On Programming with View Synchrony*

Özalp Babaoğlu
University of Bologna
Department of Computer Science
Piazza Porta S. Donato 5, 40127 Bologna, Italy
ozalp@cs.unibo.it

Alberto Bartoli, Gianluca Dini
University of Pisa
Department of Information Engineering
Via Diotisalvi 2, 56126 Pisa, Italy
{alberto,gianluca}@iet.unipi.it

Abstract

View synchrony has been proposed as a programming paradigm for developing reliable distributed applications. The paradigm is particularly attractive when the underlying computing system is asynchronous and prone to complex failure scenarios including partitions. View synchrony encourages a programming style where groups of processes cooperate closely in order to maintain some form of shared state among them. In this paper, we examine the technical problems that arise in shared state management when programming applications using view synchrony. We identify three classes of problems corresponding to state transfer upon group joins, state recreation after total failures and state merging after partition unions. We argue that shared state problems are inherent to any implementation, and without explicit support, attempts to solve them may easily obscure much of the simplicity and elegance of view synchrony. Finally, we propose an extension to the traditional view synchrony model based on the notion of subviews that addresses the problems raised by shared state management.

1. Introduction

Implementing a distributed algorithm involves making realistic assumptions on the underlying computing system. In many practical environments, including the Internet, the system is adequately modeled as an asynchronous distributed system where both processes and communication links may fail by crashing. Unfortunately, reasoning about distributed algorithms in such systems is difficult. This is especially the case when application requirements result in processes that interact closely. Unpredictable communication delays and computation speeds due to transient failures and highly-variable loads greatly complicate reasoning based on time and timeouts. Furthermore, failures may result in complex communication scenarios that include network partitions. In such systems, the inability to communicate with a certain process cannot be attributed to its real cause — the process may have crashed, it may be slow, the communication path may have been disconnected or it may be experiencing long delays [7].

An abstraction that has the potential for greatly simplifying both reasoning about and implementation of distributed applications is *view synchrony*.[1] [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, it provides global guarantees about the set of messages that have been delivered by processes as a function of the view changes that they observe. As such, it allows global reasoning based only on local information.

In this paper we consider programming reliable applications based on view synchrony. Such applications typically require maintaining certain state information that is distributed and/or replicated among the group members. Although view synchrony can be a great help 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 so-called *shared state* problems in terms of system events provoking them: processes joining a group give raise 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 the scenarios

*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%).

[1] The abstraction was first introduced in the Isis system where it is known as *virtual synchrony* [5]. 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.

that provoke them do not permit global reasoning with local information only. 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 propose an extension to the traditional view synchrony model based on *subviews* that is capable of simplifying application development with respect to the shared state problems.

2. System Model and View Synchrony

The system is a collection of processes executing at potentially remote sites that communicate via a network. Both processes and communication links may fail by crashing. We model the recovery of a process by assigning it a new identifier. We assume an infinite name space of process identifiers to model infinite executions in which processes may fail and recover an arbitrary number of times. At any time, however, there are only a finite number of processes in the system. 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 variable load on the computing and communication resources of any practical distributed system. Asynchronous systems place fundamental limits on what can be achieved by distributed computations in the presence of failures [7]. The principal difficulty stems from the inability to distinguish slow processes or communication links from those that have actually failed.

View synchrony implements the notion of a *process group* and provides reliable multicast as the basic communication primitive [9, 6, 8, 4]. Processes that would like 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 may communicate through *multicast* messages. For the multicast primitive to be terminating despite failures in an asynchronous system, view synchrony includes a membership service that provides consistent information in the form of *views* regarding the components of the group that are currently believed to be up and reachable. At each process, view synchrony accepts messages for multicasting, delivers messages and new views reflecting changes in the group's composition. The major utility of view-synchronous communication is its ability to abstract away failures, whether real or due to false suspicions, and transform them into *view change* events. 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.

