The Inherent Cost of Strong-Partial View-Synchronous Communication

Özalp Babaoğlu **Renzo Davoli** **Luigi Giachini**
Paolo Sabattini
Department of Computer Science, University of Bologna, 40127 Bologna, Italy
email: Ozalp.Babaoglu@cs.unibo.it

Process groups and group-based communication have proven to be useful paradigms for structuring reliable applications in distributed systems. The approach is typically realized through view-synchronous communication (VSC) *that integrates a reliable multicast facility with a group membership service in the form of view changes. In this paper we examine algorithmic issues associated with VSC group membership in large-scale distributed systems where network partitions may result in multiple views to be active concurrently. We first derive necessary conditions on the partial order of installed views such that VSC is meaningful and solvable in the presence of partitions. We then prove that* strong-partial VSC*, which guarantees concurrent views to be disjoint, is not easier than atomic commitment. As such, all know lower bound results for atomic commitment are also lower bounds for this problem, including the impossibility of nonblocking solutions in the presence of communication failures. We discuss the practical implications of our results in constructing group communication facilities for large-scale distributed systems.*

1 Introduction

An increasing number of applications with reliability and global consistency requirements are being deployed over distributed systems with extremely large geographic extent. Scientific computing, data sharing, collaborative work, finance and commerce are typical application domains with such requirements. For the application builder, large-scale distributed systems, exemplified by the Internet, present new technical challenges beyond those that are faced in local-area network environments. Most notably, the abundance of long-haul links and the sparse connectivity of the communication fabric may result in failures to provoke network partitions that are frequent and long lasting. Even in the absence of failures, asynchrony that prevents communication delays to be bounded, may lead to the formation of "virtual partitions" that are indistinguishable from real ones [26].

Process groups and group-based communication, initially conceived simply as structuring and naming mechanisms [12], have later proven to be appropriate support technologies for reliable computing in distributed systems [10]. Aspects of the paradigm that make it particularly suitable for building reliable applications include a membership service that transforms failures into view changes and a reliable multicast facility with delivery guarantees for communicating within the group. The abstraction is typically realized through a *view-synchronous communication service (VSC)* [8, 10, 27], which defines global ordering guarantees with respect to the set of messages delivered in response to multicasts and view changes installed in response to failures and repairs. A

large number of systems have been built that incorporate various flavors of the basic ideas in a local-area network setting [8, 23, 22, 24].

Recently, there have been numerous proposals advocating group-based communication as a suitable infrastructure for developing reliable applications also in large-scale distributed systems [3, 6, 15, 2, 14]. The principal issue to be addressed in refining VSC semantics to this environment has to do with group membership management. Unlike local-area network based systems, VSC in large-scale distributed systems may result in multiple views of the group to exist concurrently as a result of virtual or real partitions. One possibility is to avoid the issue of concurrent views by defining the notion of a *primary partition* such that at most one view exists at any given time [25, 29]. The resulting service is called *linear group membership* since it defines a total order among all installed views and is appropriate for applications with strict consistency requirements such as replicated data management with one-copy semantics [4].

For certain classes of applications such as collaborative work, scientific computing or weak-consistency data sharing, progress may be possible and desirable in multiple partitions. To minimize latency in such system, it is appropriate for the membership service to deliver views (even concurrent ones) to all functional components and let the application decide if progress is possible. The resulting service is called *partial group membership* in that the set of installed views defines a partial order [1, 21, 27, 6]. The final issue has to do with the composition of concurrent views. If two concurrent views are allowed to overlap arbitrarily in their composition, the service is called *weak-partial group membership* [21, 27]. If, on the other hand, concurrent views are guaranteed to be disjoint in their composition, the service is called *strong-partial group membership*[1]. Weak-partial semantics is not of practical interest since it is impossible to guarantee VSC message delivery semantics on top of it [5]. Strong-partial semantics, on the other hand, not only may be useful in its own right, but it can also be the basis of a linear membership service.

In this paper we consider the problem of implementing a view-synchronous communication service with strong-partial semantics in large-scale distributed systems. Our contribution is severalfold. We first derive a set of conditions that are necessary for the feasibility and well-foundedness of VSC when concurrent views are possible. We then give a formal specification for the service in terms of an abstract problem, called SP-VSC, and characterize the inherent costs of solving SP-VSC. We do so by proving that the well-known atomic commitment problem (ACP) [20, 7] of distributed transactions reduces to SP-VSC in an asynchronous system. As such, all lower bound results that have been obtained for ACP are also applicable to SP-VSC. Of particular interest are results related to nonblocking solutions — in the best case, nonblocking algorithms for SP-VSC require roughly twice as much time (as measured in end-to-end message delays) than their blocking counterparts [13]; if communication failures are admitted, there exists no nonblocking algorithm for solving SP-VSC [18, 9]. Practical implications of our results for large-scale distributed systems are significant: the cost of achieving strong-partial VSC may be prohibitive given that message delays can be extremely large; nonblocking solutions are impossible since communication failures are an intrinsic characteristic of these systems.

