View Synchronous Communication in Large Scale Networks*

Christoph Malloth André Schiper
Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
CH - 1015 Lausannne (Switzerland)
{malloth,schiper}@lse.epfl.ch

Abstract

View synchronous communication (VSC) is a paradigm initially proposed by the Isis system, that is well suited to implement fault-tolerant services based on replication. VSC can be seen as an adequate low level semantics on which ordered multicasts and uniform multicasts can easily be implemented. This paper presents the specific problems related to the implementation of VSC in a large scale network (e.g. Internet). The paper also shows how these problems are solved within Phoenix, a group oriented platform under development. Specifically the Phoenix implementation of VSC allows progress in cases where traditional solutions would block forever.

1 Introduction

Distributed systems are commonplace in the domain of local area networks. On the other hand with emerging world-wide interconnection of computers, e.g. using the Internet, large scale distributed systems become more and more important. Some examples are: a cooperative editing systems that allow different sites throughout the world to work on the same document, an air traffic control systems supervising a large territory, etc. An important requirement for a lot of distributed applications is fault tolerance, i.e. insurance of availability and consistency despite failures. Availability is obtained by replicating a service on different sites within the distributed system and consistency is obtained by using adequate communication protocols between processes. One well known paradigm for guaranteeing consistent communication in the presence of failures is the *virtually synchronous communication* paradigm, or *VSC* paradigm, initially proposed by the Isis system [3]. In the following we prefer to use the acronym VSC for *view synchronous communication*, as this term fits better the paradigm described. VSC is based on *views* defined for every group g of processes. Members of a *view* agree on: (1) the sequence of *views* (i.e. views are totally ordered [12]), and (2) multicasts issued to the group are totally ordered with respect to view changes [4]. This can be perceived as a low level semantics, but it is adequate to implement higher level communication primitives, such as totally ordered multicasts [4] (the Isis ABCAST), or uniform multicasts [14]. VSC defines thus the basic layer of an Isis-like environment.

We are currently developing *Phoenix*, an Isis-like environment, that will run on large scale networks, like the Internet. Phoenix has several advantages, in large scale as well as in local

*Research supported by the "Fonds national suisse" and OFES under contract number 21-32210.91, as part of the ESPRIT Basic Research Project BROADCAST (number 6360)

area networks, over systems like Isis. Phoenix defines some unique features, such as an uniform multicast primitive [14], two compatible totally ordered multicast primitives, a *weak* totally ordered multicast, and a *strong* totally ordered multicast [16], and a refinement of the notion of group, by distinguishing, for every group, between *core* members, *client* members, and *sink* members [2] in order to be scalable for large scale.

The real challenge in building Phoenix is the implementation of VSC on a large scale network such as the Internet. This poses two very specific problems: (1) network partitions are not unlikely to occur, and (2) transitivity of communication is not ensured all the time, due to possible inadequate routing information (e.g. in the Internet at some time t site s_1 can be able to communicate with site s_2, s_2 with s_3, but s_1 might be unable to communicate with s_3). The partition problem is discussed in Sections 2 and 4. The system model and some basic notations are introduced in Section 3. The non-transitivity problem is discussed in Section 5. Section 6 concludes with some final remarks.

2 Partition failures and the blocking problem

Partitions present a challenging problem when implementing VSC, in that partitions can lead to blocking of the system. One approach for handling partitions within VSC is through the so-called *primary partition* model [11]. For each group, the primary partition is composed of the privileged subset of processes, within which progress is possible despite partitions. Unfortunately, link failures may occur such that no primary partition exists for a certain group, and all activity for this group will block. When such a scenario occurs, we require that the system be able to resume activity once a sufficient number of link failures have been repaired. This does not occur in a system like Isis: once blocking has occurred in Isis, blocking holds forever [12]. In systems like Transis [1], minority views can continue execution and after reparation of partition, views are merged together and all messages not yet delivered by another view are transmitted to the corresponding view. Unfortunately, this implies that every partition has to keep track of all messages sent in its current view, to be used upon merging. If partition lasts for a long time, this can consume a lot of memory. Horus ([15] avoided the blocking problem by allowing progress in minority views without storing messages but by guaranteeing a primary partition. In Horus, minority views are views that have lost their *primary partition flag* and as a result, at some point of execution, it is possible that none of the views has this flag set. In this case, the primary partition is lost forever. As the view change protocol in Phoenix is based on the Chandra/Toueg consensus protocol [5], Phoenix ensures the existence of the primary partition and its progress whenever theoretically possible. This is ensured neither by Transis nor by Horus as none of these systems are based on the Chandra/Toueg consensus protocol and its weak failure suspector.

The blocking problem for the primary partition property is related to the failure suspicion model. There are basically two models that might be considered:

- *Stable suspicion model.* In this model, once a suspicion holds, it holds forever. Thus, once process p_i suspects process p_j to have crashed, it cannot later change its mind. An incorrectly suspected process can not come back anymore and has to simulate a crash failure in order to receive a new identity;
- *Unstable suspicion model.* In this model a process can always change its mind: p_i might suspect p_j to have crashed, and later change its mind to consider p_j alive.

