UNIVERSITY OF
NEWCASTLE

COMPUTING SCIENCE

Enhancing Replica Management Services to Cope with Group Failures

Paul D Ezhilchelvan and Santosh K Shrivastava

TECHNICAL REPORT SERIES

No. [CS-TR-665]
October 1999

Contact: Paul Ezhilchelvan
[Paul.Ezhilchelvan@ncl.ac.uk]

A short version of this paper appeared in the Proceedings of the second International Symposium on Object Oriented Realtime Computing (ISORC99), St Malo, France, April 99.

Published by the University of Newcastle upon Tyne,
Department of Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, UK

Enhancing Replica Management Services to Cope with Group Failures

Paul D Ezhilchelvan and Santosh K Shrivastava

Department of Computing Science
University of Newcastle upon Tyne
Newcastle upon Tyne, NE1 7RU, UK

Abstract

In a distributed system, replication of components, such as objects, is a well known way of achieving availability. For increased availability, crashed and disconnected components must be replaced by new components on available spare nodes. This replacement results in the membership of the replicated group 'walking' over a number of machines during system operation. In this context, we address the problem of reconfiguring a group after the group as an entity has failed. Such a failure is termed a group failure which, for example, can be the crash of every component in the group or the group being partitioned into minority islands. The solution assumes crash-proof storage, and eventual recovery of crashed nodes and healing of partitions. It guarantees that (i) the number of groups reconfigured after a group failure is never more than one, and (ii) the reconfigured group contains a majority of the components which were members of the group just before the group failure occurred, so that the loss of state information due to a group failure is minimal. Though the protocol is subject to blocking, it remains efficient in terms of communication rounds and use of stable store, during both normal operations and reconfiguration after a group failure.

Keywords — system availability, object groups, group failures, node crashes, network partitions, membership views, membership services.

1. Introduction

In a distributed system, component replication (where a component is taken to mean a computational entity such as a process, module) is a well known way of achieving high availability. Equally well-known are the techniques for building a replica group using services such as membership and message ordering services. In this paper, we will consider the issue of enhancing the availability of a replica group in the presence of failures, while preserving the *strong consistency* property which requires that the states of all replicas that are regarded as available be mutually consistent. *Dynamic voting paradigm* is an efficient way of achieving this end [Jajodia90]: when, say, network failures partition the replica group into disjoint subgroups, availability is maintained only in the partition (if any) that contains a majority of the replicas (called the *master subgroup*), with the replicas in all other partitions becoming unavailable. That is, the majority partition (if any) forms the master sub-group and offers the services previously provided by the group; if a partition further disconnects a majority of the current master subgroup from the rest, then this connected majority becomes the new master subgroup. Thus, each newly formed master sub-group contains a majority of the previously existed master sub-group. Replica management with dynamic voting offers a better way of maintaining system availability than *static voting* that requires a majority of all the members that initially formed the group to remain connected. The following example illustrates dynamic voting:

Stage 0: Let the group configuration be initially G_0 = {C1, C2, C3, C4, C5, C6, C7}, where Ci is the i^{th} component.

Stage 1: Say, a network partition splits G_0 into G_1 = {C1, C2, C3, C4, C5} and G'_1 = {C6, C7}; G_1 now becomes the master subgroup and thereby the new, second group configuration.

Stage 2: Say, G_1 splits into G_2 = {C1, C2, C3} and G'_2 = {C4, C5}; G_2 now becomes the master subgroup and thereby the third group configuration.

The above example indicates how the dynamic voting can preserve the availability of group services even though the original group G_0 got split into islands with each island having less than half the members of G_0. Availability can be however maitained only if the master subgroup exists after a failure. Suppose that after stage 2, each member of G_2 detaches from other members. Now, no master subgroup exists and hence the normal services can no longer be provided. We call this a *group failure* (*g-failure* for short). Note that many combinations of failures can lead to a g-failure. For example, a g-failure after stage 2 can be caused by simultaneous crashing of each member of G_2, crashing of C3 and detachment of C2 and C1, and so on. When the bound on communication delays between components is not known with certainty, a g-failure can occur even in the absence of any physical failure in the system: when a sudden burst of network traffic, for instance, increases the communication delay between two connected components beyond what was considered to be likely, each component can falsely conclude that the other is not responding and hence must have crashed or got disconnected. Therefore, g-failures should not be regarded as rare events when bound on message delays cannot be estimated accurately.

Let us assume that the components have stable states which do not get destroyed by node crashes. Given that the component state survives node crash, it would be preferable to have the replica management service enhanced to cope with g-failures, instead of relying only on cold-start to resume the group services after a g-failure. To achieve this, we propose a *configuration protocol* that enables the members of the last master subgroup prior to a g-failure, to reconstruct the group once sufficient number of those members have recovered and got reconnected. Of course, the protocol must ensure that only one such group is formed. The protocol objectives cannot be met solely by the services used to build a replica group, in particular the membership service. To illustrate this, let us continue on the above example into stage 3 described below:

Stage 3: C3 crashes before it could record in its stable store the fact that the new master subgroup G_2 ={C1, C2, C3} has been formed; the remaining members of G_2, C1 and C2, record in their stable store that G_2 is the latest master subgroup and then disconnect from each other.

No master-subgroup now exists and a g-failure has occurred. Next, suppose that C3 recovers and reconnects with C4 and C5, and C2 reconnects to C1. The set {C3, C4, C5} forms the 'master subgroup' on the basis that its members form a majority of the last group configuration G_1 that is known to all of them, while {C1, C2} also forms the 'master subgroup' on the same basis that its members are a majority in the last known configuration G_2. Now, we have two live master subgroups. To prevent this from happening, we require that (i) a new master subgroup be considered to have been formed only after a majority of the previous master subgroup have recorded in the stable store the composition of the new master subgroup (*req1*); and, (ii) the master subgroup constructed after a g-failure include at least a majority of the members of the latest master subgroup formed prior to the g-failure (*req2*). Requirement *req1* ensures that there can be only one group configuration that qualifies to be the latest master subgroup formed before a g-failure (and, in general, at any given time). In the above example, a majority of G_1 did not record G_2 before the occurrence of the g-failure of stage 3; so, only G_1 is the latest master subgroup formed before a g-failure. Requirement *req2* permits no more than one master subgroup to emerge after a g-failure.

We assume that the construction of the replica management system (with dynamic voting) can avail the use of a group membership service which provides each operational component with an agreed set of components that are currently believed to be functioning and connected. For such a replica management system, we develop a *configuration management* subsystem - the main contribution of the paper - that provides (i) a group view installation service to enable members of the master subgroup to record group membership information on stable store; and (ii) a group configuration service that makes use of these stable views to enable group formation after a g-failure as soon as enough number of the components of the last configuration have recovered and reconnected. A prototype version of the configuration management service described here has been implemented [Black97] on an existing replica management system called Somersault [Murray97]. Our service enhances Somersault by providing recovery from group failures.

The paper is structured as follows: section two introduces the system architecture, some definitions and notations; it also specifies the two services provided by the configuration management subsytem, namely the view installation and the group configuration services. The next two sections describe in detail how these services are provided. Section five compares and contrasts our work with the approaches taken in the published papers in this area, and concludes the chapter.

2. System Overview and Requirements

2.1. Assumptions and System Structure

It is assumed that a component's host node can crash but contains a stable store whose contents survive node crashes. Components communicate with each other by passing messages over a network which is subject to transient or long-lived partitions. We assume that a partition eventually heals and a crashed node eventually recovers; the bound on repair/recovery time is finite but not known with certainty. For increased availability, we permit new components created on spare nodes to join the group, with no restriction on the number of such joining nodes and on the time of their joining. Our system leaves to the administrator to decide how many among the available spare nodes should be instructed to join the group, and when. Given that the spares instructed by the administrator are attempting to join the group, our system enables them to join with a guarantee that they could compute the most uptodate component state from the existing members. For simplicity, we assume that members of a group do not voluntarily leave the group, but are only forced out because of crashes or partitions.

2.1.1. View Maker (VM) Subsystem

We assume that our replica management system has been constructed by making use of the services provided by a group membership subsystem. This subsystem resident in the host node of an active component, say p, constructs membership *views* for p, where a view is the set of components currently believed to be functioning and connected to p. We call this subsytem the *View Maker*, or VM for short, and denote the VM of p as VM_p. In delivering the uptodate views constructed, VM_p is required to provide the abstraction of *view synchrony* or *virtual synchrony* if primary-partition model is assumed for the underlying communication subsystem. We refer the reader to [Babaoglu95, Babaoglu97] and [Schiper94] for a complete list of the properties of view synchronous and virtual synchronous abstractions, respectively. Below, we highlight some of these properties that are considered important for our discussions.

vs1: p is present in any view constructed by VM_p. (*self-inclusion*.)

vs2: a message *m* from another component q is delivered to p only when the view constructed by VM_p prior to the delivery of *m* contains q. (*view-message integrity*.)

vs3: the delivery of constructed views is synchronised with the delivery of messages such that components receive identical set of messages between consecutive views that are identical. (*view-message synchrony*.)

vs4: If VM_p delivers a view *v*, then for every component q in *v*, either VM_q delivers *v* or VM_p constructs consecutive view *w* that excludes q. (*view agreement*.)