The rest of the paper is organized as follows. In the next Section we define the system model in which the problem is formulated and solved. Section 3 gives a set of properties on views and message deliveries that any view-synchronous communication must satisfy. In Section 4 we derive a set of

[1] Note that in [17], the terms *strong* and *weak* are used to characterize VSC based on guarantees for the view in which a message is delivered with respect to the one in which it was multicast.

necessary conditions on view installations such that view-synchronous communication with strong-partial semantics is well defined and solvable. Section 5 is the formal specification of the desired service in terms of the abstract problem SP-VSC. The main results of the paper are presented in Section 6 where we prove that atomic commitment reduces to SP-VSC. We discuss related work in Section 7 and the practical implications of our results in Section 8.

2 System Model

The system is a collection of processes executing at potentially remote sites. Processes communicate through a message exchange service provided by the network. The network need not be fully connected and is typically quite sparse. Both processes and communication links may fail by crashing. The system is asynchronous in the sense that communication delays cannot be bounded and there is no global clock. Processes are associated unique names drawn from a potentially-infinite domain. A process maintains the same name throughout its life as long as it does not crash. Recovery of a process after a crash is modeled by renaming it. Note, however, that other events such as view changes due to network partitions or mergers cannot rename processes.

Execution of process p is modeled through a sequence of events called its *history* and denoted h_p. Histories may be infinite sequences modeling infinite executions or they may be finite in which case the last event is either $leave$ or $crash$ representing, respectively, termination (by leaving the group) or failure of the process. Other relevant events that may appear in process histories are $mcast(m)$, $dlvr(m)$ and $vchg(v)$ denoting multicast of message m, delivery of a multicast message m and view change installing new view v, respectively. When necessary, we add subscripts to events to denote the process at which they occur. The prefix of history h_p containing the first k events executed by process p is denoted $h_p(k)$.Without loss of generality, we assume that the first event of any history is a view change that installs the initial view for the group.

Asynchronous systems place fundamental limits on what can be achieved by distributed computations in the presence of failures [16]. The principal difficulty stems from the inability to distinguish slow processes or communication links from those that have actually failed. In order to guarantee progress despite this limitation, one possibility is to employ unreliable *failure suspectors* in asynchronous systems [11]. Such "oracles" are allowed to make mistakes but have to satisfy a set of weak requirements (that are achievable) for them to be useful nevertheless [11, 27]. In this paper we are not interested in the technical issues regarding the specification and implementation of failure suspectors. It suffices to note that false suspicions are sufficient for the formation of virtual partitions [26], which the view-synchronous communication service has to cope with. In the next Section we define this service.

3 View-Synchronous Communication Service

The basic primitive of *view-synchronous communication service* (VSC) is the reliable multicast of a message to a group of processes. For the multicast primitive to be terminating in an asynchronous system with failures, VSC 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, VSC delivers multicast messages and view changes (through the $dlvr(m)$ and $vchg(v)$ events). View changes are triggered

by process crashes and recoveries, communication failures and repairs, network partitions and mergers, or explicit requests to join and leave the group. For sake of simplicity, we consider VSC as applied to a single group that has already been composed through joins.

In an asynchronous system, correctness and reachability of group members as perceived by individual processes (through their failure suspectors) may be inconsistent. It is the task of VSC to guarantee that such information is turned into consistent views through the membership service. The real utility of VSC is not in its individual components — reliable multicasts and membership service — but in their integration. Informally, VSC guarantees that sets of delivered messages are totally ordered with respect to view changes. We now give a formal definition for VSC in terms of properties that view changes and message deliveries have to satisfy.

3.1 View Changes

Views are identified through unique identifiers drawn from a potentially-infinite domain. Given a view v, the function $comp(v)$ returns the composition of v as a set of process names. Note that view identifiers are sufficient to distinguish views even if they have the same composition.

At each process p, the sequence of $vchg_p(v)$ events defines a total order among the installed views. Events at a process are said to be executed *in the view* that was most recently installed. More formally,

Definition 3.1 (Current View)*Let e_p^k denote the kth event in the history of process p. We say that event e_p^k is* executed in view v, *denoted $view(e_p^k) = v$, if*

$$(\exists i < k : e_p^i = vchg_p(v)) \wedge (\nexists w : vchg_p(w) \in h_p(k) - h_p(i)).$$

We next define an ordering relation among views. With respect to a single process, this order is defined by the sequence of view installation events. Globally, views are related if *some* process installs them in a particular order.

Definition 3.2 (Successor View)*Given two views v and w, view w is called the* immediate successor of view v at process p, *denoted $v \prec_p w$, if $view(vchg_p(w)) = v$. If there exists some process p such that $v \prec_p w$ holds, w is called the* immediate successor *of v and we write $v \prec w$. The relation $\prec^*$ denotes the transitive closure of $\prec$.*

Note that the relation $\prec^*$ does not define a total order on views. It is possible for two views to be incomparable with respect to $\prec^*$, in which case they are called *concurrent*. The possibility of concurrent views is precisely the aspect of VSC that makes it suitable for large-scale systems where partitions may occur. For VSC to be useful as an application development abstraction, there have to be some global guarantees regarding the order in which views are installed by different processes, even in the presence of partitions. In particular, those processes that install two given views should order them consistently in their histories. This requirement can be restated as follows:

Property 3.1 (Partial Order)*The successor relation $\prec^*$ induces a partial order on the set of installed views. In other words,*

$$\forall v, w : (v \prec^* w) \Rightarrow (w \not\prec^* v).$$

It is typical to represent the partial order relation among views as a labeled directed acyclic graph of immediate successors as shown in Figure 1.

