of Evangelist et al. where it is argued that multicast is not an appropriate high-level language construct for synchronizing multiple processes [26]. Even in this case, the level of abstraction is quite different from ours. This work takes into account the possibility of process failures and allows a so-called "multiparty synchronization" to complete with only a quorum of the processes involved. However, process failures are assumed to be reliably detected and failed processes never reenter the computation.

9 Conclusions

Shared state problems such as state transfer, creation, and merging, are likely to be an issue in many future applications that have reliability constraints. This is particularly true in partitionable systems such as the Internet, where multiple views of a group may exist concurrently. Even though view synchrony has the potential for being a clean and elegant programming abstraction, this elegance can easily be lost in practice, unless special provisions are made for supporting shared state maintenance.

We have given a characterization for shared state problems in terms of necessary conditions and presented an analysis of related problems that arise in practical applications. We have then presented an extension to view synchrony, called *enriched view synchrony*, explicitly conceived to simplify the task of shared state maintenance in partition-aware applications. Group views delivered to processes are enriched by structural and historical information relevant to the group's activity. Such information is defined by the application and maintained by the run-time support.

In conjunction with a simple programming methodology, local reasoning that is possible upon view changes is greatly improved with enriched view synchrony, even in the case of expanding views. Among others, a process is able to infer whether an algorithm for shared state maintenance shall be run or if it was already in progress before the view change. In the former case, a process can also infer the type of shared state problem and which other processes need to be involved. Moreover, asynchrony between run-time support and application may now be controlled in the sense that shared state problems cannot occur at instants that are inopportune for the application. This in turn simplifies the entire application and not just those parts responsible for handling events that trigger shared state problems. The methodology has been illustrated through a simple example that can be extended to a large class of applications based on the quorum model.

Appendix

A1 View Synchrony

For completeness, we give a specification for view synchrony in terms of properties on view installations and message deliveries. A more detailed discussion along with considerations for the implementability of the specification can be found elsewhere [6].

Property A.1 (View Order). *Two processes that install the same two views install them in the same order.*

Property A.2 (View Agreement). *If a correct process p installs view v, then for every process q included in v, either*

1) *q also installs v, or*
2) *p eventually installs consecutive view w that excludes q.*

Property A.3 (View Integrity). *Every view installed by a process includes itself.*

Property A.4 (View Nontriviality). *If process p is continuously able (unable) to communicate with some other correct process q, then the current view of p will eventually include (exclude) q.*

Property A.5 (View Intersection). *Concurrent views have no common members in their intersection.*

Property A.6 (Message Agreement). *If a correct process p delivers message m in view v, then for every process q included in v either*

1) *q also delivers m, or*
2) *p eventually installs consecutive view w that excludes q.*

Property A.7 (Message Uniqueness). *Each multicast message, if delivered at all, is delivered in exactly one view.*

Property A.8 (Message Integrity). *Each process delivers a message at most once and only if some process actually multicast it earlier.*

Property A.9 (Message Termination). *A correct process always delivers its own multicast messages.*

A2 Formal Specification of Enriched View Synchrony

A2.1 Definitions and Notation

Recall that e-view changes may be provoked either by failures and recoveries, or by the application through the primitives *SV-SetMerge*() and *SubviewMerge*() as described in Section 6.1. Let event *prvk*() denote the invocation of one of these primitives. As with normal view changes, e-view changes are delivered through *vchg*() events. We model the execution of a process through its *local history* that is a sequence of events occurring at the process. Let h_p denote the local history of process *p*. We extend the *consecutive* relation, originally defined between views, to e-views. An e-view is described by its name, the set of processes included in the view and their structuring in terms of subviews and sv-sets. With *comp*(*ev*) we denote the composition of e-view *ev* as the set of processes included in the view. We maintain the traditional notion of "view change" as follows. Given two consecutive e-views *ev* and *ev*′, the event *vchg*(*ev*′) corresponds to a view change if $comp(ev') \neq comp(ev)$. View changes satisfy the properties of view synchrony as described in Appendix 1.