There are many protocols in the literature which can be used to implement the assumed VM subsystem; e.g., [Birman87, Ricciardi91, Mishra91] for an asynchronous system with the primary-partition assumption, [Melliar-Smith91, Moser96, Amir92, Ezhilchelvan95, Babaoglu95] for partitionable asynchronous systems. These protocols are not designed to cope with g-failures. The subsystem described below deals with g-failures using the services of the VM subsystem.

2.1.2. Configuration Management (CM) Subsystem

On top of the VM service exists a configuration management (CM) subsystem (see Figure 2). CM of component p, denoted as CM_p, carefully records the view information provided by VM_p in the local stable store. In a traditional replica management system, a new view decided by VM_p is usually delivered straight to p. In our system, it reaches p via CM_p. VM_p regards CM_p as an application and delivers every new view it decides.

CM_p of member p essentially provides the following three functionalities.

(i) it considers each view delivered by VM_p and decides whether a g-failure may have occurred. If g-failure occurrence is ruled out, CM_p passes on that view to p, provided certain conditions are met which ensure strong consistency.

(ii) if a new view delivered to p contains a spare node attempting to join the group, CM_p facilitates the spare node (in co-operation with CM of other members in the new view) to compute the most recent component state.

(iii) if a view constructed by VM_p indicates that a g-failure may have occurred, CM_p executes a *configuration protocol* with CM of connected components. This execution ensures that if the group is reconfigured, it is the master subgroup of the configuration that existed just before the g-failure was suspected to have occurred.

It must be emphasized here that CM_p can only suspect, not accurately diagnose, the occurrence of a g-failure when it inspects a new view from VM_p. To illustrate this consider the disconnected component C3 in figure 1. With the recent group membership being {C1, C2, C3}, when CM of C3, CM_3, is delivered a singleton view {C3}, it cannot know whether the partition has split the group in three ways causing a g-failure (as in Fig. 1(a)), or in two ways (as in Fig. 1(b)) permitting C1 and C2 to form the next master sub-group. So, in both cases, CM_3 would suspect a g-failure and execute the configuration protocol. In case of 3-way partition, C3 will form the post-g-failure master subgroup with, say, C1, if it re-connects to C1 while CM_1 is also executing the protocol. In the second case, when the partition heals, C3 will learn that it has been 'walked over': C1 and C2 have formed the new master subgroup without it; C3 will then join the pool of spares. Note that it is also possible for C3 to be walked over in the first case: if the isolation of C3 lasts so long that C1 and C2 reconnect in the mean time and form the next master group. Thus, the outcome of the reconfiguration attempts by components is decided by the pattern and timing of components recovery and reconnection.

Figure 1. (a) 3-way partitioning (b) 2-way partitioning.

2.2. View Names within the System

Our replica management system (above the communication layer) is structured in two layers as shown in fig. 2. Recall that CM_p delivers to p a view constructed by VM_p, only if certain conditions are met. That is, a view becomes more significant as it moves up within the system. To reflect this, we call a view differently at different levels. The views constructed by VM_p are called the *membership views* or simply Mviews. VM_p delivers Mviews to CM_p via a queue called $ViewQ_p$ where Mviews are placed in the order of delivery. CM_p deals with one Mview at a time, and only when an Mview reaches the head of $ViewQ_p$, which is denoted as $head_p$. CM_p stores the $head_p$ in the stable store as the new component view, provided a set of conditions are met. The component view of p is called $Cview_p$. Only $Cview_p$ is made visible to component p and provides p with the current membership view. For reasons discussed earlier (see *req1* of Section 1), making $head_p$ as $Cview_p$ is done in two stages; $head_p$ is first *recorded* in stable store as the *stabilised* view of p called $Sview_p$, and then *installed* as $Cview_p$. CM uses a view numbering scheme for sequentially numbering the view contents of $Sview_p$ and $Cview_p$.

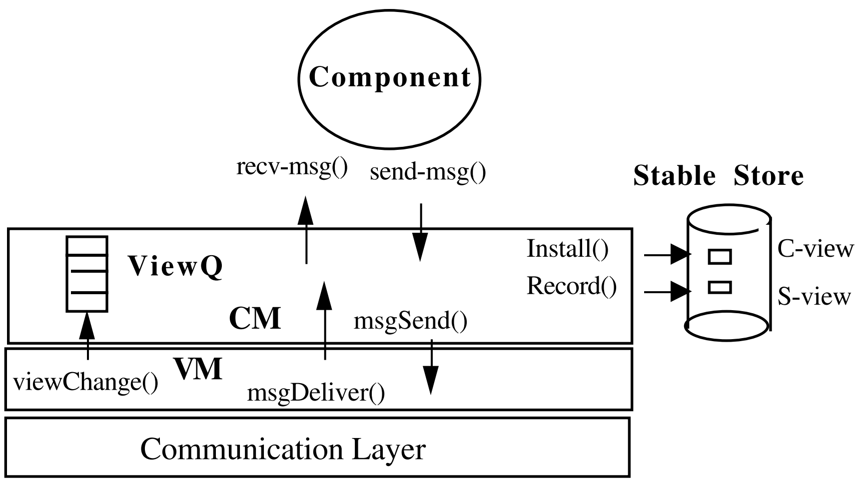

Figure 2. The system Architecture.

2.3. Notations and definitions

Each component p maintains three variables $status_p$ = (*member, spare*), $mode_p$ = (*normal, reconfiguration, waiting, joining*), and *view-number*$_p$ (an integer variable) in its stable store. In addition, it also maintains two view variables, $Cview_p$ and $Sview_p$, initialised to null set, if p is spare. $Sview_p$ has a view number associated with it, and the view number of $Cview_p$ is indicated by *view-number*$_p$. $status_p$ is set to *member* when p considers itself to be a member of the group, or to *spare* otherwise. When a member p (with $status_p$ = *member*) observes a g-failure and subsequently has to execute the configuration protocol, it sets its $mode_p$ to *reconfiguration*. The $mode_p$ changes to *normal* if p succeeds in becoming a member of the re-formed group; otherwise p becomes a spare setting $status_p$ = *spare* and $mode_p$ = *waiting*. The $mode_p$ of a spare component p can be either *waiting* or *joining*; the former is when p is waiting to be informed by its VM_p that it has been connected to members of the group; once connected, p attempts to join the group by setting its $mode_p$ to *joining*. If the join attempt by p succeeds, $status_p$ is set to *member* and $mode_p$ is set to *normal*.

The variable *view-number*$_p$ is intialised to -1 at system start time (before the group is formed) and whenever p becomes a spare; it is incremented every time CM_p installs a new Cview.

We define the terms *survivors* and *joiners* for a pair of Mviews constructed by VM_p of a component p. Let vu_i, vu_{i+1}, ... vu_j, $j \geq i+1$, be a sequence of Mviews constructed by VM_p in that order. The set *survivors*(vu_i, vu_j) is the set of all components that survive from vu_i into every Mview constructed upto vu_j: *survivors*(vu_i, vu_j) = $vu_i \cap vu_{i+1} \cap ... \cap vu_j$. The term *joiners*($vu_i$, vu_j) will refer to the set of components in vu_j which are not in *survivors*(vu_i, vu_j): *joiners*(vu_i, vu_j) = vu_j - *survivors*(vu_i, vu_j). Finally, we define M_SETS(g) for a set g of components as the set of all majority subsets s of g: M_SETS(g) = $\{s \mid s \subseteq g \wedge |s| > (|g|)/2\}$.

2.4. View Maintenance

When $viewQ_p$ is non-empty, CM_p of member p checks for the occurrence of a g-failure by inspecting the contents of $head_p$, and by evaluating the condition: *survivors*($Cview_p$, $head_p$) $\in$ M_SETS($Cview_p$). If this condition is not satisfied, a g-failure is assumed to have occurred. CM_p first sends an *Abort* message to all components in *joiners*($Cview_p$, $head_p$), informing the CM of any joiner not to attempt at recording/installing $head_p$. We will denote this *Abort* message of CM_p (which contains $head_p$) as *AMsg*$_p$($head_p$). CM_p then sets its variable *mode*$_p$ to *reconfiguration* and executes the configuration protocol to reconfigure the group. If the above condition is met, a copy of $head_p$ is atomically recorded in the local stable store as the new $Sview_p$ with the view number = (*view-number*$_p$+1), *provided recording conditions are satisfied*. This $Sview_p$ represents the potential next $Cview_p$. If the recording conditions are not met, the CM_p either concludes that a g-failure has occurred and proceeds to execute the configuration protocol setting *mode*$_p$ to *reconfiguration,* or dequeues $head_p$ and proceeds to work with the next $head_p$ (if any). The recording conditions, the need for them, and how they are verified will be discussed in the next section.

The newly recorded $Sview_p$ is regarded ready for becoming the next $Cview_p$ if *an installation condition* is satisfied (again, the need for this condition and how it is verified will be discussed in the next section). In which case, CM_p installs the new component view by replacing the current $Cview_p$ by $Sview_p$, and dequeues and discards $head_p$. The local stable store update operations are indicated here within curly braces and are carried out atomically: {$Cview_p$:= $Sview_p$; *view-number*$_p$:= *view-number*$_p$ + 1;} If the installation condition is not met, a g-failure is considered to have occurred and the configuration protocol is executed.