The composition of views that are installed by VSC should have some bearing to the state of processes and communication links. Properties on view composition for VSC can be formally defined in terms of *reachability suspectors* [5], but for our purposes, it suffices to require that no process installs a view to which it does not belong.

Property 3.2 (Self Inclusion) *Each process p installs only views that contain itself. In other words,*

$$\forall p, v : (vchg_p(v) \in h_p) \Rightarrow (p \in comp(v)).$$

3.2 Message Delivery

Processes of the group communicate with each other through the reliable multicast primitive $mcast(m)$. Ideally, VSC should guarantee that each multicast message is delivered either by all or none of the group members. To render this idea feasible in an asynchronous system, VSC integrates message delivery guarantees with view installations. To guarantee liveness, message delivery semantics can be defined only with respect to successive view installations.

Property 3.3 (Intersection) *Given a process p and view v, let M_p^v denote the set of messages delivered by p in view v. In other words, $M_p^v = \{m : dlvr_p(m) \in h_p \wedge view(dlvr_p(m)) = v\}$. All processes that survive from one view v to the same next view w must have delivered the same set of messages in view v:*

$$\forall v, w : (v \prec w) \Rightarrow (\forall p, q \in comp(v) \cap comp(w) : \ M_p^v = M_q^v).$$

Note that for a given view v, there may be several successor views. The condition must hold for each such pair of views. Next, we require that a given multicast message be delivered (by those processes that deliver it) in a single view.

Property 3.4 (Uniqueness) *No message is delivered in more than one view:*

$$\forall m : (dlvr_p(m) \in h_p) \wedge (dlvr_q(m) \in h_q) \Rightarrow (view(dlvr_p(m)) = view(dlvr_q(m))).$$

Finally, to avoid certain degenerate solutions, we require that each message m be delivered by any given process at most once, and only if some process actually multicast m. Furthermore, correct processes always deliver their own multicasts.

Property 3.5 (Integrity)

$$\forall p, (\exists e_p^k \in h_p : e_p^k = dlvr_p(m)) \Rightarrow \nexists e_p^i \in h_p, i \neq k : e_p^i = dlvr_p(m),$$
$$\forall p, (\exists m : dlvr_p(m) \in h_p) \Rightarrow \exists q : mcast_q(m) \in h_q,$$
$$\forall p, (\exists e_p^k \in h_p : e_p^k = mcast_p(m)) \Rightarrow \exists e_p^i \in h_p, i > k : (e_p^i = dlvr_p(m) \vee e_p^i = crash_p).$$

A VSC service can now be defined in terms of the above properties for view installations and message deliveries.

Definition 3.3 (View-Synchronous Communication (VSC)) *A communication infrastructure is said to be* view synchronous *if it satisfies the Partial Order, Self Inclusion, Intersection, Uniqueness and Integrity properties for view installation and message delivery.*

4 Necessary Conditions for Strong-Partial VSC

View-synchronous communication with strong-partial semantics can be seen as a refinement of VSC that adds the following requirement to those of Definition 3.3:

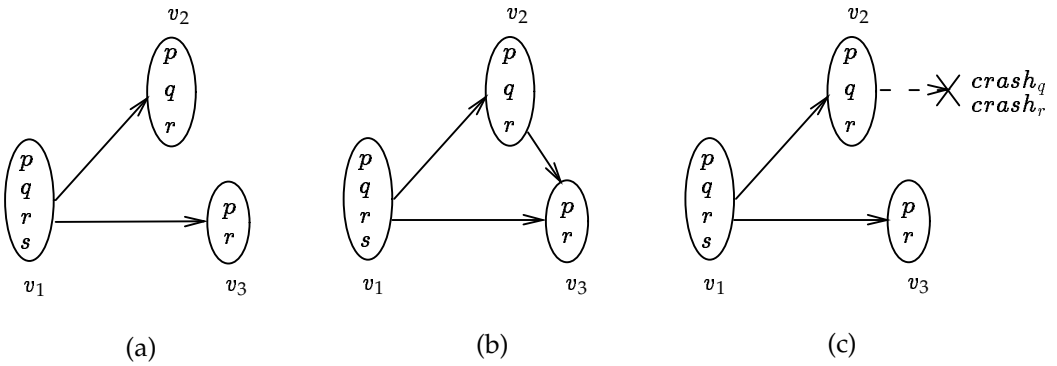

Figure 1. (a) Process p does not install view v_2 to which it belongs. (b) A possible extension of the execution restoring the Disjoint Concurrent Views property. (c) A possible extension of the execution for which the Disjoint Concurrent Views property cannot be restored.

Property 4.1 (Disjoint Concurrent Views)*Given any two views v and w,*

$$(v \not\prec^* w) \wedge (w \not\prec^* v) \Rightarrow (comp(v) \cap comp(w) = \emptyset).$$

In other words, for VSC to have strong-partial semantics, concurrent views must be such that their compositions are disjoint. Inclusion of this property to the definition has some subtle implications on the feasibility of VSC. We illustrate them through the following examples. First, consider the partial execution depicted in Figure 1(a). Let the prefixes of the histories for processes p, q and r be as follows:

$$h_p = vchg(v_1)\ vchg(v_3)$$
$$h_q = vchg(v_1)\ vchg(v_2)$$
$$h_r = vchg(v_1)\ vchg(v_2)$$

Note that up to this point, process p has not installed view v_2 even though $p \in comp(v_2)$. Process p has instead installed view v_3 which contains itself and process r. The question is if this execution satisfies Property 4.1 or not, but cannot be answered without knowing the future. For example, if process r extends its history to be $h_r = vchg(v_1)\ vchg(v_2)\ vchg(v_3)$ by installing view v_3 as shown in Figure 1(b), then the execution satisfies Property 4.1 since none of the three views are concurrent. If, however, the execution is extended by processes q and r crashing as shown in Figure 1(c), then views v_2 and v_3 will forever remain concurrent and no possible extension of the histories can restore Property 4.1. Since process p, at the time of installing view v_3, cannot know which of the two futures will occur, its only safe action is *not* to install any view and to terminate. This is clearly unreasonable since it leads to a system that does nothing to guarantee that a safety property is not violated.