Let $ev_p^i(v)$ denote the *i*th e-view to be delivered at process *p* in a given view *v*. By definition, $ev_p^0(v)$ corresponds to the view change installing view *v* and $comp(ev_p^i(v)) = comp(v)$ for all $i = 0, 1, \ldots, \ell$ until the next view change. In other words, the superscript denotes the position of an e-view delivered at a process relative to the last view change (see Fig. 9). Let $ev_p^\ell(v)$ denote the last e-view to be delivered at process *p* in view *v*. Let ev_p^i denote the *i*th e-view to be delivered at process *p* since the beginning of the computation, irrespective of the view. Finally, the predicate SameSubview(*p*, *q*, *ev*) is defined to be true if processes *p* and *q* belong to the same subview in e-view *ev*.

A2.2 Properties of Enriched View Synchrony

We now give a formal specification for enriched view synchrony that was described informally in Section 6.1. The specification is given as a set of properties on e-views, message deliveries, and related system events. It is intended

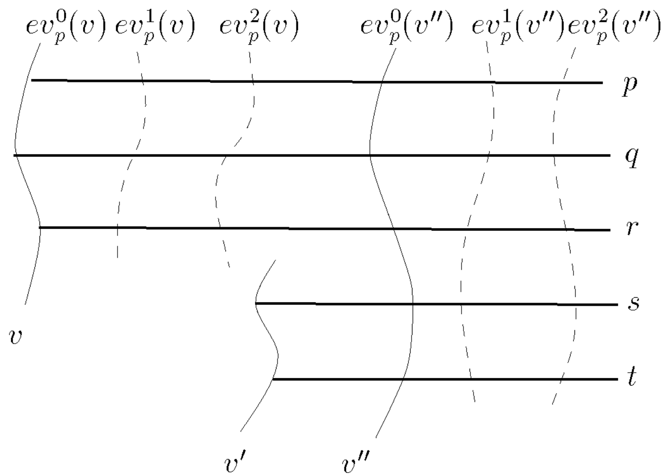


Fig. 9. Example of notation. Solid cuts indicate e-view changes corresponding to a modification in the composition of the view (i.e., a traditional view change), while dashed cuts indicate e-view changes where the view composition remains unchanged.

that these properties are satisfied in addition to those of view synchrony as defined in Appendix A1.

PROPERTY B.1 (Uniqueness). *E-views identified by their names are unique. In other words,* $\forall p, i \neq j : ev_p^i \neq ev_p^j$.

PROPERTY B.2 (Total Order). *Within any given view, e-view changes are totally ordered by processes in the view. In other words,*

$$\forall p, q \in comp(v) :$$
$$\left(vchg\left(ev_p^i(v)\right) \in h_p\right) \wedge \left(vchg\left(ev_q^i(v)\right) \in h_q\right) \Rightarrow$$
$$ev_p^i(v) = ev_q^i(v)$$

Upon delivery of e-view *ev*, process *p* cannot conclude that all other processes in *comp*(*ev*) will also eventually deliver *ev*. What is guaranteed, however, is that processes that survive a view change will have delivered the same set of e-view events in the previous view. This is formalized by the following property.

PROPERTY B.3 (E-view Agreement). *Given two consecutive views v and v′,*

$$\forall p, q \in comp(v) \cap comp(v') :$$
$$vchg\left(ev_p^i(v)\right) \in h_p \Rightarrow vchg\left(ev_q^i(v)\right) \in h_q$$

The next set of properties specify what we have informally called the *Structure Property* in Section 6.1.