The view number of $Cview_p$ is indicated in *view-number*$_p$. Since $Cview_p$ and $Sview_p$ are modified along with their view number as an atomic operation, there will be exactly one $Cview_p$ and one $Sview_p$ associated with a given view number, provided the view numbers increase monotonically. Further, CM_p installing the $Sview_p$ (as the next $Cview_p$) can be interrupted only by its suspecting a g-failure; in particular, when no g-failure is suspected, CM_p will not record a new $Sview_p$ until the existing one is installed. Thus, in the absence of g-failure suspicions, either the view number of $Sview_p$ = the view number of $Cview_p$, or the view number of $Sview_p$ = view number of $Cview_p+1$, the latter being true while the installation condition is being waited upon to be satisfied. Let $Vu_p(k)$ be the Sview or the Cview that CM_p handled with view number k; similarly, let $Vu_{p'}(k')$ be an Sview or a Cview that $CM_{p'}$ handled with view number k', where p and p' may be the same component or distinct ones. We will say $Vu_{p'}(k')$ *is later than* $Vu_p(k)$, denoted as $Vu_{p'}(k') \gg Vu_p(k)$, if and only if $k' > k$.

2.5. Requirements of the CM subsystem

We now state the two requirements the CM subsystem must meet. The first one is concerned with the "normal service" period during which no g-failure occurs, whereas the second one is concerned with group formation after a g-failure.

Existence of at most one master subgroup at any time is achieved by ensuring that any two components that install Cviews with identical view number, install identical views. Let $Cview_p(k)$ denote the Cview that p installs with view number k, $k \geq 0$. The predicate $installed_p(k)$ is true if p has installed $Cview_p(k)$, and the predicate $!Cview_p(k)$ is true if $Cview_p(k)$ is unique, i.e., no component q can install $Cview_q(k)$ that is different from $Cview_p(k)$:

$$!Cview_p(k) \Rightarrow \forall q\text{: } \neg\, installed_q(k) \vee Cview_q(k) = Cview_p(k).$$

Note that any view installed by a component must contain the installing component. So, if $Cview_p(k)$ is unique, then no component outside $Cview_p(k)$ installs a Cview with view number k; so, there can be only one kth membership set for the group, hence only one kth master subgroup.

During normal service period, the CM modules of components ensure that the Cviews installed are sequentially numbered, and that the kth Cview installed by p is unique, provided that (k-1)th Cview installed is unique.

Formally, CM subsystem ensures:

Requirement 1:

$$\forall k > 0,\ installed_p(k) \Rightarrow \exists p'\text{: } installed_{p'}(k-1)\text{; and,}$$
$$\forall k > 0,\ installed_p(k) \wedge !Cview_{p'}(k-1) \Rightarrow !Cview_p(k).$$

Section 3 discusses how this requirement is met.

If we assume that Cview(0) is unique when the group is initially formed and that the above requirement is met, then there will exist a unique latest Cview at any time. We define this latest view as the *last Cview*, or simply the *last*.

Requirement 2: following a g-failure, a set Σ of functioning and connected components with identical Cview, *restart-view*, should be formed as soon as possible, with the following properties:

Uniqueness: $\Sigma \cap last \in$ M_SETS(*last*). If *last* is unique before g-failure, there can be only one Σ that can contain a majority of the *last*.

Continuity: *restart-view* $\neq$ *last* $\Rightarrow$ view-number(*restart-view*) =view-number(*last*)+1. The sequentiality of CView numbering is preserved across g-failures. Thus, coping with g-failures is transformed into a view installation of different kind which nevertheless preserves the uniqueness and numbering of Cviews during the normal service period. Section 4 discusses how requirement 2 is met.

3. Maintaining Unique Component Views

We describe the recording and installation conditions mentioned earlier and discuss how they help meet Requirement 1. We will first define a predicate $recd_p(m_q)$ which becomes true when CM_p of component p receives a message m_q from CM_q of another component q, and becomes false if CM_p believes that q had crashed or got disconnected before m_q is sent. We later present a non-blocking algorithm for CM_p to evaluate this predicate.

3.1. Recording Conditions for a member component

Let us assume (as induction hypothesis) that any two members have identical Cview with identical view number. That is, for members p and q, $Cview_p = Cview_q$ and $view\text{-}number_p = view\text{-}number_q = k$ (say). Let $head_p$, the Mview at the head of $viewQ_p$, become non-empty for member p. *survivors*($Cview_p$, $head_p$) and *joiners*($Cview_p$, $head_p$) follow different procedures for recording Sviews. Let us consider the *survivor* or *member* p first and let *survivors*($Cview_p$, $head_p$) ∈ M_SETS($Cview_p$). As discussed in subsection 2.4, CM_p can record a copy of $head_p$ as $Sview_p$ only if recording conditions are satisfied. These recording conditions essentially ensure that all *joiners*($Cview_p$, $head_p$) have obtained view information as well as replica states from *survivors*($Cview_p$, $head_p$) and made it stable. This is necessary, as a *joiner* component j has no replica state and other view related information. (It will only have $view\text{-}number_j = -1$, $Sview_j = Cview_j = null$, $mode_j = waiting$ and $status_j = spare$.) So, the recording conditions need to be satisfied only if there are *joiners* in $head_p$, i.e., *joiners*($Cview_p$, $head_p$) ≠ { }.

Suppose that there are *joiners* in $head_p$. CM_p multicasts a *State* message to every component in $head_p$ (including itself). This message contains a copy of $head_p$, $Cview_p$, *survivors*($Cview_p$, $head_p$), $view\text{-}number_p$, and p's state. We will denote this message of CM_p as $SMsg_p(head_p)$. CM_p then waits to see whether (i) enough number of *survivors* in $head_p$ have sent their *State* messages, and (ii) all *joiners* have computed and recorded the component state and also the view information in their stable store.

We will suppose that a *joiner* j in $head_p$ can compute the component state only by receiving *State* messages from some minimum number of distinct components in $Cview_p$ which is the group membership when $head_p$ is being dealt with. We will assume that this number is proportional to the size of $Cview_p$ and is some function of $|Cview_p|$, denoted as $\Phi(Cview_p)$. (If it is a fixed one and not proportional to the size of $Cview_p$, then $\Phi(Cview_p)$ will be a constant function.) Since at most less than ($|Cview_p|/2$)components need not survive into $head_p$ without causing a g-failure, $\Phi(Cview_p)$ cannot exceed $(|Cview_p|/2)+1$. So, $1 \le \Phi(Cview_p) \le (|Cview_p|/2)+1$.

Recording Condition 1 (*rc1*): It is to verify that at least $\Phi(Cview_p)$ survivors in $head_p$ have sent their *State* messages. Formally,

$$|\{q \in survivors(Cview_p, head_p): recd_p(SMsg_q(head_p))\}| \ge \Phi(Cview_p).$$

Recording Condition 2 (*rc2*): It is to ensure that all *joiners* in $head_p$ have computed and stored the component state and also recorded Sview which is the same as $head_p$. We will suppose that after CM_j of joiner j has stored the component state and recorded an Mview, say *vu*, as $Sview_j$, it multicasts a *Recorded* message to every component in *vu*. This message contains the recorded view *vu* and is denoted as $RMsg_j(vu)$. So, the second condition is that CM_p receive an $RMsg_j(head_p)$ from every joiner j in $head_p$. Formally, $\forall j \in joiners(Cview_p, head_p)$: $recd_p(RMsg_j(head_p))$.

If *rc1* and *rc2* are met, CM_p atomically records a copy of $head_p$ as its next $Sview_p$ with view number = $view\text{-}number_p+1$. It then multicasts an $RMsg_p(head_p)$ to all components (including itself) in $head_p$. If *rc1* is met but not *rc2*, CM_p dequeues $head_p$ from $ViewQ_p$ but retains a copy to evaluate $survivors(Cview_p, head_p)$ for the next $head_p$. If *rc1* is not met, CM_p proceeds to execute the reconfiguration protocol after setting $mode_p$ to *reconfiguration*. Since no joiner can send $RMsg(head_p)$ without first receiving at least $\Phi(Cview_p)$ *State* messages, it is not possible for *rc2* to be met without *rc1*.

3.2. Recording Condition for a joining component

The recording condition is verified by CM_j of joiner j (with $mode_j$ = *waiting*) as soon as its $head_j$ - the first Mview in $ViewQ_j$ - is constructed by VM_j. It should be designed to become false if it is not possible for CM_j to receive the minimum number of *State* messages from members in $head_j$. The design is made somewhat difficult by the fact that when VM_j delivers an Mview it cannot indicate who in that Mview are members and who else (except j itself) are joiners. VM_j can obtain such information only from VMs of member components. Recall that, as far as VM modules of member components are concerned, the local Cview is transparent and is merely an internal variable used by a local application called CM (see figure 2). Moreover, when *rc1* is met but not *rc2*, CM_p of member p dequeues $head_p$, and proceeds to work with the next Mview in $ViewQ_p$; therefore, VM_p cannot even assume that when a given Mview reaches the $head_p$, the Mview it delivered immediately before $head_p$ would have been installed as $Cview_p$. So, CM_j cannot rely on VM_j to indicate the Cview of members in $head_j$.