It is easy to see that the above problem arises whenever the execution is such that there exists some installed view that is not installed by all of the correct processes in its composition. Thus, we obtain the first necessary condition for the solvability of strong-partial VSC.

Condition 4.1 (Closure of View Installation)*Any group membership algorithm that is live and achieves Property 4.1 must satisfy the following condition:*

$$\forall p, v : (vchg_p(v) \in h_p) \Rightarrow \forall q \in comp(v) : (vchg_q(v) \in h_q) \vee (crash_q \in h_q).$$

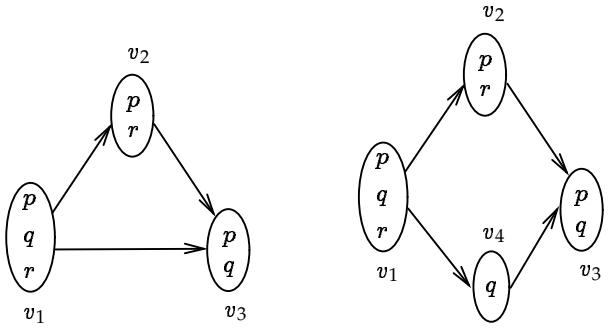

Figure 2. (a) Execution that satisfies Closure of View Installation but not the message delivery semantics for VSC. (b) Modified execution that satisfies strong-partial VSC.

While Condition 4.1 is necessary for strong-partial VSC, it is not sufficient. Consider the example depicted in Figure 2(a). The prefixes of the process histories are as follows:

$$h_p = vchg(v_1)\ vchg(v_2)\ ?\ vchg(v_3)$$
$$h_q = vchg(v_1)\ mcast(m)\ dlvr(m)\ vchg(v_3)$$
$$h_r = vchg(v_1)\ vchg(v_2)$$

A simple verification shows that Condition 4.1 is satisfied in that each view is installed by all of the processes in its composition and the execution satisfies Property 4.1 for strong-partial. The problem, however, is that execution cannot satisfy VSC with respect to message delivery semantics.

Consider the prefix of process p's history up to the question mark. Before p installs view v_3, we are faced with the question of which messages to deliver, if any. Note that with respect to successor views v_1 and v_2, processes p and r are the only ones to survive between them. Both p and r deliver no messages in view v_1, thus Properties 3.3 and 3.4 are both satisfied. For view v_3, there are two different immediate predecessor views: v_1 and v_2. Process q has delivered message m in view v_1, obliging p to deliver it as well since $comp(v_1) \cap comp(v_3) = \{p, q\}$. Thus, to satisfy Property 3.3, p must deliver m before installing v_3. Doing so, however, violates Property 3.4 since m is delivered by q in view v_1 and by p in view v_2. It is easy to see that no matter how the question is resolved, it is impossible to satisfy both Properties 3.3 and 3.4.

The scenario of the above example can be easily generalized to executions where the resulting partial order on views contains a "triangle" pattern — existence of two views that are related both directly and indirectly through the successor relation $\prec$. Thus, we obtain the following result as a second necessary condition for the solvability of strong-partial VSC.

Condition 4.2 (Separation of Paths) *Any service that implements strong-partial VSC must be based on a group membership component that guarantees the following condition:*

$$\forall v, w : (v \prec w) \Rightarrow \nexists u : (v \prec^* u) \wedge (u \prec^* w).$$

Figure 2(b) indicates one possible modification of the execution depicted in (a) such that Condition 4.2 is satisfied. For this execution, history of process q is

$$h_q = vchg(v_1)\ mcast(m)\ dlvr(m)\ vchg(v_4)\ vchg(v_3)$$

where it installs view v_4 before installing v_3. Clearly, Condition 4.2 is also sufficient to guarantee that concurrent views are disjoint as required by strong-partial semantics.

5 The SP-VSC Problem

In light of the results from the previous Section, we define an abstract problem, called SP-VSC, that characterizes the complexity of the desired service. Conceptually, an instance of SP-VSC has to be solved every time a new view has to be installed. This action is typically triggered by a notification from the underlying failure suspector signaling a change in the group membership. Given that VSC guarantees message delivery semantics only between two consecutive views, during long failure-free periods, applications cannot reason globally about the set of messages delivered by the other group members. In such situations, it might be appropriate to force a new view installation even when no failures are detected simply to "flush out" pending multicast deliveries and achieve agreement on the message sets.

Note that the reliable multicast and message delivery during periods where there are no view changes are not of interest towards the definition of SP-VSC. They can be implemented through "best-effort" algorithms that do not contribute additional complexity to the problem.