The stable suspicion model (adopted by Isis), is adequate in a setting where incorrect failure suspicions are not too frequent. This condition is, however, not met in large scale networks where link failures are likely to occur: a transient link failure between processes p_i and p_j will almost inevitably lead to incorrect suspicions (p_i will incorrectly suspect p_j, p_j will incorrectly suspect p_i). The stable suspicion model has the further disadvantage that, once every process suspects a majority of processes, no new primary partition can ever be defined. To overcome the blocking problem, Phoenix adopts a mixed model in which a suspicion is stabilized if and only if the VSC layer is able to define a new primary partition excluding the suspected processes. This is discussed further in Section 4.

3 System model and basic definitions

The distributed system is composed of a finite set $\mathsf{S} = \{p_1, \ldots, p_n\}$ of processes, connected through a set of communication links $\mathsf{L}= \{l_{ij}\}$. Communication is realized by message passing, is asynchronous (there is no bound on the transmission delays), and reliable[1]. Processes fail by crashing (we do not consider Byzantine failures). Crashed processes can recover, but a stably stored incarnation number, which is incremented each time the process restarts, gives other processes the possibility to recognize the crash. A local module, called *failure suspector* FS_i, is attached to every process p_i. The failure suspector FS_i maintains a set of processes Susp_i that it currently suspects to be unreachable (either p_j has crashed, or the link l_{ij} is currently down). This failure suspicion information is forwarded to the process p_i. We introduce the following notation[2]:

$$\mathsf{Susp}_i(c) = \{p_j \mid p_j \in \mathsf{S} \quad \wedge \quad p_j \text{ is suspected by } FS_i \text{ on cut } c\}$$

A failure suspector can make mistakes by incorrectly suspecting a process. Suspicions are not stable, thus if at a given instant FS_i suspects p_j, it can later learn that the suspicion was incorrect and FS_i remove p_j from Susp_i. We are not presently concerned how suspicions are generated, but come back to this issue in Section 5.

We define the $\mathsf{CommSet}_i(c)$ of a process p_i as the set of processes, p_i thinks it can communicate with on the cut c. This is the set of processes that FS_i does not currently suspect:

$$\mathsf{CommSet}_i(c) = \mathsf{S} - \mathsf{Susp}_i(c)$$

Based on the $\mathsf{CommSet}_i(c)$, we can define on every cut c the following $\textsc{Comm}$ relation:

$$\textsc{Comm}(c, p_i, p_j) \quad \Leftrightarrow \quad p_j \in \mathsf{CommSet}_i(c)$$

We would ideally like $\textsc{Comm}$ to be transitive on every cut c, i.e.

$$\textsc{Comm}(c, p_i, p_j) \wedge \textsc{Comm}(c, p_j, p_k) \Rightarrow \textsc{Comm}(c, p_i, p_k).$$

[1] A reliable link ensures that a message sent by p_i to p_j is eventually received by p_j if p_i and p_j are correct (i.e. do not crash). This does not exclude link failures, if we require that any link failure is eventually repaired. A reliable link can be implemented by retransmitting lost or corrupted messages.

[2] We suppose that the reader is familiar with notion of *cuts*.

However, because of possible inadequate routing in large scale (e.g. the Internet), transitivity is not ensured all the time. This might lead to problems, as will be also shown in Section 5.

4 View Synchronous Communication (VSC)

4.1 Definition of VSC

As introduced in Section 1, the definition of view synchronous communication is based on *views* defined for every group of processes. Given a group g, a view is a set of correct processes as perceived by e.g. a membership service. We note $\mathsf{View}_k(g) = \{p_1, \dots, p_i\}$ as the k^{th} view of group g. The view of group g evolves as processes in g crash[3], processes join g, or processes leave g. Given a group g, VSC is defined informally by the following two properties:

1. processes in g agree on the sequence of views $\mathsf{View}_0(g)$, $\mathsf{View}_1(g)$, ..., $\mathsf{View}_k(g)$, ... (i.e. the views of g are totally ordered);
2. processes in g agree on the set of messages delivered[4] between each pair of view changes, i.e. they agree on the set of messages delivered between the delivery of $\mathsf{View}_k(g)$ and the delivery of $\mathsf{View}_{k+1}(g)$.

In order to simplify, we consider only one group, and use without ambiguity View_k instead of $\mathsf{View}_k(g)$. VSC can be defined considering one single view change, e.g. the view change from View_k to View_{k+1}. We introduce the following notations:

- $\mathsf{View}_{k,i}$ is the k^{th} view of group g delivered by p_i;
- the formula $\textsc{Deliv}_{k,i}$, defined on p_i, is true if and only if the process p_i has delivered $\mathsf{View}_{k,i}$.