PROPERTY B.4 (Initialization). *The first e-view delivered at a process is such that the process appears in a new subview by itself. In other words,*

$$\forall p : \textsc{SameSubview}\left(p, q, ev_p^0\right) \Rightarrow p = q$$

PROPERTY B.5 (Not Overlap). *Subviews do not overlap. In other words,*

$$\forall p, q, r :$$
$$\textsc{SameSubview}(p, q, ev) \wedge \textsc{SameSubview}(p, r, ev)$$
$$\Rightarrow \textsc{SameSubview}(q, r, ev)$$

PROPERTY B.6 (Structure 1). *A view change may split subviews only if their components result in different views. In other words, given two consecutive views v and v′,*

$$\forall p, q \in comp(v) \cap comp(v'):$$
$$\textsc{SameSubview}\left(p, q, ev_p^{\ell}(v)\right)$$
$$\Rightarrow \textsc{SameSubview}\left(p, q, ev_p^0(v')\right)$$

PROPERTY B.7 (Structure 2). *A view change never merges subviews. In other words, given two consecutive views v and v′,*

$$\forall p \in comp(v) \cap comp(v'), q \in comp(v'):$$
$$\neg\textsc{SameSubview}\left(p, q, ev_p^{\ell}(v)\right)$$
$$\Rightarrow \neg\textsc{SameSubview}\left(p, q, ev_p^0(v')\right)$$

Properties B.4-B.7 defined for subviews also hold for sv-sets. The following property states that delivery events for a given e-view at different processes define a consistent cut of the computation.

PROPERTY B.8 (Causal Order). *Causal precedence relations between message multicasts and e-view deliveries are preserved. In other words, if the delivery of an e-view ev and the multicast of message m both occur in view v and the former event causally precedes the latter, then any process that delivers m must have first delivered ev.*

Note that with respect to *view* changes, sets of delivered messages are totally ordered by all processes common to both views. With respect to *e-view* changes, however, the ordering guarantee preserves only causal precedence. Extending the total ordering of messages with respect to e-view changes as well would require more expensive protocols that would make our extensions less suitable to a large-scale system.

Finally we give properties for the primitives *Subview-Merge*() and *SV-SetMerge*() described in Section 6.1. Recall that *prvk*() denotes the event corresponding to the invocation of these primitives.

PROPERTY B.9 (Provoked E-views).

1) *Multiple prvk() requests executed by the same process are serviced in FIFO order,*
2) *Each prvk() event generates at most one vchg() event,*
3) *Each vchg() event corresponds to at most one prvk() event,*
4) *A vchg() event without a corresponding prvk() event must be a view change.*

Consider a *prvk*() event issued by process *p* in e-view *ev*. Let the predicate LEGAL(*prvk*()) be true if and only if the parameters of *SubviewMerge*() (*SV-SetMerge*()) contain at least two subviews (sv-sets) defined in *ev*. In case the primitive is *SubviewMerge*(), the subviews defined in *ev* must all belong to the same sv-set as the process invoking the primitive.

PROPERTY B.10 (Nontriviality). *If the predicate* LEGAL(*prvk*()) *holds when some process p issues prvk() in e-view ev, then eventually either p will crash or it will be delivered the vchg(ev′) event in ev that is provoked by prvk(). If the predicate* LEGAL(*prvk*()) *does not hold when p issues prvk(), then p will never be delivered the vchg(ev′) event.*

A3 Implementing Enriched View Synchrony

To implement Enriched View Synchrony, it suffices to extend a run-time support for view synchrony as sketched informally in this Appendix. We assume that the reader is familiar with the basic mechanisms required to implement view synchrony [2], [12], [3].

Each process is associated with two identifiers (in addition to its name), one for subviews, the other for sv-sets. Processes belong to the same subview (sv-set) if their corresponding identifiers are identical. An e-view *ev* is represented by its name and, for each process in *comp*(*ev*), by the triple (name, subview, sv-set) of identifiers. The name of *ev* is constructed by coupling the name of the current view (as constructed by view synchrony) with the number of e-view changes delivered in that view before the delivery of *ev*.