When CM_j does not know $Cview_p$ of member p in $head_j$, its attempt to record $head_j$ can result in a deadlock if $head_j$ contains more than one joiner. For example, if every member p in $head_j$ crashes before sending the *State* or the *Abort* message, then each joiner will wait for ever to receive *State* messages from other joiners. Therefore, it is essential that CM_j first constructs a *reference Cview* which can be effectively used in place of $Cview_p$ of member p in $head_j$ until an $SMsg_p(head_j)$ is received from p which will contain a copy of $Cview_p$. This reference Cview constructed for working with $head_j$ is denoted as $RefCview_j(head_j)$ and is initially set to $head_j$ itself. (Since the discussions are for a given $head_j$, we will refer to $RefCview_j(head_j)$ as simply $RefCview_j$.) CM_j then sends a *Join* message to every component in $head_j$, announcing that it is a joiner. We denote this message as $JMsg_j(head_j)$. Whenever CM_j receives $JMsg_{j'}(head_j)$, it removes the sender j' of that message from $RefCview_j$. However, if it receives an $SMsg(head_j)$, it irreversibly sets $RefCview_j$ to the Cview contained in that message. No $JMsg(head_j)$ that is received after receiving the first $SMsg(head_j)$ modifies

RefCview$_j$. The *survivors* and *view-number* contained in the received *SMsg*($head_j$) are noted in variables *members*$_j$ and *RefCviewNo*$_j$, respectively.

Once *RefCview*$_j$ is initialised to $head_j$, the recording condition stated below is waited upon to become true or false. (Verifying the recording conndition is done concurrently to modifying or irreversibly setting *RefCview*$_j$.) This condition is similar to *rc1* stated above for a member:

Recording Condition for joiner (*rc_joiner*): It verifies whether at least $\Phi(RefCview_j)$ distinct components sent their *State* messages. Formally,

$$|\{q \in RefCview_j: recd_j(SMsg_q(head_j))\}| \geq \Phi(RefCview_j).$$

If *rc_joiner* is met, CM_j atomically records a copy of $head_p$ as its next *Sview*$_j$ with view number = *RefCviewNo*$_j$+1 and sets *mode*$_j$ = *joining*. It then multicasts an *RMsg*$_j$($head_j$) to all components (including itself) in $head_j$. If *rc_joiner* is not met or if an *Abort* message *AMsg*($head_j$) is received, CM_j dequeues $head_j$ from $ViewQ_j$ and discards it.

Recall that CM_p multicasts *AMsg*$_p$($head_p$) only if *survivors*($Cview_p$, $head_p$) $\notin$ M_SETS($Cview_p$) when it starts to deal with $head_p$. So, it sends either *SMsg*$_p$($head_p$) or *AMsg*$_p$($head_p$), not both, for a given $head_p$; hence CM_j will not receive an *AMsg*$_q$($head_j$) once *rc_joiner* is met. Otherwise, this would mean that CM_p sent *SMsg*$_p$($head_j$) without suspecting a g-failure at the start, while CM_q of member q has $head_q = head_j$ and *survivors*($Cview_q$, $head_q$) $\notin$ M_SETS($Cview_q$). This would in turn mean that $Cview_p$ and $Cview_q$ are not identical which is a violation of the induction hypothesis.

To illustrate how certain failure cases that could lead to deadlock are handled, consider the group {p,q,r} with unique $Cview_p$; i.e., $Cview_p = Cview_q = Cview_r$ = {p,q,r}. Let the VM modules deliver an enhanced Mview such that $head_p = head_q = head_r$ = {p,q,r,j,j1,j2,j3} = $head_j$, where j, j1, j2, and j3 are joiners. Say, p, q, and r crash before multicasting their *State* messages; if {j, j1, j2, j3} remain connected, *RefCview*$_j$ eventually changes to {p, q, r} from its initial value of $head_j$. By its definition, *recd*$_j$(*SMsg*$_c$($head_j$)) will become false for crashed c = p, q, and r and CM_j will deduce that *rc_joiner* cannot be met.

3.3. Installation Conditions

Having recorded $head_p$ as $Sview_p$, CM_p installs the $Sview_p$ as the new $Cview_p$ only after verifying that a majority of the existing $Cview_p$ have recorded the $head_p$.

Installation condition (*ic*): {q ∈ *survivors*($Cview_p$, $head_p$): *recd*$_p$(*RMsg*$_q$($head_p$))} ∈ M_SETS($Cview_p$).

The CM_j of a *joiner* j has two installation conditions. The first one verifies whether all joiners of $head_j$ have recorded $head_j$; the second one is the same as the *ic* stated above for member p. Note that CM_j has recorded $head_j$ means that it has received *State* messages from some member p in $head_j$; so, *RefCview*$_j$ = $Cview_p$ and *members*$_j$ = *survivors*($Cview_p$, $head_p$).

Installation condition 1 for joiner (*ic1_joiner*):

$$\forall c \in \text{head}_j - \mathit{members}_j: \mathit{recd}_j(\mathit{RMsg}_c(\text{head}_j)).$$

Installation condition 2 for joiner (*ic2_joiner*):

$$\{p \in \mathit{members}_j: \mathit{recd}_j(\mathit{RMsg}_p(\text{head}_j))\} \in \text{M\_SETS}(\mathit{RefCview}_j).$$

If both conditions are met CM_j makes component j a member by atomically executing: {$Cview_j$:= $Sview_j$; $view\text{-}number_j$:= $RefCviewNo_j$ + 1; $status_j$ = *member;* $mode_j$ = *normal*; }. The $head_j$ is then dequeued and discarded. If the first condition is not met, no member p in $head_j$ would have recorded $head_j$; so, CM_j's recording of $head_j$ is undone by atomically executing: {$Sview_j$ = *null*; $mode_j$ = *waiting*;}. The $head_j$ is then dequeued and discarded. If only the second condition is not met, CM_j sets its $mode_j$ to *reconfiguration* and executes the reconfiguration protocol. Observe that when CM_j sets $mode_j$ to *reconfiguration*, $Cview_j$ and $view\text{-}number_j$ remain unchanged at their initial values which are *null* and -1 respectively.

3.4. Correctness and Liveness

Correctness: Suppose that CM_p installs $head_p$ as the new $Cview_p$ with view number (k+1). The majority requirement in the installation condition (*ic*) implies that a majority of $Cview_p(k)$ have recorded $head_p$ as their Sview with view-number (k+1). The recording condition (*rc2*) ensures that every CM of *joiners*($Cview_p(k)$, $Cview_p(k+1)$) has also recorded $head_p$ as its Sview with view-number (k+1). No CM records a new Sview before the existing one is installed. Therefore, given that $Cview_p(k)$ is unique, if CM_q of a *survivor* or *joiner* q installs $Cview_q(k+1)$, then $Cview_q(k+1) = Cview_p(k+1)$. This means that $Cview_p(k+1)$ is also unique.

Liveness: CM_p verifying the recording/installation condition requires the evaluation of the predicate $recd_p(m_q)$ which in turn involves checking whether an expected message m_q has been/can be received from CM_q. Since the node of q can crash before m_q can be sent, the evaluation of $recd_p(m_q)$ must involve checking whether q continues to be present in the subsequent Mviews constructed by VM_p. With this in mind, we present an algorithm for evaluating $recd_p(m_q)$ which does not block indefinitely.

Figure 3 shows the ViewQ's of CM_p and CM_q which, for simplicity, are taken to be identical. We will also assume that $Cview_p = Cview_q$ = {p, q, r1, r2, r3} and view-$number_p$ = view-$number_q$ = k (say). That is, $Cview_p(k)$ is unique. Let us denote the Mviews of $ViewQ_p$ and $ViewQ_q$ as: *vu1* = {p, q, r1, r2, j}, *vu2* = {p, q, r1, j} and *vu3* = {p, q, j}. *vu1* indicates the disconnection of member r3 (from p and q) and the inclusion of a new component j, *vu2* the disconnection of r2, and *vu3* the disconnection of r1.

We define $List_p$(Mview) as the set of messages which VM_p intends to deliver between the delivery of Mview and the delivery of the immediate successor view to Mview. $List_p$(*vu3*) is shown to be open and will remain so until a successor view to *vu3* is constructed. $List_p$(*vu1*) and $List_p$(*vu2*), on the other hand, are shown 'closed' to indicate that no received message can enter these lists any longer. By the view-message synchrony property of the VM subsystem (see §2.1.1), $List_p$(*vu1*) = $List_q$(*vu1*), and $List_p$(*vu2*) = $List_q$(*vu2*).