Between the delivery of $\mathsf{View}_{k,i}$ and the delivery of $\mathsf{View}_{k+1,i}$, process p_i delivers a sequence of messages: *deliver(*$\mathsf{View}_{k,i}$*) ; deliver(m) ; deliver(m′) ; ... ; deliver(*$\mathsf{View}_{k+1,i}$*)*.
The set of messages $\{m, m', \dots\}$ delivered by p_i between the delivery of $\mathsf{View}_{k,i}$ and the delivery of $\mathsf{View}_{k+1,i}$ is noted $\mathsf{MsgSet}_{k,i}$. This set of messages is said to be *delivered by p_i in view k*.

Consider two processes p_i and p_j that have agreed on View_k, i.e. $\mathsf{View}_{k,i} = \mathsf{View}_{k,j}$. VSC can be formally defined by two agreement conditions: (1) agreement on the next view $k+1$, and (2) agreement on the set of messages delivered in view k.

(A1) Agreement on the next view. Let p_i and p_j agree on View_k (i.e. $\mathsf{View}_{k,i} = \mathsf{View}_{k,j}$). If p_i and p_j both deliver the next view, then they agree on this view:

$$\bigwedge_{p_i,p_j} \left(\textsc{Deliv}_{k+1,i} \wedge \textsc{Deliv}_{k+1,j} \;\; \Rightarrow \;\; \mathsf{View}_{k+1,i} = \mathsf{View}_{k+1,j} \right)$$

[3]More precisely, when processes are suspected to have crashed, which does not exclude incorrect suspicions.
[4]As usual, we distinguish between the *reception* of a message at the system level, and the *delivery* of a message at the application level.

(A2) Agreement on the set of messages. Let p_i and p_j agree on View_k (i.e. $\mathsf{View}_{k,i} = \mathsf{View}_{k,j}$). If p_i and p_j both deliver the next view, then they agree on the set of messages delivered in View_k:

$$\bigwedge_{p_i,p_j} \left(\textsc{Deliv}_{k+1,i} \wedge \textsc{Deliv}_{k+1,j} \quad \Rightarrow \quad \mathsf{MsgSet}_{k,i} = \mathsf{MsgSet}_{k,j} \right)$$

In order to avoid the trivial solution where the new view is either always the set S of all processes, or always the empty set, we need to add the following non-triviality condition:

(NT) Non-triviality. Crashed processes are eventually removed from a view, and new processes that want to join are eventually included in a view.

The two agreement conditions, together with the assumption that initially the processes in View_0 agree on View_0 (i.e. for every $p_i, p_j \in \mathsf{View}_0$, $\mathsf{View}_{0,i} = \mathsf{View}_{0,j} = \mathsf{View}_0$), lead the processes to an agreement on a sequence of views and on the set of messages delivered in each view. The non-triviality condition (NT) leads the sequence of views to approximate the set of correct processes.

Note also that the agreement condition (A2) naturally leads the delivery of messages to be ordered with respect to view changes: if p_i delivers a message m before the delivery of $\mathsf{View}_{k+1,i}$, then process p_j also delivers m before the delivery of $\mathsf{View}_{k+1,j}$ (otherwise the agreement condition (A2) is violated).

4.2 Consensus and failure suspector

Because of the agreement conditions (A1) and (A2) given in the previous section, VSC is a consensus-like problem, where consensus has to be reached on a set of messages and the next view. The consensus problem is defined as follows. Consider a set of processes S, where each process $p_i \in \mathsf{S}$ initially proposes a value v_i. The consensus problem consists in deciding on some value v such that the following three properties hold [5]:

Termination:	each correct process eventually takes a decision.
Agreement:	if two processes take a decision, they will take the same decision.
Validity:	if a process decides on v, then v was proposed by some process.

The consensus algorithm described in [5] is particularly interesting, as it solves consensus in an asynchronous environment extended with the failure suspector $\diamond\mathcal{W}$. The failure suspector added to the asynchronous environment allows to overcome the FLP impossibility result [10]. Moreover, [6] shows that $\diamond\mathcal{W}$ is the weakest failure suspector that allows to solve consensus in an asynchronous system with $f < n/2$, where f is the bound on the number of processes that may crash. The $\diamond\mathcal{W}$ failure suspector satisfies the following properties [5]:

Weak completeness Eventually every crashed process is permanently suspected by some correct process.

Eventual weak accuracy There is a time after which some correct process is not suspected by any correct process.

The remarkable thing about VSC is that it can be reduced to consensus, i.e. whenever consensus can be solved, VSC is also solvable[5]. To our knowledge, Phoenix is the first VSC system built on top of a consensus protocol, and the first VSC system to ensure progress whenever the properties of the failure suspector $\diamond\mathcal{W}$ are met.