Our definition of SP-VSC as an abstract problem follows closely the formulation for the *primary-partition VSC problem* (PP-VSC) by Schiper and Sandoz [29]. Let v be the current view along the cut of system execution on which SP-VSC is defined. From each process p in the composition of v, SP-VSC takes as input the set of messages p has already delivered in view v. With this input, the goal of SP-VSC is then to output, at each process that has not crashed, the composition of the next view w and the set of additional messages that have to be delivered in view v before installing w. Formally,

SP-VSC Input: Given a cut c with current view v, for each process $p \in comp(v)$, input to SP-VSC is denoted $I\text{–}Msg_p$ and consists of the set of messages already delivered by p in view v on or before c. In other words, $I\text{–}Msg_p = \{m : (dlvr_p(m) \in h_p(c)) \wedge (view(dlvr_p(m)) = v)\}$ where $h_p(c)$ denotes the prefix of process p's history at cut c.

SP-VSC Output: At each process $p \in comp(v)$ that has not crashed, SP-VSC outputs $O\text{–}Msg_p$ and $O\text{–}Proc_p$ denoting the set of messages to be delivered in view v and the composition of the next view w such that $v \prec_p w$ and $comp(w) = O\text{–}Proc_p$.

The contents of the message set $O\text{–}Msg_p$ may be constructed incrementally. The outputting of $O\text{–}Msg_p$ at each process p, however, occurs as a single action. Let $POUT_p(c)$ be a predicate that is true on cut c if and only if process p has been output the set $O\text{–}Proc_p$ by cut c. Similarly, predicate $MOUT_p(c)$ is true on cut c if and only if process p has been output the set $O\text{–}Msg_p$ by cut c.

The following seven properties formally define a solution to the SP-VSC problem. The first property states that the algorithm eventually "terminates" at each process that has not crashed by outputting the new view composition. The second property requires that the set of messages to be delivered in view v is output before the composition of the new view. The third property enforces the new view composition to include a process at which it is output and states that view composition should not change during failure-free executions. In all of the following properties, free variables are assumed to be universally quantified.

SP-VSC1. Termination: $\exists c : POUT_p(c) \vee (crash_p \in h_p(c))$.

SP-VSC2. Order: $POUT_p(c) \Rightarrow MOUT_p(c)$.

SP-VSC3. Proc-Validity: $POUT_p(c) \Rightarrow (p \in O\text{–}Proc_p)$. Furthermore, in the absence of failures, $O\text{–}Proc_p = comp(v)$.

The next two properties relate the messages output by SP-VSC to the input. In particular, each message that is output must have been input by some process. Furthermore, a process cannot "undeliver" a message that it had already delivered when the problem was defined.

SP-VSC4. Msg-Validity 1: $MOUT_p(c) \Rightarrow O\text{–}Msg_p \subseteq \bigcup_{q \in comp(v)} I\text{–}Msg_q$.

SP-VSC5. Msg-Validity 2: $MOUT_p(c) \Rightarrow I\text{–}Msg_p \subseteq O\text{–}Msg_p$.

The final two properties are agreement requirements on the set of messages to be delivered and the composition of the next view. At all processes that survive from the initial view into the same next view, SP-VSC must output the same message set. Any pair of processes in the initial view install next views that are either identical or disjoint in their composition.

SP-VSC6. Msg-Agreement:

$$POUT_p(c) \Rightarrow \forall q \in O\text{–}Proc_p : (POUT_q(c) \Rightarrow (O\text{–}Msg_q = O\text{–}Msg_p)).$$

SP-VSC7. Proc-Agreement:

$$(POUT_p(c) \wedge POUT_q(c)) \Rightarrow (O\text{–}Proc_p = O\text{–}Proc_q) \vee (O\text{–}Proc_p \cap O\text{–}Proc_q = \emptyset).$$

Since our lower-bound results will be based on SP-VSC, it is essential that the above formulation for the problem correctly characterize the inherent complexity of implementing a view-synchronous communication service with strong-partial semantics. This is indeed the case as shown in the following result.

Lemma 5.1 (SP-VSC and VSC Service Equivalence) *SP-VSC and view-synchronous communication service with strong-partial semantics are equivalent problems in that given a solution for one, we can obtain a solution for the other.*

PROOF SKETCH: For sake of brevity, we only outline how a solution for SP-VSC can be used to implement view-synchronous communication service with strong-partial semantics. The multicast of a message through $mcast_p(m)$ is handled by any "best-effort" algorithm as long as the delivery of m is ensured at p. Notifications from an underlying failure suspector mechanism (for example based on periodic "pinging" and timeouts) cause an invocation of the SP-VSC algorithm in order to terminate the current view. When the algorithm terminates at process p, for each message $m \in O\text{–}Msg_p$, process p executes a $dlvr_p(m)$ event, unless it has already done so, and then it executes a view change event $vchg_p(v)$ installing view v where $comp(v) = O\text{–}Proc_p$. Note that multiple instances of the SP-VSC algorithm may be invoked concurrently, for example, in case of merging partitions after repairs. Conceptually, each instance of the SP-VSC algorithm terminates the view in which it was invoked in and a separate agreement algorithm is executed for establishing the composition of the final view representing the merger. In practice, these two separate algorithms can be combined into a single one such that, when invoked in view u, it installs *two* back-to-back views v and w where v is specified by the SP-VSC requirements terminating view u, and w is the potential union of merged views (that must correspond to concurrent instance of the SP-VSC algorithm) with the guarantee that no messages are delivered in view v. Details are beyond the scope of this paper and can be found in [5]. It is easy to easy to show that this construction along with the properties of the SP-VSC algorithm indeed satisfy those required for strong-partial VSC. □

6 Reduction of Atomic Commitment to SP-VSC

We now show that the Atomic Commitment Problem (ACP) reduces to SP-VSC. In other words, any algorithm that solves SP-VSC as defined in the previous Section can be transformed to solve also ACP.