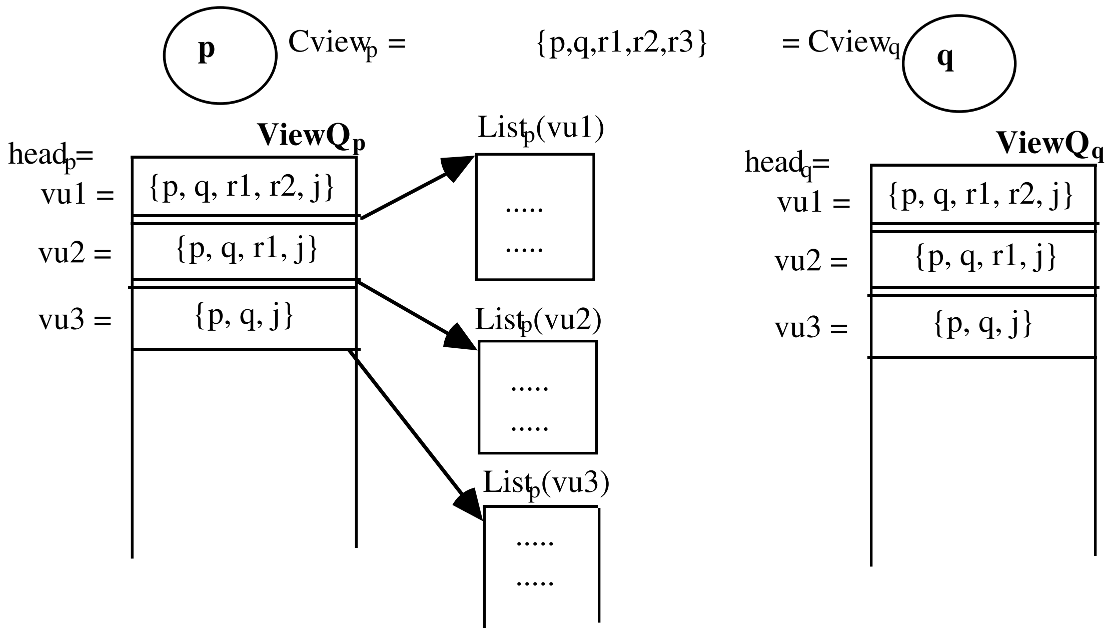

Figure 3. Closed and Open lists of messages delivered by VM after a given Mview.

The algorithm for evaluating $recd_p(m_q)$ is as follows: CMp waits for one of the following two comditions to become true.

Evaluation condition 1 (*ec1*): $\exists\ vu \in$ ViewQ$_p$: $m_q \in$ List$_p(vu) \wedge (vu \neq$ head$_p \Rightarrow$ q $\in$ *survivors*(head$_p$, vu)).

Evaluation condition 2 (*ec2*): $\exists\ vu \in$ ViewQ$_p$: q $\notin vu$.

The condition *ec1* is true when m_q is present in List$_p(vu)$ for some Mview *vu* in ViewQ$_p$ and q is present in all the views VM$_p$ constructed from head$_p$ through to this *vu*; *ec2* becomes true when VM$_p$ constructs an Mview without q.

boolean $recd_p(m_q)$

{wait until *ec1* $\vee$ *ec2*; *if ec1 then* return *true else* return *false*;}

Recall that CM$_p$ evaluates $recd_p(m_q)$ only for such q $\in$ head$_p$. Suppose that VM$_p$ constructs an Mview *vu* that does not contain q (*ec2*). By message-view integrity property of VM, the expected message from q cannot be in List$_p(vu)$. Every List$_p(vu')$ for *vu'* constructed prior to *vu*, is closed. If none of these closed lists contains m_q (not *ec1*), then q crashed or disconnected before sending m_q and $recd_p(m_q)$ is evaluated to be false.

3.5. Examples

We illustrate the working of the view recording and installation procedures through examples. The first one is based on Figure 3. We assume that Cview$_p$ is unique and *view-number*$_p$ = k; also that p, q, and j remain connected and functioning, and hence VM$_j$ also constructs *vu1*, *vu2*, and *vu3* as shown in the figure. Let us define $\Phi(CVu) = \lfloor(|CVu|/2)\rfloor+1$.

Suppose that r1 crashes after multicasting its *State* message *SMsg*(*vu1*) and *Recorded* message *RMsg*(*vu1*). CM_c, c = p, q, or j, will find their respective recording and installation conditions being met, and install *vu1* = {p, q, r1, r2, j} as their (k+1)th Cview. Following the installation of *vu1*, CM_c delivers messages in $List_c$(*vu1*) to c which will be identical for every c. When CM_c has $head_c$ = *vu2*, no g-failure is suspected, as $head_c$ contains a majority of components in the current $Cview_c$ = *vu1*. Since $head_c$ has no joiner, no recording condition needs to be met. Since p, q, and j remain connected and functioning, CM_c will find the installation condition being met, install *vu2* as the (k+1)th Cview, and deliver messages in $List_c$(*vu2*) to c. Then, CM_c will install *vu3* as the (k+3)th Cview and deliver messages in $List_c$(*vu3*). This example shows that when VM_c and $VM_{c'}$ construct an identical sequence of Mviews, CM_c and $CM_{c'}$ behave identically; they also deliver an identical set of messages between two consecutive Cviews they install.

Suppose that r1 and r2 crash before multicasting their *State* message *SMsg*(*vu1*), then CM_c, c = p, q, or j, will find the recording conditions not being met for $head_c$ = *vu1*. CM_p and CM_q will proceed to execute the configuration protocol, while CM_j remains with no change in its *status* (= *spare*) and *mode* (= *waiting*).

Example with Concurrent Mviews

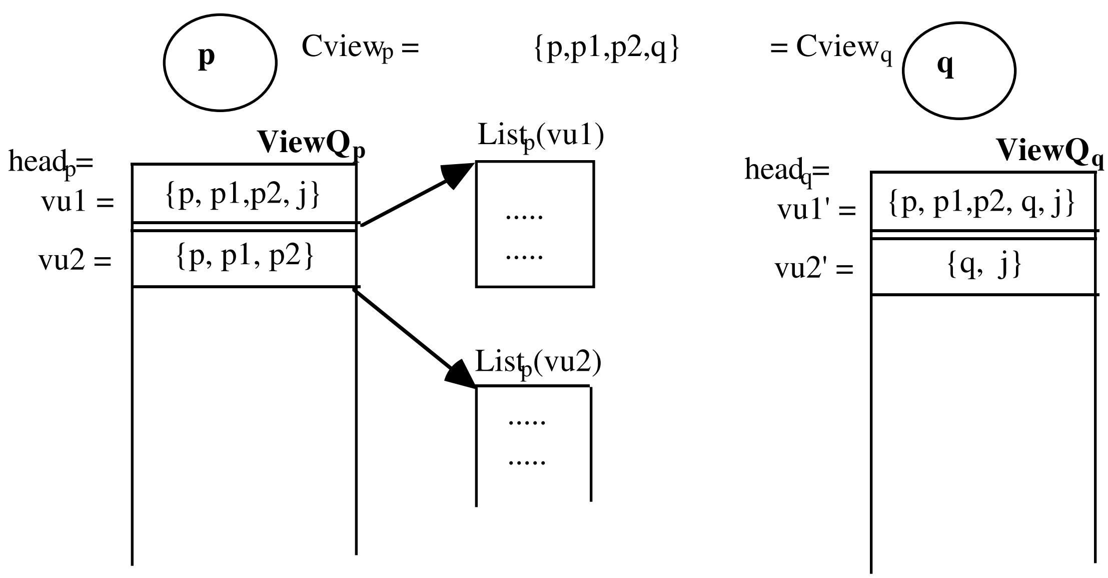

Figure 4. Concurrent and overlapping head views.

The second example is based on Figure 4 and illustrates scenarios that lead CM_p of a member p to dequeue $head_p$ without recording it, and CM_j of joiner j to execute the configuration protocol with $Cview_j$ = *null*. As in the previous example, we will assume that $Cview_p$ is unique and *view-number*$_p$ = k. The figure shows component j attempting to join the group {p, p1, p2, q}, and VM_p and VM_q reaching different view agreement due to the subsequent detachment of {p, p1, p2} from {q, j}. Let us suppose that p, p1, and p2 remain connected to each other, and so do q and j. Let c denote p, p1, or p2, and c' denote q or j. Every VM_c constructs *vu1* and *vu2* shown for p in the figure, and every $VM_{c'}$ constructs *vu1*' and *vu2*' shown for q. Given that VM_c and $VM_{c'}$ have constructed non-identical, overlapping *vu1* and *vu1'*, they must subsequently construct *vu2* and *vu2'* respectively, due to the view agreement property (see subsection 2.1.1). Let $\Phi(CVu)$ be defined to be 1, that is, a joiner can compute the component state by receiving a single *State* message.

When CM_c has $head_c = vu1$, it will find *rc1* met; but it will not receive $RMsg_j(head_c)$ from CM_j and will find *rc2* not being met. Hence, it dequeues $head_c$, and delivers messages in $List_c(vu1)$ when $Cview_c$ is still {p, p1, p2, q}. Note that $List_c(vu1)$ will not contain any application message from j as j does not yet consider itself as a member. Thus, the view-message integrity property (of subsection 2.1.1) is preserved by CM_c. Since $\Phi(CVu) = 1$, both CM_q and CM_j will find for $head_{c'} = vu1'$ the recording conditions being met, but not the installation conditions. They will proceed to execute the reconfiguration protocol, with $Cview_q$ unchanged, $Sview_q = vu1' = Sview_j$, and $Cview_j = null$.