4.3 Reduction of VSC to consensus

We show how VSC can be implemented, basing it on a solution to the consensus problem. Recall that $\mathsf{CommSet}_i(c)$ has been defined as the set of processes that p_i does not suspect on cut c. Consider the current view View_k. The VSC protocol is launched whenever the current view has to be changed, i.e. whenever there exists a cut c and a process $p_i \in \mathsf{View}_k$ such that $\mathsf{CommSet}_i(c) \neq \mathsf{View}_k$. More specifically, we consider the case $\mathsf{CommSet}_i(c) \subset \mathsf{View}_k$, i.e. one or more processes in View_k are suspected (process joins are considered in Section 4.4). The VSC protocol can be divided into two steps, where a consensus problem is solved in Step 2. During Step 1 every process p_i defines the initial value for the consensus problem solved in Step 2. Notice that Steps 1 and 2 have been separated for reasons of clarity. In our current implementation both steps are integrated (i.e. Step 1 is integrated within the consensus protocol). An example can be found in Section 4.5.

4.3.1 Step 1: initial value for the consensus

Let c be the cut on which the VSC protocol is launched, and let $\mathsf{MsgSet}_{k,i}(c)$ be the set of messages delivered by p_i in view k, on the cut c. Every process $p_i \in \mathsf{View}_k$ starts by multicasting $\mathsf{MsgSet}_{k,i}(c)$ to all processes in View_k, and waits to get the same information from the processes in View_k. Process p_i waits until there exists a cut c' that satisfies both of the following conditions:

1. on c', $\mathsf{CommSet}_i(c')$ is a majority of View_k: $|\mathsf{CommSet}_i(c')| > |\mathsf{View}_k|/2$;
2. if on c' process p_i has not received $\mathsf{MsgSet}_{k,j}$ from $p_j \in \mathsf{View}_k$, then $p_j \in \mathsf{Susp}_i(c')$.

Assuming that p_i does not crash, less than half of the other processes in View_k may crash, and a crashed process is eventually suspected by every correct process[6], then a cut c' satisfying both conditions eventually exists. Let $\mathsf{Rcv}_i(c') \subseteq \mathsf{View}_k$ be the set of processes p_j from which p_i has received $\mathsf{MsgSet}_{k,j}(c)$ on the cut c' (including p_i itself). On c', process p_i defines the initial value for the consensus problem of Step 2 as a pair (m_i, vw_i), where m_i is an estimate for the set of messages delivered in view k, and vw_i is an estimate for the next view $k+1$:

Estimate for the messages: The estimate for the set of messages is the union of the MsgSet received on the cut c':

$$m_i \stackrel{\text{def}}{=} \bigcup_{p_j \in \mathsf{Rcv}_i(c')} \mathsf{MsgSet}_{k,j}(c)$$

[5] This is not in contradiction with [13] where the result has led us to consider a different definition for VSC.

[6] The reader might notice that this assumption is stronger than the *weak completeness* property of $\diamond\mathcal{W}$. Our failure suspector guarantees however *strong completeness* (which can be implemented using a failure suspector guaranteeing *weak completeness*): a crashed process is eventually suspected by every correct process [5].

Estimate for the next view: The estimate for the next view is the $\mathsf{CommSet}$:

$$vw_i \stackrel{\text{def}}{=} \mathsf{CommSet}_i(c')$$

Notice that the initial value for the consensus satisfies the following *inclusion* property:

$$p_j \in vw_i \;\Rightarrow\; \mathsf{MsgSet}_{k,j}(c) \subseteq m_i \tag{1}$$

In other words, p_i's estimate is such that if p_j is proposed to be member of the next view, then every message delivered by p_j in view k is in the proposition for the set of messages to be delivered in view k.

4.3.2 Step 2: consensus

Once $p_i \in \mathsf{View}_k$ has defined its initial values (m_i, vw_i), it switches to the consensus problem. Step 2 solves the following consensus problem:

- the consensus is defined on the set of processes View_k;
- process $p_i \in \mathsf{View}_k$ proposes the initial value (m_i, vw_i) defined in Step 1;

The outcome of the consensus problem is the pair $(\mathsf{MsgSet}_k, \mathsf{View}_{k+1})$, where MsgSet_k is the set of messages delivered in View_k. Because Step 1 ensures the inclusion property (formula (1) on page 7), every initial estimate (m_i, vw_i) satisfies also the inclusion property. Hence, the decision pair $(\mathsf{MsgSet}_k, \mathsf{View}_{k+1})$ also satisfies the inclusion property. Thus, every process p_i receiving the decision value, first delivers the messages from MsgSet_k that it might not yet have delivered, and then delivers the new view View_{k+1}[7]. This trivially ensures the agreement conditions (A1) and (A2).

4.4 Handling of joins

Joins have not been considered up to now. In Section 4.3 we have considered the view change from View_k to View_{k+1} such that $\mathsf{View}_{k+1} \subset \mathsf{View}_k$ (possibly $\mathsf{View}_{k+1} \subseteq \mathsf{View}_k$). Joins can easily be handled within this context.