ACP arises in a distributed database context where the actions of subtransactions need to be coordinated to preserve the consistency of the data despite failures. An instance of ACP needs to be solved whenever a transaction completes. Informally, a solution to ACP must guarantee that either all or none of the processes that participated in the transaction make their changes to the data permanent.

Let T denote the set of processes that executed actions on behalf of the transaction. The input to ACP for each process $p \in T$ is a $vote_p \in \{YES, NO\}$ indicating the willing and ableness of p in making its changes permanent. A process that supplies an input value to ACP is said to *vote.* The output of ACP at each process p that has not crashed is a $decision_p \in \{COMMIT, ABORT\}$ that conveys the action to be taken. A process at which ACP outputs a value is said to *decide.* Formally, ACP can be specified through the following properties [7]:

ACP1: All processes that decide reach the same decision.

ACP2: If any process decides *COMMIT*, then all participants must have voted *YES*.

ACP3: If all processes vote *YES* and no failures occur, then all processes decide *COMMIT*.

ACP4: Each process decides at most once (that is, a decision is irreversible).

The transformation of an algorithm that solves SP-VSC into one that solves ACP consists of establishing mappings inputs and outputs of the two problems. In particular, we need to show how the input for SP-VSC can be obtained from the input for ACP and how the output of ACP can be obtained from the outputs of SP-VSC. One such transformation is shown in Figure 3.

Input: At each $p \in T$:
$I\text{–}Msg_p = (p, vote_p)$

Invoke SP-VSC Algorithm

Output: At process p, when SP-VSC outputs $O\text{–}Proc_p$ and $O\text{–}Msg_p$:
if $((O\text{–}Proc_p = T) \wedge (\forall p \in T : (p, YES) \in O\text{–}Msg_p))$ **then**
$decision_p = COMMIT$
else
$decision_p = ABORT$

Figure 3. Transforming a solution for the SP-VSC problem into one for ACP.

Theorem 6.1 (ACP to SP-VSC Reduction)*The Atomic Commitment Problem reduces to SP-VSC.*

PROOF: We show that given an algorithm guaranteeing Properties SP-VSC1 through SP-VSC7 of Section 5, the transformation outlined in Figure 3 indeed achieves Properties ACP1–ACP4 of the atomic commitment problem. We proceed by cases.

ACP1. For contradiction, assume that for two correct processes $p \neq q$, the transformation results in $decide_p = COMMIT$ and $decide_q = ABORT$. For p to decide $COMMIT$ it must be that $O\text{–}Proc_p = T$ and $O\text{–}Msg_p$ contains YES

votes from all processes in T. For q to decide $ABORT$, either $O\text{–}Proc_q \neq T$, or $O\text{–}Msg_q$ does not contain a vote for all processes in T, or it contains at least one NO vote. By SP-VSC3, $q \in O\text{–}Proc_q$ and by hypothesis, $q \in O\text{–}Proc_p$. Thus by SP-VSC7 it must be that $O\text{–}Proc_q = O\text{–}Proc_p$ since the intersection of the two sets $O\text{–}Proc_q$ and $O\text{–}Proc_p$ has at least q in common. So, it is not the case that $O\text{–}Proc_q \neq T$. By SP-VSC6, it must also be the case that $O\text{–}Msg_q = O\text{–}Msg_p$, contradicting the hypothesis.

ACP2. For contradiction, assume that for some correct process p the transformation results in $decide_p = COMMIT$ even though some process q had $vote_q = NO$. By construction, $I\text{–}Msg_q = (q, NO)$. Since p decides $COMMIT$, $O\text{–}Msg_p$ must contain YES messages from all processes in T, including q. Thus, $(q, YES) \in O\text{–}Msg_p$ and by SP-VSC4, $(q, YES) \in \bigcup_{p \in T} I\text{–}Msg_p$. In other words, there must exist some process r such that $(q, YES) \in I\text{–}Msg_r$. Since each input set $I\text{–}Msg_p$ is constructed to contain only the vote of p, and by SP-VSC5, it must be that $r = q$. This is a contradiction since by assumption, q had voted NO.

ACP3. Assume that $vote_p = YES$ for all process $p \in T$ and there are no failures during the execution of the algorithm solving SP-VSC. In other words, for all $p \in T$, we have $I\text{–}Msg_p = (p, YES)$. SP-VSC3 guarantees that $O\text{–}Proc_p = T$ for all $p \in T$ and that $p \in O\text{–}Proc_p$. Thus, by SP-VSC6 and SP-VSC7, respectively, we have $O\text{–}Msg_q = O\text{–}Msg_p$ and $O\text{–}Proc_p = O\text{–}Proc_q = T$ for all processes $p, q \in T$. SP-VSC4 and SP-VSC5 together require that $O\text{–}Msg_p = \bigcup_{q \in T} I\text{–}Msg_q$ for all $p \in T$. These define exactly the conditions for deciding $COMMIT$ at each process.