Provoked e-view changes are implemented by sending a request to a designated coordinator process in the current view. The sending process records such a request until delivery of the matching e-view. The coordinator constructs a message describing the new e-view and (reliably) multicasts it within the view using view synchrony. Upon delivery of this message, the run-time support updates accordingly its representation of *ev* and delivers to the application the corresponding *vchg*(*ev*) event.

Requests arriving at the coordinator are queued and processed in FIFO order, that is, in consecutive e-views. Moreover, once the coordinator has started processing a given request, it does not participate in any view agreement until it has delivered the corresponding *vchg*(*ev*) event. In other words, the sequence of steps <process request, multicast new e-view, deliver new e-view> is not interrupted by view changes.

View agreement is augmented as follows. Processes participating in the agreement exchange the local information related to:

a) previous e-view; and
b) requests sent to the coordinator for which the matching e-view has not yet been delivered.

In principle, this information may be easily piggybacked onto the existing messages for agreement. Before installing the e-view corresponding to the new view, existing subview (sv-set) identifiers may have to be modified, otherwise the run-time support could merge subviews (sv-sets) that come from different views but happen to have the same identifier, thus violating the Structure 2 property. Each process constructs the new e-view by modifying the identifiers of subviews and sv-sets so that:

1) Properties B.6-B.7 (Structure 1 and Structure 2) are fulfilled; and
2) The new e-view has the same representation at all processes participating in the agreement (e.g., the identifiers of any given process are the same at all processes).

We omit the details for sake of brevity. This step may be performed locally, provided each process is able to figure out, for any pair of processes in view *v* that is being installed, whether they are joining in *v* or they already belonged to the same view. Even this case can be handled without introducing any additional messages. Finally, processes elect the coordinator, that constructs a queue of pending requests based on b and on the new view composition (e.g., it discards pending requests originated by processes that do not belong anymore to its view).

Message exchange is implemented by means of a simple causal delivery algorithm in which all multicast events between two consecutive e-view changes are treated as if they were concurrent: e-views in the same view are numbered consecutively; each message includes an indication of the e-view in which it was sent; a message *m* is delivered to the process immediately after being received if the e-view in which *m* was multicast has already been delivered, otherwise *m* is inserted in a buffer; upon delivering an e-view *ev*, all buffered messages that were multicast in *ev* are also delivered. Each of these actions is atomic in the sense that it is not interrupted by view changes, which may be achieved easily because the related code executes in the run-time support. Once again, this algorithm does not require additional message exchanges.

We give a very informal correctness argument for the above implementation. Property B.1 (Uniqueness) is straightforward. As for Property B.2 (Total Order), let m_1 and m_2 be any two messages, each requesting to install a new e-view, and that are delivered in view *v* (that is, the last view agreement completed before their delivery is the one that installed *v*). Observe that a view change cannot occur in between the sending of either message and its delivery, because, in that interval, a coordinator does not participate in view agreement. It immediately follows that the sender of messages m_1 and m_2 must be the same process. Since multicasts are FIFO-ordered, any two processes that deliver m_1 and m_2 deliver them in the same order. Property B.3 (E-view Agreement) follows immediately from the fact that these messages are multicast with view synchrony semantics.

Property B.5 (Not Overlap) is guaranteed because each process is associated with a single subview (sv-set) identifier, and this identifier is identical at all processes in the view. Properties B.6 and B.7 (Structure 1 and 2) follow from the updating of identifiers during view agreement, that we have omitted. The same consideration applies to Property B.4. Property B.8 (Causal Order) is guaranteed by the algorithm that implements message exchange. The fact that this algorithm does not interfere with the semantics of view synchrony, that is, that processes surviving a view change have delivered the same set of messages in the previous view, follows from:

1) Property B.3 (E-view Agreement); and
2) the handling of buffered messages is not interrupted by view changes.

As for the primitives for provoking e-view changes, Property B.9 (Provoked E-views) follows from:

1) there is a single coordinator in each view;
2) the coordinator sends messages instructing to deliver e-view events (that are not view changes) only upon receiving requests issued by other processes in the view;
3) the coordinator sends at most one such message for each request;
4) the coordinator processes requests in FIFO order.