Consider a process $p_j \notin \mathsf{View}_k$. In order to join, p_j needs to send *join-req*(p_j) to at least one member of View_k. Assume p_i delivers *join-req*(p_j) in view k. The reception of a join request, similarly to a failure suspicion, will trigger the view change protocol. Moreover, *join-req*(p_j) is included in p_i's estimate of the messages delivered in view k. Let $(\mathsf{MsgSet}_k, \mathsf{View}_{k+1}^{temp})$ be the decision of the VSC protocol[8], and let $\mathsf{JoinSet}_k$ be defined as the set of join requests included in MsgSet_k:

$$\mathsf{JoinSet}_k \stackrel{\text{def}}{=} \{p_j \mid join\text{-}req(p_j) \in \mathsf{MsgSet}_k\}$$

[7] If $p_i \notin \mathsf{View}_{k+1}$, p_i might as well kill itself: as p_i is not in the next view, it will not receive the messages multicast to View_{k+1} anyway!

[8] The decision for the next view is noted here $\mathsf{View}_{k+1}^{temp}$, as it is not yet the next view.

The next view is then defined to include the *joined* processes to the decision value:

$$\mathsf{View}_{k+1} \stackrel{\text{def}}{=} \mathsf{View}_{k+1}^{temp} \cup \mathsf{JoinSet}_k$$

This scheme simultaneously handles both joins and leaves (i.e. failures) in a single view change.

4.5 Cost analysis of the VSC protocol

The VSC protocol of Section 4.3 has two steps, the second being the Chandra/Toueg consensus algorithm [5]. The advantages of this protocol are obvious: it solves the consensus with the $\diamond\mathcal{W}$ failure suspector if less than half of the processes crash, and $\diamond\mathcal{W}$ is the weakest failure suspector that allows to solve consensus when less than half of the processes crash [6]. In other words, there is no consensus protocol that solves the consensus problem in a weaker environment. So the question is: what are the drawbacks of the Chandra/Toueg consensus algorithm? Is it expensive in number of phases? The algorithm is based on the rotating coordinator paradigm: in each round of the protocol, a different process plays the coordinator role. The $\diamond\mathcal{W}$ failure suspector ensures that eventually there is a round r and a coordinator $coord_r$ that decides the consensus value. The possible multiple rounds might suggest that it is a very costly algorithm. This is, however, not necessarily the case: the number of rounds needed to complete the protocol is dependent on the number of incorrect failure suspicions. If the failure suspector is very aggressive (i.e. uses a small time-out, for example 500 msec), then the number of incorrect failure suspicions may be high, and the likewise number of rounds needed to complete the algorithm important. If the failure suspector is more conservative (time-outs around 5-10 sec), then incorrect failure suspicions will be uncommon[9], and the number of rounds needed to complete the consensus will be low: typically one round will be sufficient in most cases. To be more concrete, consider the following scenario:

- on a cut c_1, the current view is $\mathsf{View}_k = \{p_1, p_2, p_3, p_4, p_5\}$;
- consider on a later cut c_2 where p_1 is correctly suspected by all other processes, which launches the VSC protocol.

If no other suspicions are generated after the cut c_2, clever engineering of the VSC protocol will allow it to complete in two phases, with p_2 as the coordinator for the consensus[10] (Fig. 1):

[9] A link failure might indeed generate an incorrect suspicion.
[10] The message initiating the VSC protocol has to be added to these two phases.

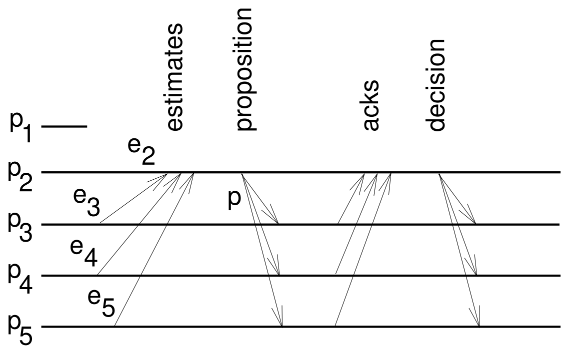

Figure 1: VSC protocol

Phase 1. As part of Step 1 of the VSC protocol (Sect. 4.3), $\mathsf{MsgSet}_{k,i}(c_2)$ is multicast to the coordinator p_2. Once p_2 has received these messages, it defines its initial value (m_2, vw_2) for the consensus as described in Section 4.3 (e.g. $vw_2 = \{p_2, p_3, p_4, p_5\}$), and switches to the consensus protocol.
Phase 1 completes by having process p_2 multicast its proposition (m_2, vw_2) for the consensus to View_k;

Phase 2. Phase 2 starts after p_3, p_4, p_5 have received the proposition (m_2, vw_2) from p_2. Once this proposition is received, p_3, p_4, p_5 send an acknowledgment to p_2. As soon as p_2 has received these acknowledgments, (m_2, vw_2) is the decision value, and the decision is multicast to View_k.

Thus, if the number of incorrect suspicions is low, the VSC protocol needs only one round of the Chandra/Toueg algorithm to complete the view change.