The real utility of view synchrony is not in its individual components — reliable multicasts and membership service — but in their integration. Informally, view synchrony can be specified through the following properties on message deliveries:

Property 2.1 (Agreement) *All processes that survive from one view to the same next view deliver the same set of messages.*

Property 2.2 (Uniqueness) *A message is delivered in at most one view.*

Property 2.3 (Integrity) *A message is delivered at most once by any process and only if some process actually multicast it.*

This specification can be completed by adding properties on view compositions and view installations governing the group membership service [4]. Note that there are no conditions imposed on the relative ordering of messages delivered within a given view. It will become clear from our discussion that such guarantees can only help in *solving* shared state problems but cannot *prevent* them.

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 failures. Including permanent data as part of the local state allows us to model applications that may recover after failures.

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

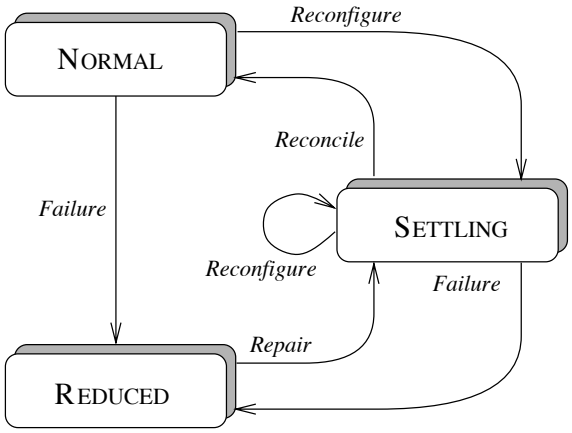

Figure 1. Possible execution modes for a group object process. Each transition is labeled by its cause.

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 global states 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 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 executing 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 Figure 1. An application may be structured such that only a subset of these modes or transitions are relevant.

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 even though the actual cause may be a process crash, communication failure or false suspicions as discussed in Section 2. A view change that restores the conditions 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 re-establishing connectivity between two partitions. To return back 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 has to go on 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* characterize overlapping global state reconstruction instances.

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 allowable to return stale data (missing some of the more recent updates). One possible implementation of this group object is to associate with each replica of the file a vote and define a quorum to be a collection of votes that can be obtained in at most one concurrent view. For this example, the interpretation of the various modes for a process is as follows. Any current view defining a quorum corresponds to *N-mode* since both read and write operations can be supported. A non-quorum view corresponds to *R-mode* since reads can be supported but not writes. Finally, any view in which only a subset of the processes hold up-to-date replicas of the file correspond to *S-mode*. A process in *S-mode* must obtain an up-to-date copy of the file (if it does not already have it) before returning to *N-mode*.

Next consider a group object implementing a database with a 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 subset of the database. Clearly 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 performing the look-up among the view members. An inconsistency in this global state information could result in some portion of the database not being searched at all or being searched multiple times.

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 the $vchg$ 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 since it joined the group. In other words, after k delivery 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 re-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. For simplicity, we assume that the mode function may be dependent on the entire history with respect to message deliveries, but depends only on the current view with respect to view changes. 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 activity necessary for this reconciliation 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 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 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.

If a shared state problem occurs and which internal operations must be performed to solve it depends highly on the application. For this reason we give only *necessary* conditions for the shared state problem. Deriving also sufficient conditions for it requires knowledge of the application semantics and thus cannot be achieved in general. For sake of brevity, we present the necessary conditions without formal proofs.

Let us focus on the *Repair* and *Reconfigure* transitions that lead to *S-mode* and bring about the shared state problem. Consider the event $vchg(v)$ 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 any 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 fails.

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*. 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 when they were in *N-mode*.[2] 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 $RS(v)$ suspend serving external operations, compare its local state to the state of at least one process in NS and possibly modify it as a consequence of this comparison before resuming its normal operation.

State Creation. This problem arises whenever the global state must be reconstructed from scratch, for example

[2] The same reasoning holds for processes in RS. However, this case is not meaningful for our discussion.

after a total failure scenario. A necessary condition for the creation problem is that $NS(v)$ is empty but $RS(v)$ is not. Creation involves having each process p in v suspend serving external operations and compare its local state to the state of all other processes in v. Before resuming its activity, p may have to modify its local state as a consequence of such a comparison. Identifying which local state is to be used for recreation of the others may require determining the last process to fail [11].