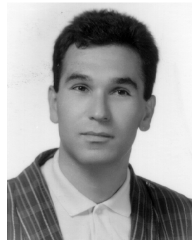
Moreover, in each e-view, the coordinator evaluates the next request and, if the request may be satisfied, it delivers the corresponding e-view event before analyzing the next request. The proof of Property B.10 (Nontriviality) may be obtained by combining this observation with Property B.2 (Total Order).

ACKNOWLEDGMENTS

This work has been supported in part by the Commission of European Communities under ESPRIT Programme Basic Research Project 6360 (BROADCAST), the Italian National Research Council and the Italian Ministry of University, Research and Technology (MURST 40%). We are grateful to Ken Birman and the anonymous referees for their comments and suggestions leading to an improved presentation.

REFERENCES

[1] M.J. Fischer, N.A. Lynch, and M.S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM*, vol. 32, no. 2, pp. 374-382, Apr. 1985.
[2] K. Birman, R. Cooper, T. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, and M. Wood, *The ISIS—System Manual, Version 2.1*, Dept. of Computer Science, Cornell Univ., Sept. 1993.
[3] A. Schiper and A. Ricciardi, "Virtually-Synchronous Communication Based on a Weak Failure Suspector," *Proc. 23rd Int'l Symp. Fault-Tolerant Computing*, pp. 534-543, June 1993.
[4] K.P. Birman, *Reliable Distributed Computing with the Isis Toolkit*, chapter "Virtual Synchrony." Los Alamitos, Calif.: IEEE CS Press, 1994.
[5] D. Malki, Y. Amir, D. Dolev, and S. Kramer, "The Transis Approach to High Availability Cluster Communication," Technical Report CS94-14, Inst. Computer Science, The Hebrew Univ. of Jerusalem, 1994.
[6] Ö. Babaoglu, R. Davoli, and A. Montresor, "Group Membership and View Synchrony in Partitionable Asynchronous Systems: Specifications," Technical Report UBLCS-95-18, Dept. of Computer Science, Univ. of Bologna, Sept. 1996.
[7] A. Schiper, A. Ricciardi, and K. Birman, "Understanding Partitions and the "No Partition" Assumption," *Proc. Fourth IEEE Workshop Future Trends of Distributed Systems*, pp. 354-360, Sept. 1993.
[8] A. Schiper and A. Sandoz, "Primary Partition Virtually Synchronous Communication Harder Than Consensus," *Distributed Algorithms*, G. Tel and P. Vitányi, eds., *Lecture Notes in Computer Science*, pp. 39-52. Springer-Verlag, 1994.
[9] T.D. Chandra and S. Toueg, "Unreliable Failure Detectors for Asynchronous Systems," *Proc. 10th ACM Symp. Principles of Distributed Computing*, pp. 325-340, Aug. 1991.
[10] D. Skeen, "Determining the Last Process to Fail," *ACM Trans. Computer Systems*, vol. 3, no. 1, pp. 15-30, Feb. 1985.
[11] Ö. Babaoglu, A. Bartoli, and G. Dini, "Replicated File Management in Large-Scale Distributed Systems," *Distributed Algorithms*, G. Tel and P. Vitányi, eds., *Lecture Notes in Computer Science*, pp. 1-16. Springer-Verlag, 1994.
[12] Ö. Babaoglu, R. Davoli, L.A. Giachini, and M.G. Baker, "Relacs: A Communications Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems," *Proc. 28th Hawaii Int'l Conf. System Sciences*, pp. 612-621, Jan. 1995.
[13] R. van Renesse, K.P. Birman, and S. Maffeis, "Horus: A Flexible Group Communication System," *Comm. ACM*, vol. 39, no. 4, pp. 76-83, Apr. 1996.
[14] A. Schiper and A. Sandoz, "Uniform Reliable Multicast in a Virtually Synchronous Environment," *Proc. 13th Int'l Conf. Distributed Computing Systems*, pp. 561-568, May 1993.
[15] I. Keidar, "A Highly Available Paradigm for Consistent Object Replication," MS thesis, Inst. of Computer Science, The Hebrew Univ. of Jerusalem, 1994, Also available as Technical Report CS95-5.
