# View-based Reconfiguration of Replicated Typed Data

Abdelsalam Heddaya*
Computer Science Department
Boston University
heddaya@cs.bu.edu

November 23, 1990

Technical Report BU-CS-90-001

Draft–Draft–Draft
Do not distribute

LaTeX'd on November 26, 1990

**Abstract**

Distributed systems suffer from various kinds of failures, of which communication failures are particularly disruptive. Data replication methods based on *voting* have been used to maintain the availability of distributed computing services in the presence of undetected failures. We present a new reconfiguration protocol for replicated data that adapts to *detected* failures and repairs by: (1) changing the set of sites responsible for storing the data object, and (2) adjusting the individual event *quorums* (an event quorum is a set of of sites whose affirmative votes are sufficient to complete the event). According to this protocol, every site makes decisions based purely on information that is either local to the site, or available within the set of reachable and running sites. Thus, our protocol is suitable for use in systems that may suffer from both site crashes and communication failures that may lead to *network partitions*.

To prove the correctness of our protocol, we formulate a new way to reason about the flow of information achievable for replicated objects under communication failures.
**Keywords:** replicated data, reconfiguration, availability, reliability, network partition failures, distributed systems.

*This work was supported in part by the National Science Foundation under grant IRI-8910195, and in part by the Defense Advanced Research Projects Agency under grant number N00039-83-C-0105. Address: Computer Science Department, Boston University, 111 Cummington Street, Boston, MA 02215. Internet: *heddaya@cs.bu.edu*. Phone: (617) 353-8922; Fax: (617) 353-8100.

# 1 Introduction

The advent of distributed systems increased the complexity of managing shared computing services because of the high probability of independent site failures, among other factors. For example, if the machine hosting a file server fails, then the entire file system becomes unavailable to the sites that are still connected and running. One way to maintain high *availability* of services despite such failures is by *replication*. This paper addresses the problem of reconfiguring replicated objects to match changing circumstances of failure and load distribution.

We model a distributed service as an object belonging to an abstract data type (ADT). That is, objects can be accessed only through the operations defined by the type. An *event* is a time-stamped operation invocation, together with its arguments and results. More precisely, the tuple

$$\langle\ s,\ o,\ \text{args},\ \text{results}\ \rangle,$$

represents the event of executing operation $o$ with the given arguments and results, at timestamp $s$. The state of an object can be represented by the set of all events experienced by it. This definition of an event enables us to use a non-traditional method of replication. Traditionally, replicating a data object consists of making copies of its value to be stored on different sites. Recent work suggests that replicating the individual events experienced by the object, instead of its entire value, can be a practical way to support higher concurrency and a wider range of availability characteristics [9, 12, 22].

We assume that a distributed atomic transaction facility is available as a sub-system in our model. For our purposes, a transaction is a flat sequence of events that is atomic with respect to concurrency and failures [8]. General purpose transaction systems that provide such properties abound, such as Camelot [23, 24] and Argus [19, 18]. We structure a replicated data service as a set of *representatives* collectively responsible for storing the object state, and an arbitrary number of *front-ends* consisting of pure code, that mediate the interaction between clients and representatives.

In this paper, we describe a new reconfiguration method for objects that are replicated using *general quorum consensus* (GQC) [11, 12]. Under general quorum consensus, a (typed) data object is stored at a set of representatives, each of which keeps a *partial history* of the events that alter the object's state. Client transactions issue operations on objects of a particular type by sending them to a front-end server capable of operating on objects belonging to this type. The front-end queries an *initial quorum* of the representatives for their local histories and merges them together to produce a history that contains all the past events relevant to the requested operation. This history is then applied to a suitable initial value of the object to obtain a current value on which the operation can be carried out. After the results are produced, the front-end records the merged history together with the new event, at any *final quorum* of the object's representatives. The choice of initial and final quorums is constrained so that any two events that depend on each other are guaranteed to observe each other's effects.

The motivations to reconfigure range from enhancing the reliability (availability and recoverability) of the replicated object in the face of multiple failures, to managing the allocation of system resources by its administrators. As discussed further in the paragraphs below, the availability of an object decreases rapidly if the effects of failures are allowed to accumulate. System administrators need to reconfigure system services in order to support scheduled maintenance, balance the load, or match organizational changes.