4. Reconfiguration after a g-failure

The configuration protocol presented here meets *Requirement 2* stated in §2.5: following a g-failure, a unique set of functioning and connected components is formed to become the (first post-g-failure) master subgroup. These components restart the group operations with an identical Cview called the *restart-view*. In our protocol, the *restart-view* is taken to be either the *last* view or an Sview *SVu* that is later than the *last*, the latter being the case if and only if a majority of *last* components had recorded[1] *SVu* before the g-failure occurred. This invariant qualifies the *restart-view* to be unique and ensures the continuity in the numbering of Cviews despite a g-failure. The master subgroup is guaranteed unique by ensuring that it contains a majority of *last*. The rationale behind the formation of the master subgroup is briefly described first.

4.1. The Rationale

Let R be a set of components that get connected after a g-failure. To achieve reconfiguration, it needs to be determined whether a subset of components in R can become the master subgroup. Let $Sview_r$ and $Cview_r$ denote the Sview and the Cview of a component r in R, respectively. (If r is recovering from a crash it obtains $Cview_r$ and $Sview_r$ from its stable store.) Let *presumed_last* be the latest Cview among the $Cview_r$ of all r in R: for every r in R, either *presumed_last* = $Cview_r$ or *presumed_last* » $Cview_r$. By definition, *presumed_last* is either the *last* Cview or some Cview installed prior to the *last*.

Let us consider the Sviews recorded by the components of *presumed_last* (not just those in *presumed_last* ∩ R). One of the following three (mutually exclusive) situations must exist:

(i) A majority of *presumed_last* components recorded (at some time in the past) an identical Sview that is later than the *presumed_last*.

(ii) A majority of *presumed_last* components never recorded an Sview that is later than the *presumed_last*.

[1]No *last* component could have installed *SVu*; otherwise the installed version of *SVu* would be the *last* which, by definition, is the latest Cview installed by a component.

(iii) Neither 1 nor 2. That is, the number of *presumed_last* components that recorded an Sview that is later than the *presumed_last*, is at most |*presumed_last*|/2; similarly, the number of *presumed_last* components that never recorded an Sview that is later than the *presumed_last*, is at most |*presumed_last*|/2.

Let us first consider situation (1). Suppose that R contains (a) more than |*presumed_last*|/2 components with Sview = *presumed_last*+ and Cview = *presumed_last*, and (b) a majority subset of *presumed_last*+. We claim that if (a) and (b) are met, *restart-view* = *presumed_last*+. The (simple) proof is by contradiction.

Proof: Suppose that (a) and (b) are met but *presumed_last*+ ≠ *restart-view*. Meeting of (a) implies that a majority of components in *presumed_last* have recorded *presumed_last*+. By the definition of *restart-view*, if *restart-view* ≠ *presumed_last*+, then *restart-view* » *presumed_last*+. For this to be true, a majority of components in *presumed_last*+ must have installed *presumed_last*+ as their Cview and then must have proceeded to record an Sview *presumed_last*++ (say), *presumed_last*++ » *presumed_last*+. None of these components that installed *presumed_last*+ as their Cview, can be in R, as per the way *presumed_last* is computed. This means that (b) cannot be true - a contradiction.

Thus, when (a) and (b) are met, *presumed_last*+ becomes the *restart-view* and R∩*presumed_last*+ consider themselves to be the master subgroup.

In both situations (2) and (3), a majority of *presumed_last* have not recorded a progressive Sview that is later than *presumed_last*; therefore *presumed_last* must be the *last*, and also the *restart-view*. To deduce the existence of (2), R must contain more than |*presumed_last*|/2 components with Sview not later than *presumed_last*; and for (3) R must contain all *presumed_last* components.

Observe that deducing which one of the three situations exists, requires R to contain at least a majority of *presumed_last* components with appropriate Sviews, or all of them in the third situation. So, it is possible that a given R does not meet this requirement. In that case, the attempt to form the master subgroup with R is given up, and the recovery and reconnection of more number of components need to be awaited.

4.2. The Protocol

The protocol is made up of five steps:

Step 0. CM_p sets $mode_p$ to *reconfiguration* and waits for p to be connected with other components, i.e., for $viewQ_p$ to become non-empty. Say, $ViewQ_p$ becomes non-empty and R = $head_p$. (Note: the first Mview in $ViewQ_p$ is only copied into R, not dequeued.) The remaining four steps are done using R.

Step 1. Send {$Sview_p$, $Cview_p$} to every r in R (including itself);
Receive {$SviewRecd_r$, $CviewRecd_r$} from every r in R;

Step 2. Determine the *presumed_last* to be the latest $CviewRecd_r$;

Step 3. Determine the *restart-view* if possible; if not possible dequeue R from $viewQ_p$, discard R and go to step 0.

Step 4. components of R ∩ *restart-view*:
install *restart-view* and resume group services;

components of R - *restart-view*:
become *spares;*

Each step is described in detail in the subsections below.

4.2.1. Step 1: View Exchange

CM_p multicasts a message *msg*($Sview_p$, $Cview_p$, $mode_p$) containing $Sview_p$, $Cview_p$ and $mode_p$. It then evaluates the predicate $recd_p$(msg_r($SviewRecd_r$, $CviewRecd_r$, $mode_r$)) for every $r \in R$. If this predicate is true for a given r, CM_p then checks whether $CviewRecd_r$ » $Cview_p$ and $mode_r$ = *normal*. If this condition is true, an exception *Walked-Over* is raised indicating that p has been slow in recovery, during which time the group is reconfigured without p. This exception is handled by making p a spare and exiting the execution of the protocol. If the predicate is false for some r, then working with R is given up: terminate the execution with R, dequeue R from $viewQ_p$, and go to step 0. The pseudo-code for step 1 is given below:

multicast *msg*($Sview_p$, $Cview_p$, $mode_p$) to all r in R;
evaluate $recd_p$(msg_r($SviewRecd_r$, $CviewRecd_r$, $mode_r$)) for every r in R;}
catch (*Walked-Over*): { write atomically: {$Sview_p$:= *null*; $status_p$:= *spare*;
$mode_p$:= *waiting*;}
exit; }
if ($\exists r \in$ R: $\neg recd_p$(msg_r($SviewRecd_r$, $CviewRecd_r$, $mode_r$)))
then {give up on R;}

4.2.2. Step 2: Determine presumed_last

presumed_last is computed to be the latest non-null view among the received Cviews. If a majority of *presumed_last* is not in R, then the execution with current R is given up.

{ *presumed_last* := $CviewRecd_r$ of some $r \in$ R: (*presumed_last* $\neq$ *null)* $\wedge$
($\forall$ r' $\in$ R: *presumed_last* = $CviewRecd_{r'}$ $\vee$ *presumed_last* »$CviewRecd_{r'}$);
if (| *presumed_last* $\cap$ R| $\leq$ (| *presumed_last* |/2) then { give up on R;}
}

Note that by requiring that *presumed-last* be non-null, an R of only spare components with *mode* = *waiting* are prohibited from forming the *master subgroup*.

4.2.3. Step 3: Attempt to Determine restart-view

CM_p divides the components in *presumed_last* $\cap$ R into non-overlapping subsets, called candidate sets and denoted as CS_v, $v \geq 0$, based on the components' Sview. Let *presumed_last*$+_i$, $i \geq 1$, be an Sview[2] that is later than *presumed_last*. Each CS_v, $v \geq 1$,

---

[2] *presumed_last*+ need not be unique after a g-failure; different *presumed_last* components could have recorded different progressive Sviews, due to their VM modules concluding view agreement at different points. Let, for example, *last* = {1,2,3,4,5}. Let C5 crash and VM of C4 reach agreement on, and deliver {1,2,3,4}. If VMs of C1, C2, and C3 suspect C4 before they reach agreement on {1,2,3,4}, they will reach agreement straightaway on {1,2,3}. If a g-failure occurs after CMs have recorded the delivered views, $Sview_4$ = {1,2,3,4} (say, *presumed_last*$+_1$), and $Sview_1$ = $Sview_2$ = $Sview_3$ = {1,2,3} (say, *presumed_last*$+_2$).

contains the components in *presumed_last* $\cap$ R whose $SviewRecd_r$ = *presumed_last*$+_v$; CS_0 contains those components in *presumed_last* $\cap$ R whose $SviewRecd_r$ is not later than *presumed_last*. The code for this third step is given below.

```
(1)     if (∃ CS_v: v ≥1 ∧ CS_v ∈ M_SETS(presumed_last)
                        ∧ presumed_last+_v ∩ R ∈ M_SETS(presumed_last+_v) ) then
            {restart-view := presumed_last+_v;}
                        // existence of situation (1) is deduced
(2)     else if (CS_0 ∈ M_SETS(presumed_last) then
            {restart-view := presumed_last;}
                        // existence of situation (2) is deduced
(3)     else if (presumed_last ⊆ R) then
            {restart-view := presumed_last;}
                        // existence of situation (3) is deduced
        else {give up on R;}
```

4.2.4. Step 4: Commencing Group Operations

Any p that is not in the *restart-view* becomes a spare, otherwise CM_p updates view information in its stable store. The pseudo code is as follows:

```
{       if (p ∉ restart-view) then { write atomically:
                                      {Sview_p, Cview_p := null; status_p := spare;
                                      mode_p := waiting; view-number_p := -1;}
                                 exit; }
        write atomically:
            {view-number_p:= view-number(restart-view);
            Sview_p, Cview_p:= restart-view; status_p := member;
            mode_p := normal; }
}
```

4.2 Examples

We explain the working of the protocol with the help of examples and by referring to the evolution of Sviews depicted in Table 1. For simplicity, assume that all g-failures considered in this discussion are caused by node crashes only, and partitions may occur only when the group is being reconfigured after a g-failure.