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 somehow reconciles the divergence that may have taken place. The necessary condition for this situation is that NS is not empty and is composed of at least two clusters.

The state merging problem does not exclude the possibility that the set RS is also non empty. 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.

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. To be able to classify with local information only, the process has access to the new view composition as provided by view synchrony. Unfortunately, this information is not sufficient for classification of the shared state problem since views as defined by view synchrony are flat structures and do not contain information regarding RS, NS and possible clusters.

For example, suppose that 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[3] 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 complex and costly protocols [11]. 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). Effectively attacking these problems depends mostly on the application semantics and programming skills [1]. Clearly, this asynchrony is a source of significant complexity that may obscure the conceptual simplicity and elegance of view synchrony.

[3] The set $RS(v)$ contains at least process p itself.

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 [5]. The programmer has to only define what constitutes 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 do 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 run-time 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 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 process 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 [3], Horus [12] and Transis [8] 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 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 [10].

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 state merger problems do not exist by definition. The price to pay for this simplification is the inability to support applications with weak consistency requirements that could make progress in multiple concurrent partitions.

It is highly desirable for systems implementing view sysnchrony 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 (small) piece that needs to be transferred in synchrony with the join event; another (large) piece that can be transferred concurrently with application activity in the new view [1]. 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.

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* extension we propose in the next Section is an attempt in this direction.

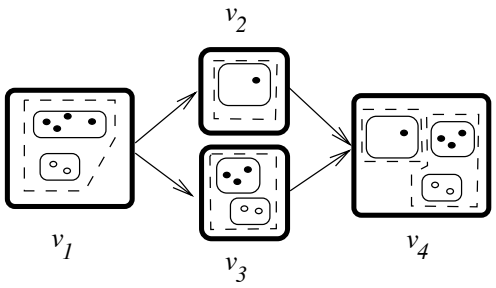

Figure 2. Basic features of the enriched view synchrony model. Views, subviews and sv-sets are indicated with thick, thin and dashed frames, respectively.

6. Extensions to View Synchrony

In this section we present a novel extension to view synchrony that is aimed to help programmers in reasoning about the shared state problem. This extension requires minor modifications to the view synchrony run-time support and can be implemented efficiently [2].

6.1. Enriched Views

Our proposed extension to view synchrony is based on the notions of *subviews* and *subview sets* (*sv-sets* for short). 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.

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. Along any given cut of the computation, 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 nnd sv-sets ever 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 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 (See Figure 2).

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

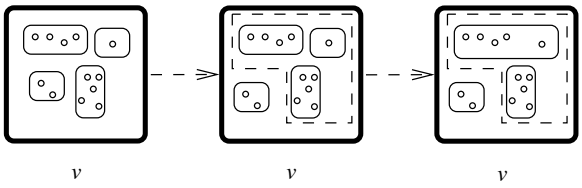

Figure 3. The figure illustrates two e-view changes within a single view v. For simplicity, sv-sets that contain a single subview are omited.

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 simply cannot 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, may be admitted into an existing subview. As with views, failures may cause subview and sv-set compositions to shrink at arbitrary times, asynchronously to the application (See Figure 2).

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* and return its identifier.

SubviewMerge(*sv-list***)** Create a new subview that is the union of the subviews given in *sv-list* and return its identifier. If all the subviews in *sv-list* do not initially belong to the same sv-set, the call has no effect. The resulting subview belongs to the sv-set containing the input subviews.

Figure 3 illustrates a sequence of e-view changes provoked by the above calls. The first change is due to a *SV-SetMerge()* call merging three sv-sets, each containing a single subview. The second change is due to a *SubviewMerge()* call merging two of the subviews in the newly created sv-set.

This extended service maintains the semantics of view synchrony regarding view changes and message deliveries, exactly as described in Section 2. With respect to e-view changes, the following additional properties are guaranteed, which we state informally:

Property 6.1 (Total Order) *The 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.*

6.2. Structuring Applications based on Enriched Views

Our proposed extension to view synchrony presents an opportunity for systematic and simplified solutions to shared state problems. It enhances the global reasonings that can be achieved based on local information after view changes and simplifies the 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 subview and not across different subviews; (2) Internal operations are performed across subviews belonging to the same sv-set; upon successful completion of an internal operation, the corresponding subviews are merged into a single one.