Assuming that the protocol needs only one round to complete, the number of messages needed to complete the view change protocol is equal to $4n$, where n is the size of the View_k (2 messages from the coordinator to View_k, 2 messages from each member of View_k to the coordinator). Moreover, groups are small (typically $n \leq 5$), as each group is intended to manage some replicated state. Large groups, as they occur sometimes in Isis, can be avoided in our system. Access to the group state information from outside is provided by the *clients* and *sinks* semantic described in [2], which is satisfactory for most applications.

4.6 When is progress ensured

In Section 1, we have discussed the challenging blocking problem. In a stable suspicion model, a partition might lead to block the system forever. We start by considering only link failures, and show the superiority of our mixed stable/unstable suspicion model over the classical stable suspicion model. For completeness, we will discuss then progress considering process crash failures.

4.6.1 Link failures

In a stable suspicion model (e.g. the model adopted by Isis), a partition might lead to blocking of the system forever. Consider the following scenario:

1. $\mathsf{View}_k = \{p_1, p_2, p_3, p_4, p_5\}$
2. on a cut c_1, simultaneous link failures lead to creating three partitions: $\Pi_1 = \{p_1\}$, $\Pi_2 = \{p_2, p_3\}$, $\Pi_3 = \{p_4, p_5\}$
3. on a later cut c_2, one link failure is repaired, e.g. two partitions remain: $\Pi_{1,2} = \{p_1, p_2, p_3\}$ and $\Pi_3 = \{p_4, p_5\}$
4. finally on a cut c_3, consider all link failures to be repaired, i.e. $\{p_1, p_2, p_3, p_4, p_5\}$ again fully connected.

Unless the link failure duration is shorter than the time-out of the failure suspector, the suspicions will be as follows on a cut c somewhere between the cuts c_1 and c_2:

- $\mathsf{Susp}_1(c) = \{p_2, p_3, p_4, p_5\}$
- $\mathsf{Susp}_2(c) = \mathsf{Susp}_3(c) = \{p_1, p_4, p_5\}$
- $\mathsf{Susp}_4(c) = \mathsf{Susp}_5(c) = \{p_1, p_2, p_3\}$

We discuss this scenario in the two suspicion models: stable suspicion model and mixed stable/unstable suspicion model.

Stable suspicion model. If suspicions are stable, none of the processes of View_k will ever be able to define View_{k+1} (a majority condition is required to define a new view [12]). Because of the stable suspicion model, repairing the links does not help! In other words the system remains blocked, even after the link failures have disappeared.

Mixed model: stable/unstable suspicions. Our VSC protocol will not block forever in the above scenario. As long as the partitions Π_1, Π_2, Π_3 exist, every $p_i \in \mathsf{View}_k$ is blocked in Step 1 of the VSC protocol (Sect. 4.3). As soon as the system evolves to the partitions $\Pi_{1,2}$, Π_3, the processes in the partition $\Pi_{1,2}$ will be able to complete Step 1 of the VSC protocol, and then solve the consensus problem in Step 2. In our mixed model, suspicions are unstable as long as a process has not started Step 2 of the VSC protocol: *for process p_i, the switch from Step 1 to Step 2 of the VSC protocol stabilizes its suspicions.* Thus, as long as link failures result in the absence of a majority partition, failure suspicions remain unstable. We can state a more general property of our mixed model. Assume for the moment that processes do not crash: if every link failure is eventually repaired, then failures never lead to infinite blocking (assuming of course the property of the weakest failure suspector $\diamond\mathcal{W}$).

4.6.2 Process crashes

We consider process crashes only, and discuss blocking in our mixed suspicion model. The discussion of the stable suspicion model would be similar. Termination of the consensus algorithm used in Step 2 of the VSC protocol requires that less than half of the processes crash, and the $\diamond\mathcal{W}$ failure suspector. Violation of the first condition obviously leads to infinite blocking. Consider for example $\mathsf{View}_k = \{p_1, p_2, p_3, p_4, p_5\}$. If p_1, p_2 and p_3 crash simultaneously, no further progress is possible any more. To ensure progress, the first requirement (less than half of the processes crash) has to be ensured in every view View_k. Consider the following scenario:

Scenario 1
- $\mathsf{View}_0 = \{p_1, p_2, p_3, p_4, p_5\}$
- a link failure leads to two partitions $\Pi_1 = \{p_1, p_2, p_3\}$ and $\Pi_2 = \{p_4, p_5\}$. $\mathsf{View}_1 = \{p_1, p_2, p_3\}$ is defined
- p_2 and p_3 crash, blocking the system forever.

The reader has probably noticed that in this scenario less than half of the initial view has crashed! Infinite blocking could have been avoided if the view change from View_0 to View_1 had not occurred. In other words, blocking would have been avoided if the failure suspicions would not have triggered a view change. However, we can imagine a different scenario in which changing view avoids blocking:

Scenario 2
- $\mathsf{View}_0 = \{p_1, p_2, p_3, p_4, p_5\}$
- p_4 and p_5 crash, and $\mathsf{View}_1 = \{p_1, p_2, p_3\}$ is defined
- p_2 crashes, and $\mathsf{View}_2 = \{p_1, p_2\}$ is defined.