Table 1 depicts a possible sequence of membership changes for a group of size 5 and adopts the following style to represent the state of the view installation: an $Sview_p$ in normal font indicates that it has been installed as the $Cview_p$; an $Sview_p$ that is yet to be installed is written in mixed fonts: survivors (from the current $Cview_p$ into this $Sview_p$) in normal font, joiners in italics and excluded components (i.e., the ones that are in the current $Cview_p$ but not in the $Sview_p$) in bold. The superscript of an $Sview_p$ indicates its view-number.

Stage No	Sview of C1, C2	Sview of C3	Sview of C4, C5	Sview of C6, C7	Description
1	$\{1,2,3,4,5\}^0$	$\{1,2,3,4,5\}^0$	$\{1,2,3,4,5\}^0$	---	group initialised, n=5; C6 and C7 are spares
2	{1,2,3, **4,5**}^{1}	$\{1,2,3,4,5\}^0$	$\{1,2,3,4,5\}^0$	---	C4 and C5 crash; CM1 and CM2 record their exclusion first
3	{1,2,3, **4,5**}^{1}	{1,2,3, **4,5**}^{1}	$\{1,2,3,4,5\}^0$	---	Slow CM3 records Sview(1)
4	$\{1,2,3\}^1$	{1,2,3, **4,5**}^{1}	$\{1,2,3,4,5\}^0$	---	CM1 & CM2 install Sview(1)
5	$\{1,2,3\}^1$	$\{1,2,3\}^1$	$\{1,2,3,4,5\}^0$	---	CM3 install its Sview(1)
6	{1,2,3 ,*6,7*}^{2}	{1,2,3 ,*6,7*}^{2}	$\{1,2,3,4,5\}^0$	{1,2,3 ,*6,7*}^{2}	C6 & C7 join; all active CM record Sview(2) = {1,2,3,6,7}
7	{1,2,3 ,*6,7*}^{2}	$\{1,2,3,6,7\}^2$	$\{1,2,3,4,5\}^0$	$\{1,2,3,6,7\}^2$	CM3, CM6, and CM7 install Sview(2)
8	{1,2,3 ,*6,7*}^{2}	$\{3,6,7\}^3$	$\{1,2,3,4,5\}^0$	$\{3,6,7\}^3$	C1, C2 crash before installing Mview(2); CM3, CM6, CM7 record and then install {3,6,7}

Table 1. An evolution of Sviews.

The group is initially formed with {C1, C2, C3, C4, C5}. At the end of stage 1, each member has $\{1,2,3,4,5\}^0$ as its (initial) Sview in stable store; this is also the Cview. At the end of stage 2, CM_1 and CM_2 have recorded Sview(1) which cannot be installed now as $\{1,2\} \notin$ M_SETS(Sview(0)). The situation changes after stage 3, and CM_1 and CM_2 install Sview(1) in stage 4. In stage 6, the spares C6 and C7 join the group: CM1, CM2, CM3, CM6, and CM7 record the Sview {1,2,3,*6,7*}. In the next stage, C3, C6, and C7 install the Sview, since all components in the old view {1,2,3} are known to have recorded {1,2,3,*6,7*}. But C1 and C2 crash before they could install the recorded view. In stage 8, C3, C6, and C7 install the Cview without C1 and C2. According to Table 1, {1,2,3,*6,7*} » {1,2,3} and {1,2,3,*6,7*} » {1,2,3,4,5}. Further, *last* is $\{1,2,3,4,5\}^0$ until stage 3, $\{1,2,3\}^1$ in stages 4, 5, and 6, $\{1,2,3,6,7\}^2$ in stage 7, and $\{3,6,7\}^3$ in stage 8.

Example 0: This example shows that the protocol is safe in not allowing more than one master subgroup to be formed after a g-failure. Let C1 and C2 recover and get connected after stage 8 but remain partitioned from other components. So, R = {C1, C2}. Both C1 and C2 will estimate *presumed_last* to be $\{1,2,3\}^1$. Since {C3, C6, C7} is already functioning as the group, another master subgroup should not be allowed to emerge from R even though R contains a majority subset of *presumed_last*. Components of R will find that they have an identical (progressive) Sview {1,2,3,*6,7*}^{2} » *presumed_last*, and R does not contain a majority subset of {1,2,3,*6,7*}^{2}. So, none of the conditions in Step 3 of the protocol is satisfied and R will be given up.

Example 1: This exemplifies the behaviour of the protocol under the situation 1 mentioned in section 4.1. Suppose that a g-failure occurs immediately after stage 6.

Here, *last* is $\{1,2,3\}^1$ and all of the *last* components have recorded $\{1,2,3,\mathit{6,7}\}^2$. So, *restart-view* is $\{1,2,3,6,7\}^2$. Say, R = {C1, C2, C4, C6}. By step 3.1 of the protocol, each component r in R determines *restart-view* to be $\{1,2,3,6,7\}^2$. Finding itself not in *restart-view*, C4 will exit the protocol and join the pool of spares. The others in (R ∩ *restart-view*) install *restart-view* as their Cview and resume normal group services. Note that the view R is still the at head of every $ViewQ_r$, $r \in R$. Upon detecting ViewQ not empty, CMs of (R ∩ *restart-view*) will execute the view installation protocol as members and CM of C4 as a joiner. Assuming no further failures or disconnections, C1, C2, C4, and C6 will get install R as the Cview.

Example 2 considers situation 2 where a majority of *presumed-last* have not recorded an Sview that is later than *presumed-last*. Say, a g-failure occurs at the end of stage 1. *last* = $\{1,2,3,4,5\}^0$. Since no *last* component has recorded a later Sview, *restart-view* is also $\{1,2,3,4,5\}^0$; further, since all *last* components have identical Sview, an R that contains *any* three (majority subset) of the *last* members will lead to the master subgroup. Say, R = {C3, C4, C5}. Assuming that R remains connected for long, each CM of R computes *presumed-last* to be $\{1,2,3,4,5\}^0$, i.e., *last* itself. Next, each CM of R forms CS_0 = R and decides in step 3.2 of the protocol the *restart-view* to be *presumed-last* = $\{1,2,3,4,5\}^0$. After (re-)installing *restart-view* as their Cview, CMs of C3, C4, and C5 subsequently install R as their next Cview = $\{3,4,5\}^1$.

Say, after CMs of C3, C4, and C5 have installed $\{3, 4, 5\}^1$ in the above scenario, let C1 and C2 recover and reconnect with C3, C4, and C5. While CMs of C1 and C2 execute the reconfiguration protocol with R = {1,2,3,4,5}, CMs of C3, C4, and C5 will execute view installation protocol for the delivered view $\{\mathit{1}, \mathit{2}, 3, 4, 5\}^2$ in which C1 and C2 are regarded as joiners. This conflict gets resolved very easily: CMs of C3, C4, and C5 expect CMs of C1 and C2 to send *recorded* messages but instead find messages of configuration protocol. They would then respond by sending its *mode* and Cview to CM_1 and CM_2 which would get *Walked_Over* exception, become *spares*, and then start executing the view installation protocol as joiners, with the head of their ViewQ (still) having R = {1,2,3,4,5}.

In **example 3**, we illustrate the need for R to contain *all* the *last* members in certain circumstances. Let a g-failure occur at the end of stage 2. The *last* view here is $\{1,2,3,4,5\}^0$ which is also the Cview of every member component. We will assume R = {C1, C2, C4, C5}. The *presumed_last* is the same as *last* = $\{1,2,3,4,5\}^0$. Each component of R knows that a minority of *presumed_last* (i.e. two) have not recorded an Sview that is later than *presumed_last*; and also that only a minority of *presumed_last* (i.e. two again) are known to have recorded an Sview $\{1,2,3,\mathbf{4},\mathbf{5}\}^1$ that is later than *presumed_last*. When Sview of C3 is $\{1,2,3,4,5\}^0$ (as it is now), then *restart-view* becomes $\{1,2,3,4,5\}^0$. If Sview of C3 had been $\{1,2,3,\mathbf{4},\mathbf{5}\}^1$ (as it is at the end of Stage 3), then *restart-view* becomes $\{1,2,3\}^1$. Hence determining the *restart-view* requires that the components of R know the Sview of C3. Here, R is given up in step 3.3 of the protocol which requires R to contain *presumed_last* when neither the situation 1 nor 2 is known to prevail.

5. Related Work

Using the traditional, 2-Phase Commit (2PC) protocol [Gray78] for atomically updating membership-related information, [Jajodia90] maintains at most one distinguished partition in a replicated database system. Our CM subsystem also uses a variation of this traditional 2PC for Cview installation and the variations are inspired by our requirements and efficiency. In the traditional 2PC way of installing Cviews, the coordinator - a deterministically chosen member in the new Cview - would initiate the second (view-installation) phase after learning during the first phase that *every* component of the new Cview has recorded the view. Note that while view installation is in progress, delivery of application messages is put on hold to maintain view-synchrony. Since we only require that at least a majority subset, not necessarily all, of the current Cview install the next Cview, we can speed up the view-installation by having the coordinator initiate the second phase as soon as a *majority* of the current Cview and all joiners (if any) in the new Cview have recorded the new view i.e. as soon as the installation conditions of section 3.3 are met. Further, the coordinator based execution of traditional 2PC are susceptible to co-ordinator crashes. We eliminate this weakness by executing our version of 2PC in a decentralised manner where every component checks installation conditions.