[16] D. Dolev, D. Malki, and R. Strong, "A Framework for Partitionable Membership Service," Technical Report CS95-4, Inst. of Computer Science, The Hebrew Univ. of Jerusalem, 1995.
[17] F.M. Kaashoek and A.S. Tanenbaum, "An Evaluation of the Amoeba Group Communication System," *Proc. 16th Int'l Conf. Distributed Computing Systems*, pp. 436-447, May 1996.
[18] M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D.C. Steere, "Coda: A Highly Available File System for a Distributed Workstation Environment," *IEEE Trans. Computers*, vol. 39, no. 4, pp. 447-459, Apr. 1990.
[19] A. El Abbadi, D. Skeen, and F. Cristian, "An Efficient, Fault-Tolerant Protocol for Replicated Data Management," *Proc. Fourth ACM Symp. Principles of Database Systems*, pp. 215-229, 1985.
[20] I. Keidar and D. Dolev, "Increasing the Resilience of Atomic Commit at No Additional Cost," *Proc. 14th ACM Symp. Principles of Database Systems*, pp. 245-254, May 1995.
[21] L.E. Moser, Y. Amir, P.M. Melliar-Smith, and D.A. Agarwal, "Extended Virtual Synchrony," *Proc. 14th Int'l Conf. Distributed Computing Systems*, pp. 56-65, June 1994.
[22] F. Cristian, "Group, Majority, and Strict Agreement in Timed Asynchronous Distributed Systems," *Proc. 26th Int'l Symp. Fault-Tolerant Computing*, June 1996.
[23] J.H. Saltzer, D.P. Reed, and D.D. Clark, "End-to-End Arguments in System Design," *ACM Trans. Computer Systems*, vol. 2, no. 4, pp. 277-288, Nov. 1984.
[24] A. El Abbadi and S. Toueg, "Maintaining Availability in Partitioned Replicated Databases," *ACM Trans. Databases Systems*, vol. 14, no. 2, pp. 264-290, June 1989.
[25] J. Howard and S. Katz, "Reconciliations," *Proc. 13th ACM Symp. Principles of Distributed Computing*, pp. 14-21, 1994.
[26] M. Evangelist, N. Francez, and S. Katz, "Multiparty Interactions for Interprocess Communication and Synchronization," *IEEE Trans. Software Eng.*, vol. 15, no. 11, pp. 1,417-1,426, Nov. 1989.



Özalp Babaoglu received a PhD in 1981 from the University of California at Berkeley where he was one of the principal designers of BSD Unix. He is a professor of computer science at the University of Bologna, Italy. Before moving to Bologna in 1988, Dr. Babaoglu was an associate professor in the Department of Computer Science at Cornell University. He is active in several European research projects exploring issues related to fault tolerance and large scale in distributed systems. Dr. Babaoglu serves on the editorial boards for ACM *Transactions on Computer Systems* and Springer-Verlag *Distributed Computing*.

Alberto Bartoli received a degree in electrical engineering, cum laude, in 1989, and a doctorate in computer engineering in 1994, both from the University of Pisa. He is an assistant professor in the Dipartimento di Ingegneria dell'Informazione of the University of Pisa, Italy. His research interests include large scale distributed systems, group-based computing, and mobile computing.

Gianluca Dini received a degree in electrical engineering from the University of Pisa in 1990, and a doctorate in computer science from the Scuola Superiore S. Anna, Pisa, in 1995. He is an assistant professor of computer science at the University of Pisa, Italy. His research interests are in distributed systems with special emphasis on large scale distributed systems, distributed programming, distributed systems architecture.