ACP4. From the structure of the transformation, it is clear that once process p assigns to its variable $decide_p$, it cannot modify it further. Thus, every process decides at most once. □

An important property of fault-tolerant distributed algorithms has to do with the ability of correct processes to make progress in the presence of failures in other components of the system. If an algorithm is such that correct processes are guaranteed to terminate their operations despite any possible failure scenario, it is called *nonblocking* [13]. An immediate consequence of the reduction in Theorem 6.1 is the following impossibility result.

Corollary 6.1 (Impossibility of Nonblocking SP-VSC)*In a system where communication or total process failures are possible, there exists no nonblocking algorithm that solves the SP-VSC problem.*

PROOF: As we have defined it, SP-VSC includes the Termination Property requiring each process p to either output both sets $O\text{–}Proc_p$ and $O\text{–}Msg_p$ or to crash. Thus, the transformation of Figure 3 would result in an algorithm that satisfies not only Properties ACP1–ACP4, but also the following

ACP5. Every correct process eventually decides

making it a nonblocking solution for ACP. It is well known that in systems that admit either the possibility of communication failures or the total failure of process, nonblocking solutions for ACP do not exist. □

The *size* of the SP-VSC problem defined on view v is established by the cardinality of the set $comp(v)$. In an asynchronous system, the number of tandem end-to-end message delays is the only reasonable measure for time complexity. The following lower-bounds on the time and message complexity of SP-VSC are immediate consequences of Theorem 6.1 and the corresponding results for ACP[2] [13].

Corollary 6.2 (Message Lower Bound)*Any SP-VSC algorithm (blocking or non-*

[2] The time complexity results are based on a "linear communication" model where a process can communicate with at most one other process in each time step.

blocking) of size n requires at least $2(n-1)$ messages in the absence of processor failures.

Corollary 6.3 (Time Lower Bound) *Any SP-VSC algorithm of size n requires at least $\log n$ time.*

Corollary 6.4 (Nonblocking Time Lower Bound) *Any nonblocking SP-VSC algorithm of size n requires at least $2\log n - 3\log\log n - O(1)$ time.*

7 Related Work

In [27] Schiper and Ricciardi describe how strong-partial group membership semantics can be implemented based on weak-partial semantics through the use of "process signatures." The idea is to extend process names by including a component that is derived from the identifier of the view in which they appear. In other words, processes are renamed at every view change. While, from an external observer's point of view, concurrent views are indeed disjoint with respect to these extended names, the technique is not effective for use from within the system: a process that installs a view v can never be sure which of the other processes in $comp(v)$ share v with it.

The relationship between view synchrony and atomic commitment has been studied in several other works. In [28] Schiper and Sandoz show how uniform view-synchronous multicast with a linear membership service can be used to implement atomic commitment. In [19] Guerraoui and Schiper show how both view-synchronous communication and atomic commitment can be implemented from a single primitive called *dynamic terminating multicast*. Inherent cost results for view-synchronous communication are obtained in [29] for the linear membership semantics relating it to the consensus problem.

8 Discussion

To better appreciate the results of Section 6, it is useful to consider upper bound costs in terms of concrete algorithms for ACP. Two-Phase Commit (2PC) and Three-Phase Commit (3PC) are canonical algorithms for ACP representing the blocking and nonblocking classes, respectively [9]. In the presence of communication failures, 3PC can be made nonblocking in one of the partitions; in all others it will block. In their "coordinator-based" versions, 2PC and 3PC require 3 and 5 tandem end-to-end message delays, respectively, in failure-free executions. We feel that, even if blocking were not an issue, these time costs may be unacceptable for view termination in large-scale distributed systems. Precisely the characteristics of such systems — large end-to-end message delays and frequent view changes due to virtual or real partitions — may render VSC with strong-partial semantics too expensive. It is for this reason that in the Relacs system, we implement *quasi-strong* semantics where concurrent views may overlap as long as they are proper subsets of each other. Not only is this semantics useful for application development, it can be achieved at a cost of 2 tandem end-to-end message delays without blocking even in the presence of communication failures [5].

Acknowledgments We are grateful to Ian Jacobs, Mary Baker and Niels Nes who have contributed at various phases to the Relacs project. 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 Ministry of University, Research and Technology.

References

[1] Y. Amir, D. Dolev, S. Kramer and D. Malki. Membership Algorithms in Broadcast Domains. In *Proc. 6th Intl. Workshop on Distributed Algorithms*, A. Segall and S. Zacks (Eds.), Haifa, Israel, Lecture Notes in Computer Science, vol. 647, Springer-Verlag, November 1992, 292–312.

[2] Y. Amir, D. Dolev, S. Kramer and D. Malki. Transis: A Communication Sub-System for High Availability. In *Proc. 22nd Annual International Symposium on Fault-Tolerant Computing Systems*, July 1992, 76–84.

[3] Ö. Babaoğlu and A. Schiper. On Group Communication in Large-Scale Distributed Systems. In *Proc. ACM SIGOPS European Workshop*, Dagstuhl, Germany, September 1994. Also appears as *ACM SIGOPS Operating Systems Review*, 29(1):62–67, January 1995.