Since we use a 2PC protocol for view installations, the configuration protocol cannot be non-blocking. This blocking can be removed by using a 3PC protocol [Skeen81]. The protocol of [Dolev97] employs the principles of an extended 3PC protocol [Keid95] and builds a unique master subgroup after a g-failure. Not surprisingly, our architecture is remarkably similar to theirs. They differ from our protocol in one other major aspect: a component can have, and may have to exchange, more than one Sview; so more stable information needs to be maintained and message size is increased. Obviously these features of [Dolev97] increase the overhead of the protocol. The advantage, on the other hand, is that a reconnected set need only contain a particular majority subset of *last*, never all *last* components as we would require in certain cases when a g-failure occurs during view update (see example 3 of section 4.2).

The primary partition membership service in [Birman87, Ricciardi91, Mishra91] make the assumption that a majority of components in the Cview do not suspect each other and that a functioning component is rarely detected as failed. This assumption may not hold true during periods of network instability caused for example by bursty traffic or network congestion. This instability can lead to incorrect failure detections which in turn can lead to g-failures. In these circumstances, our CM subsystem (also [Dolev97]) can provide recovery from g-failures once the network traffic stabilises.

[Chandra96] establishes the weakest failure detector (denoted as ◊S/◊W) for solving the consensus problem. Using this consensus protocol, a (primary-partition) membership service is designed [Malloth95] and implemented [Felber98]. This membership service can construct a totally ordered sequence of views, with a majority of each view surviving into the next view. It blocks from delivering a new view during the periods of g-failures (i.e., when a connected majority does not exist) and the blocking is released as soon as the requirements of ◊S are realised. Does this mean a ◊S based membership service provides recovery from g-failures for the weakest system requirements of ◊S? The answer appears to be no. The first view which the ◊S based membership service constructs after a g-failure, is what we call the *restart-view* (see requirement 2 in §2.5). Determining the *restart-view* does not necessarily mean that the new master subgroup exists to restart the group services. To see this, consider the following example. Let the current view be {p, q, r, s, t} with view-number = k, and s and t crash before a g-failure occurs. With the ◊S based membership service, it is possible for R = {r, s, t} to reconnect and decide that the (k+1)th view is {p, q, r}.

Though the *restart-view* is now known, the group operations cannnot be resumed as R does not contain a majority subset of the *restart-view*. (Permitting any subset of R to form the master subgroup will lead to two concurrent master subgroups if {p, q} is operating in a seperate partition.) Group re-configuration with R must therefore wait for either p or q to recover/reconnect. Our approach is different in that the *restart-view* cannot be determined until the reconnected set (R) contains at least a majority of the *restart view* itself (see section 4.1); in other words, determining the *restart-view* straight leads to the re-formation of the group (barring the occurrence of further g-failures). As a future work, we intend to compare these two approaches further in a more detailed manner.

6. Conclusions

Group failures can occur even in the absence of any physical failures, and be caused by sudden bursts in message traffic with potentials to lead to virtual partitions. We have designed and implemented a configuration management subsystem which can provide automatic recovery from group failures, once the real/virtual partitions disappear and components recover. Our system employs a variation of two-phase commit protocol for view updates. Consequently, the recovery provided is subject to blocking. On the other hand, it is efficient in terms of message size, message rounds and use of stable store, during both normal operations and reconfiguration after a group failure; it costs only one extra message round to update views in the normal, failure-free periods. This low, failure-free overhead makes our system particularly suited to soft real-time systems where it can be incorporated in the manner proposed in [Hurfin98].

Acknowledgements

The work described here has been supported in part by Hewlett-Packard Laboratories, Bristol. Roger Flemming and Paul Harry from Hewlett-Packard Laboratories provided facilities and resources for implementing our reconfiguration system over Somersault. Discussions with Andre Schiper over ◊S-based consensus protocol were useful in clarifying many subtle issues.

References

[Amir92] Y. Amir, Dolev, D., Kramer, S., and Malki, D., "Membership Algorithm for Multicast Communication Groups", Proc. of 6th Intl. Workshop on Dist. Algorithms, pp 292-312, November 1992.

[Babaoglu95] O.Babaoglu, R. Davoli, and A Montresor, "Group Membership and View Synchrony in Partitionable Asynchronous Distributed Systems: Specifications", Technical Report UBLCS-95-18, Dept. of Computer Science, University of Bologna, Italy, Nov 1995.

[Babaoglu97] O.Babaoglu, A. Bartoli, and G Dini, "Enriched View Synchrony: A Programming Paradigm for Partitionable Asynchronous Distributed Systems", IEEE ToCS, 46(6), June 1997, pp.642-658.

[Birman87] K.Birman and T. Joseph, "Exploiting virtual synchrony in distributed systems", Proc. of 11th ACM Symposium on Operating System Principles, Austin, November 1987, pp. 123-138.

[Black97] D. Black, P. Ezhilchelvan and S.K. Shrivastava, “Determining the Last Membership of a Process Group after a Total Failure”, Tech. Report No. 602, Dept. of Computing Science, University of Newcastle upon Tyne.

[Chandra96] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The weakest Failure Detector for Solving Consensus", JACM, 43(4), pp. 685 - 722, July 1996.

[Ezhil95] P. Ezhilchelvan, R. Macedo and S. K. Shrivastava, “Newtop: a fault-tolerant group communication protocol”, 15th IEEE Intl. Conf. on Distributed Computing Systems, Vancouver, May 1995, pp. 296-306.

[Felber98] P Felber, R Guerraoui and A Schiper, "The implementation of CORBA Object service", Theory and Prctice of Object Systems, Vol. 4, No. 2, 1998, pp. 93-105.

[Gray78] J N Gray. Notes on Database Operating Systems. In *Operating Systems: An Advanced Course*, Lecture Notes In Computer Science, Vol 60, pp. 393-481. Springer Verlag, Berlin, 1978.

[Guerr95] R Guerraoui, M Larrea and A Schiper. Non Blocking Atomic Commitment with an Unreliable Failure Detector. Proceedings of *IEEE Symposium on Reliable Distributed Systems* (SRDS), Bad Neunhar, Germany, September 1995, pp. 41-51.

[Hurfin98] M. Hurfin and M. Raynal, "Asynchronous Protocols to Meet Real-Time Constraints: Is It Really Sensible? How to Proceed?", Proc. of 1st Int. Symp. on Object-Oriented Real-Time Distributed Computing, (ISORC98) pp. 290-297, April 98.

[Jajodia90] S Jajodia and D Mutchler. Dynamic Voting Algorithms for Maintaining the Consistency of a Replicated Database. *ACM Transactions on Database Systems*, Vol 15, No 2, June 1990, pp. 230-280

[Keid95] I Keidar and D Dolev. Increasing the Resilience of Distributed and Replicated Database Systems. *Journal of Computer and System Sciences* (JCSS). 1995.

[Lotem97] E Y Lotem, I Keidar and D Dolev. Dynamic Voting for Consistent Primary Components. Proceedings of ACM Symposium on Principles of Distributed Computing (PODC), pp. 63-71, 1997.

[Malloth95] C Malloth and A Schiper, "Virtually Synchronous Communication in Large Scale Networks", BROADCAST Third Year Report, Vol 3, Chapter 2, July 1995. (Anonymous ftp from broadcast.esprit.ec.org in directory projects/broadcast/reports)

[Melliar-Smith91] P. M. Melliar-Smith, Moser L.E., and Agarwala, V., "Membership Algorithms for Asynchronous Distributed Systems", Proc. of 12th Intl. Conf. on Distributed Comp. Systems, pp. 480-488, May 1991.

[Mishra91] S. Mishra, L. Peterson and R. Schlichting, “A membership Protocol Based on Partial Order”, Proc. IFIP Conf. on Dependable Computing For Critical Applications, Tuscon, Feb. 1991, pp 137-145.

[Moser96] L.E. Moser, P.M. Melliar-Smith et al, “Totem: a Fault-tolerant multicast group communication system”, CACM, 39 (4), April 1996, pp. 54-63.

[Murray97] P. Murray, R. Flemming, P. Harry and P. Vickers, “Somersault software fault-tolerance”, Hewlett-Packard Technical Report, 1997.

[Ricciardi91] A. Ricciardi and K P Birman, "Using Process Groups to Implement Failure Detection in Asynchronous Environments", In Proceedings of ACM symposium on PoDC, pp. 480-488, May 91.

[Schiper94] A Schiper and A Sandoz, "Primary-Partition Virtually Synchronous Communication is Harder Than Consensus", Proc. of the 8th International Workshop on Distributed Algorithms (WDAG-94), Sept. 94, LNCS 857, Springer Verlag. (Also in BROADCAST Second Year Report, Vol 2, October 1994).

[Skeen 81] D. Skeen, "Non-Blocking Commit Protocols", ACM SIGMOD, pp.133 - 142, 1981.