As an example, suppose that external operations can be run only in a view containing a majority of processes and that their implementation involves the management of a mutually-exclusive write lock within such a 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 as the immediate successor of view u such that v defines a majority 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 majority. It cannot distinguish between the following scenarios: (i) a majority already existed in one of the immediate predecessors of v (i.e., a state transfer

since NS and RS are both non empty); (ii) the shared state was being reconstructed at the time v was installed (i.e., a creation problem since NS is empty but RS is not); (iii) a majority is reborn after it had disappeared temporarily (i.e., a creation problem 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 majority, 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 majority, then cases (ii) and (iii) can be distinguished by controlling if v contains an sv-set defining a majority.

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 (ii) 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.

7. Conclusions

Shared state problems such as state transfer, merging and creation, are likely to be an issue in most applications designed to run on top of view synchrony. We have given a characterization for them in terms of necessary conditions and presented an analysis of related problems that arise in practical applications.

Even though view synchrony has the potential for being a clean and elegant programming framework, this elegance can easily be lost in practice, unless special provisions are made for supporting shared state maintenance. A major reason for this is the asynchrony between run-time support and application, that may lead to shared state problems at instants that are inopportune for the application. This in turn results in modifications to the entire application and not just those parts responsible for handling events that trigger shared state problems. Finally, we have presented a simple extension to view synchrony that can effectively attack the shared state problem.

Acknowledgments

We are grateful to Ken Birman and the anonymous referees for their comments and suggestions leading to an improved presentation.

References

[1] O. Babaoğlu, A. Bartoli, and G. Dini. Replicated File management in large-scale distributed systems. In G. Tel and P. Vitányi, editors, *Distributed Algorithms*, Lecture Notes in Computer Science, LNCS 857, pages 1–16. Springer-Verlag, 1994.

[2] O. Babaoğlu, A. Bartoli, and G. Dini. Enriched View Synchrony: A Paradigm for Programming Dependable Applications in Partitionable Asynchronous Distributed Systems. Technical Report UBLCS-96-3, Department of Computer Science, University of Bologna, Feb. 1996.

[3] O. Babaoğlu, R. Davoli, L. Giachini, and M. Baker. Relacs: A communications infrastructure for constructing reliable applications in large-scale distributed systems. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, pages 612–621, Maui, Hawaii, Jan. 1995.

[4] Ö. Babaoğlu, R. Davoli, and A. Montresor. Failure detectors, group membership and view-synchronous communication in partitionable asynchronous systems. Technical Report UBLCS-95-18, Department of Computer Science, University of Bologna, Nov. 1995.

[5] K. Birman, R. Cooper, T. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, and M. Wood. *The ISIS - System Manual, Version 2.1*. Department of Computer Science, Cornell University, Sept. 1993.

[6] K. P. Birman. *Reliable Distributed Computing with the Isis toolkit*, chapter Virtual Synchrony. IEEE Computer Society Press, Los Alamitos, CA, 1994.

[7] M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, Apr. 1985.

[8] D. Malki, Y. Amir, D. Dolev, and S. Kramer. The Transis approach to high availability cluster communication. Technical Report CS94-14, Institute of Computer Science, The Hebrew University of Jerusalem, 1994.

[9] A. Schiper and A. Ricciardi. Virtually-synchronous communication based on a weak failure suspector. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, pages 534–543, June 1993.

[10] A. Schiper and A. Sandoz. Uniform reliable multicast in a virtually synchronous environment. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 561–568, May 1993.

[11] D. Skeen. Determining the last process to fail. *ACM Trans. Comput. Syst.*, 3(1):15–30, Feb. 1985.

[12] R. van Renesse, T. Hickey, and K. Birman. Design and Performance of Horus: A Lightweight Group Communication System. Technical Report TR-94-1442, Department of Computer Science, Cornell University, Aug. 1994.