[4] Ö. Babaoğlu, A. Bartoli and G. Dini. Replicated File Management in Large-Scale Distributed Systems. In *Proc. 8th Int. Workshop on Distributed Algorithms*, G. Tel and P. Vitányi (Eds.), Lecture Notes in Computer Science, vol. 857, Springer-Verlag, Berlin, September 1994, 1–16.

[5] Ö. Babaoğlu, R. Davoli and A. Montresor. Efficient Algorithms for Group Membership and View-Synchronous Communication in the Presence of Partitions. Technical Report, Department of Computer Science, University of Bologna, April 1995.

[6] Ö. Babaoğlu, R. Davoli, L.A. Giachini and M.G. Baker. Relacs: A Communication Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems. In *Proc. 28th Hawaii International Conference on System Sciences*, Maui, January 1995, vol. II, 612–621.

[7] Ö. Babaoğlu and S. Toueg. *Non-Blocking Atomic Commitment* In *Distributed Systems*, Sape J. Mullender (Ed.), Addison-Wesley-ACM Press, New York, 1993, 147–168.

[8] K.P. Birman and T.A. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *Proc. 11th ACM Symposium on Operating Systems Principles*, 1987, 123–138.

[9] P.A. Bernstein, V. Hadzilacos and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, Massachusetts, 1987.

[10] K. Birman, The Process Group Approach to Reliable Distributed Computing, *Communication of the ACM*, 9(12):36–53, December 1993.

[11] T.D. Chandra and S. Toueg. Unreliable Failure Detectors for Asynchronous Systems. In *Proc. 10th ACM Symposium on Principles of Distributed Computing*, August 1991, 325–340.

[12] D.R. Cheriton and W. Zwaenepoel. Distributed Process Groups in the V Kernel. *ACM Trans. Computer Systems*. 3(2):77–107, May 1985.

[13] C. Dwork and D. Skeen. The Inherent Cost of Nonblocking Commitment. In *Proc. 2nd ACM Symposium on Principles of Distributed Computing*, Montreal, Canada, August 1983, 1–11.

[14] P.E. Ezhilchelvan, R.A. Macedo and S.K. Shrivastava. Newtop: A Fault-Tolerant Group Communication Protocol. Technical Report, Computer Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne, United Kingdom, August 1994.

[15] P. Felber, C. Malloth, A. Schiper and U. Wilhelm. Phoenix: A Group-Oriented Infrastructure for Large-Scale Distributed Systems. Technical Report, EPFL-LSE, Lausanne, Switzerland.

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

[17] R. Friedman and R. van Renesse. Strong and Weak Virtual Synchrony in Horus. Technical Report TR95-1491, Department of Computer Science, Cornell University, Ithaca, New York, March 1995.

[18] J.N. Gray. Notes on Database Operating Systems. In *Operating Systems: An Advanced Course*, R. Bayer, R.M. Graham and G. Seegmuller (Eds.), Lecture Notes in Computer Science, vol. 60, Springer-Verlag, 1978.

[19] R. Guerraoui and A. Schiper. Transaction model vs. Virtual Synchrony model: bridging the gap. To appear in *Distributed Systems: From Theory to Practice*, K. Birman, F. Cristian, F. Mattern, A. Schiper (Eds.), Springer Verlag, LNCS, 1995.

[20] V. Hadzilacos. On the Relationship Between the Atomic Commitment and Consensus Problems. In *Fault-Tolerant Distributed Computing*, B. Simons and A. Z. Spector (Eds.), Lecture Notes in Computer Science, vol. 448, Springer-Verlag, New York, 1990, 201–208.

[21] F. Jahanian and W.M. Morgan. Strong, Weak and Hbrid Group Membership. In *Proc. 2nd IEEE Workshop on the Management of Replicated Data*, November 1992, 34–38.

[22] M.F. Kaashoek and A.S. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proc. 11th International Conference on Distributed Computer Systems*, IEEE Computer Society Press, Arlington, Texas, May 1991, 222–230.

[23] L.L. Peterson, N.C. Bucholz, and R.D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

[24] R. van Renesse, K. Birman, R. Cooper, B. Glade and P. Stephenson. The Horus System. In *Reliable Distributed Computing with the Isis Toolkit*, K.P. Birman, R. van Renesse (Ed.), IEEE Computer Society Press, Los Alamitos, CA, 1993, 133–147.

[25] A. Ricciardi and K. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *Proc. 10th ACM Symposium on Principles of Distributed Computing*, August 1991, 341–351.

[26] A. Ricciardi, A. Schiper and K. Birman. Understanding Partitions and the "No Partition" Assumption. In *Proc. 4th IEEE Workshop on Future Trends of Distributed Systems*, Lisboa, September 1993.

[27] A. Schiper and A. Ricciardi. Virtually-Synchronous Communication Based on a Weak Failure Suspector. In *Proc. 23rd International Symposium on Fault-Tolerant Computing Systems*, Toulouse, France, June 1993, 534–543.

[28] A. Schiper and A. Sandoz. Uniform Reliable Multicast in a Virtually Synchronous Environment. In *Proc. 13th International Conference on Distributed Computing Systems*, May 1993, 501–568.

[29] A. Schiper and A. Sandoz. Primary Partition "Virtually-Synchronous Communication" Harder than Consensus. In *Proc. 8th Int. Workshop on Distributed Algorithms*, G. Tel and P. Vitányi (Eds.), Lecture Notes in Computer Science, vol. 857, Springer-Verlag, Berlin, September 1994, 38–52.