In scenario 2, three processes out of the five initial processes have crashed, and progress is still possible (more than the half).

The reader might have noticed at that point the strong similarity between the definition of new views, and the dynamic voting technique for handling replicated data described in [8].

5 Transitivity of communications in the Internet

5.1 The problem of the non-transitivity of the COMM relation

The failure suspector FS_i attached to process p_i is responsible for suspecting processes and maintaining Susp_i, the set of processes suspected by p_i. Thus $\mathsf{S} - \mathsf{Susp}_i$, noted $\mathsf{CommSet}_i$, is the set of non-suspected processes, i.e. the set of processes with whom p_i is able to communicate. Ideally, for every process p_i in a view View_k, the set of processes, with whom p_i can communicate, should be equal to View_k:

$$\bigwedge_{p_i \in \mathsf{View}_k} \mathsf{CommSet}_i = \mathsf{View}_k \tag{2}$$

As soon as for some p_i, $\mathsf{View}_k \neq \mathsf{CommSet}_i$, the view change protocol of Section 4 is launched in order to eventually define View_{k+1} such that the new view matches $\mathsf{CommSet}_i$.

The relation COMM has been defined (Sect. 3) such that on every cut c

$$\mathrm{COMM}(c, p_i, p_j) \;\Leftrightarrow\; p_j \in \mathsf{CommSet}_i(c).$$

To understand the problem that occurs when COMM is not transitive, consider the following example:

- initially on a cut c_0, $\text{View}_k = \{p_1, p_2, p_3\}$;
- on a cut c_1, the link between p_1 and p_3 breaks, leading to the situation depicted on Figure 2: only the links between p_1, p_2, and between p_2, p_3 are operational;
- if communication between p_1 and p_3 are not re-routed through p_2, process p_1 will suspect p_3 and p_3 will suspect p_1[11], i.e. we get $\mathsf{CommSet}_1 = \{p_1, p_2\}$, $\mathsf{CommSet}_2 = \{p_1, p_2, p_3\}$, $\mathsf{CommSet}_3 = \{p_2, p_3\}$: both p_1 and p_3 will initiate a view change. Because p_2 is accessible by p_1 and p_3, the view change protocol will terminate (the property of the failure suspector $\diamond\mathcal{W}$ is met). The outcome of the view change can be one out of three possibilities (dependent on the specific implementation of the view change protocol):
  1. $\mathsf{View}_{k+1} = \{p_1, p_2, p_3\}$ (i.e. View_{k+1} is identical to View_k)
  2. $\mathsf{View}_{k+1} = \{p_1, p_2\}$
  3. $\mathsf{View}_{k+1} = \{p_2, p_3\}$

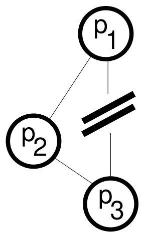

Figure 2: Transitivity problem

In none of the three cases, a stable state is reached, i.e. for none of these cases the above formula (2) (see page 11) is satisfied for all processes in View_{k+1}. In case 1, formula (2) is not satisfied for p_1, p_3; in case 2, formula (2) is not satisfied for p_2; in case 3, not for p_2. As a result, as soon as View_{k+1} is delivered, the view change protocol will again be launched, to end up again with one of the three cases above, etc. The problem is the non-transitivity of the COMM relation. In the above example, once the link p_1-p_3 has failed, we have $\text{COMM}(c, p_1, p_2)$ and $\text{COMM}(c, p_2, p_3)$, but not $\text{COMM}(c, p_1, p_3)$. It can easily be shown that, given View_k, a view change is not initiated if and only if the following property holds:

$$\bigwedge_{p_i, p_j, p_l \in \mathsf{View}_k} \Big(\text{COMM}(c, p_i, p_j) \wedge \text{COMM}(c, p_j, p_l) \;\Rightarrow\; \text{COMM}(c, p_i, p_l) \Big)$$

In other words, to prevent instability, the implementation has to do its best to ensure the transitivity of the COMM relation. As the transitivity property is ensured by routing, it is usually obtained for free in a local area network. This is, however, not the case in a wide area network, e.g. the Internet. Routing on the Internet is a very complex task. As routing tables are of bounded size, every possible route from a site s_i to a site s_j can not be stored in these tables. The situation depicted in the above example is thus not uncommon in the Internet[12]!

[11] A failure suspicion is the result of a time-out to a *request-reply* message.

[12] A real life experience on the Internet has shown the utility of the site routing. We give just one of several

5.2 Ensuring the transitivity of the Comm relation

The implementation of Phoenix uses UDP [7] as the basic layer for communication. We ensure the transitivity of the Comm relation by implementing routing on top of UDP. This is done using the self-stabilizing algorithm described in [9]. Each site s_i manages a routing table[13]. This table is based on local accessibility criteria, and on information in routing tables from other accessible sites. A link crash or a process crash will lead to modifications in some routing tables, and these modifications will initiate changes within other routing tables. The propagation is not immediate, but eventually the routing tables will become stable. On top of the routing protocol, transitivity of the Comm relation is ensured, which may of course increase the transmission time of messages. In Phoenix the time-outs of the failure suspector are automatically adapted based on the round-trip time of messages.

6 Conclusion

This paper has presented the problems of implementing view synchronous communication (VSC) in a large scale system prone to partition failures. This paper has also presented how these problems are solved in the Phoenix platform, a group infrastructure for developing fault-tolerant distributed applications. VSC defines the basic layer of the Phoenix architecture, on top of which various total order multicasts and uniform multicast can easily be implemented [4, 16, 14]. The Phoenix implementation of VSC solves the two large-scale specific problems: (1) infinite blocking, that can result from partition failures in the stable suspicion model (e.g. the Isis model), and (2) instability of view changes that can result from non-transitivity of communications. Problem (1) has been solved by adopting a mixed *unstable/stable* suspicion model, and a view change protocol based on the Chandra/Toueg consensus protocol [5], which is guaranteed to terminate in an asynchronous environment with the failure suspector $\diamond\mathcal{W}$. Problem (2) has been solved by implementing a site level routing protocol within Phoenix. This limits suspicions due to link failures, which in turn avoids unnecessary executions of the view change protocol. Solving the problems (1) and (2) ensures progress in Phoenix within a group g whenever the properties of the failure suspector $\diamond\mathcal{W}$ holds within g. As $\diamond\mathcal{W}$ is the weakest failure suspector to solve consensus in an asynchronous system [6], no VSC implementation can allow progress at the application layer in a case where Phoenix would not.

References

[1] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership Algorithms for Multicast Communication Groups. In *6th Intl. Workshop on Distributed Algorithms proceedings (WDAG-6), (LCNS, 647)*, pages 292–312, November 1992.

[2] Ö. Babaoğlu and A. Schiper. On Group Communication in Large-Scale Distributed Systems. In *Proceedings of the 7th ACM SIGOPS Workshop*, September 1994.

[3] K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, February 1987.

concrete examples. During some testing, involving one site in Chile, one in Italy and one in Switzerland, we observed that for more than 3 minutes, IP routing was not able to send messages from Switzerland to Chile. Our routing algorithm successfully rerouted these messages through Italy

[13] Phoenix does the routing on the site level.

[4] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, August 1991.

[5] C.T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. Technical Report 93-1374, Department of Computer Science, Cornell University, August 1993. A preliminary version appeared in the *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 325–340. ACM Press, August 1991.

[6] T. D. Chandra, V. Hadzilacos, and S. Toueg. The Weakest Failure Detector for Solving Consensus. In *proc. 11th annual ACM Symposium on Principles of Distributed Computing*, pages 147–158, 1992.

[7] D. E. Comer. *Internetworking With TCP/IP:Principles,Protocols,Architecture.* Prentice Hall, Stevenage, 1988.

[8] D. Davcec and A. Burkhard. Consistency and Recovery Control for Replicated Files. In *Proceedings of the 10th Symposium on Operating Systems Principles*, pages 87–96, 1985.

[9] S. Dolev, A. Israeli, and S. Moran. Self-Stabilization of Dynamic Systems Assuming Only Read/Write. *Journal of Distributed Computing*, 7(1), 1993.

[10] M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32:374–382, April 1985.

[11] A. Ricciardi, A. Schiper, and K. Birman. Understanding Partitions and the "No Partition" Assumption. In *IEEE 4th Workshop on Future Trends of Distributed Systems (FTDCS-93)*, September 1993.

[12] A. M. Ricciardi and K. P. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *Proc. of the 10th ACM Symposium on Principles of Distributed Computing*, pages 341–352, August 1991.

[13] A. Schiper and A. Sandoz. Primary Partition "Virtually-Synchronous Communication" Harder than Consensus. TR 94/49, Ecole Polytechnique Fédérale de Lausanne, Lausanne (Switzerland), April 1994.

[14] A. Schiper and A. Sandoz. Uniform Reliable Multicast in a Virtually Synchronous Environment. In *IEEE 13th Intl. Conf. Distributed Computing Systems*, pages 561–568, May 93.

[15] R. van Renesse, K. Birman, R. Cooper, B. Glade, and P. Stephenson. The Horus System. In K. Birman and R. van Renesse, editors, *Reliable Distributed Computing with the Isis Toolkit*, pages 133–147. IEEE Computer Society Press, 1993.

[16] U. Wilhelm and A. Schiper. A Hierarchy of Totally Ordered Multicast Protocols. Technical Report (in preparation), Ecole Polytechnique Fédérale de Lausanne, Lausanne (Switzerland), 1994.