The *availability* of a particular event on a data object replicated using the GQC protocol is

governed by the availability of the quorums of sites whose consensus suffices to complete the event. If at least one quorum for an event is operational and connected to a client, then we say that this event is available for the client. Thus, the availability of an event can be measured by the *probability that at least one of its quorums is up and connected.* We consider an object as a whole to be available if all of its events are available.

Consider the availability of an event after a *sequence* of failures. Every failure reduces the number of the event's quorums that can achieve consensus, and hence, reduces its availability. This raises the question: is it possible to avoid the accumulation of losses in availability? The answer, from previous work [3, 13, 17], is "yes." Our protocol performs the same function, with the added advantage that it enables such reconfiguration in a completely dynamic manner, whereas previous works [3, 13] always kept some aspect of the configuration static[1]. A dynamic protocol generally provides higher availability and facilitates the administration of the replicated object, since otherwise, the system would have to be quiesced (brought out of service) in order to change the static components of the configuration.

Our protocol guarantees *one-copy serializability* [4] of the behavior of objects that obey the protocol, despite the configuration changes that might take place. That is, the representatives and front-ends permit only computations that are equivalent to those of a system that contains no concurrency, failures, replication, or reconfiguration. In section 3 we discuss further the notion of one-copy serializability.

We specify our protocol in the form of *constraints* on the acceptable new configurations under a given failure situation, leaving the choice of the actual new configuration open. In general, our protocol favors transactions whose availability is bound by the initial quorums, rather than those that are bound by final quorums. In other words, we make it easier to shrink the initial quorums than the final quorums, and this means that transactions that cannot run because they cannot collect an initial quorum may be able to run if appropriate reconfiguration is done (reflecting an increase in availability). We can achieve a similar increase in availability for final quorums, but only when (very roughly) a majority of representatives agree to the reconfiguration.

As a bonus that we reap from our specific approach to replication and reconfiguration we can make special classes of transactions (read-only and blind-write) available under circumstances that would normally render them completely unavailable, even with reconfiguration [10]. Read-only transactions have been optimized in this manner in the past for non-replicated data (see, for example, [5, 14, 15, 26]), but blind-write transactions have not—to the extent of the author's knowledge. Blind-write events acquire special importance in an event-based system because many of the events that would normally be update events in value-based models turn out to be blind-writes to the event history. Blind-write events include such common operations as *enqueue*, counter *increment*, and set *insert*. The optimizations we give in [10] to handle read-only and blind-write transactions are relatively independent of our particular reconfiguration scheme. However, our scheme specifically identifies these classes of operations and provides a framework within which we can exploit them in a straightforward manner.

As a second bonus of our approach, we can treat site recovery under mild failure conditions, as a special case of reconfiguration. Traditional methods typically employ a *log* of system events, stored on a *stable storage* device, to recover the state of a repaired site in a distributed system. We observe that, as long as enough sites survive to reconstruct completely the state of the object, we can recover a site by reconfiguring the object to include the recovering site as a new representative. Thus, *under*

[1] Except for [17], but that protocol cannot tolerate transient communication failures.

*mild failure conditions, we can achieve recoverability from failure without using memory devices that survive failures.*

In the following section, we provide a general overview of our reconfiguration method. In section 3, we formally define the notion of a *view*, specify the configuration state, and relate views to event quorums. We describe the reconfiguration protocol in section 4, and establish its invariant properties and its overall correctness in section 6. We conclude this paper with a section on related work followed by a discussion of blind-write and read-only transactions, recovery, and locality issues.

## 2 Protocol Overview

The reconfiguration method that we describe in this paper achieves three goals. First, it permits sites that remain connected and running after a failure to recruit new representatives for the object, and to decommission failed or disconnected ones. Second, the quorums for events that remain available after a failure can be changed to improve event availability in the face of subsequent failures. Lastly, our protocol enables read-only and blind-write transactions to run in all partitions, even those that do not contain proper quorums, by manipulating their serialization order.

In spite of the changing configurations supported by our protocol, it maintains overall *one-copy serializability* of the replicated data object. To establish this property, we show that every event experienced by an object obeying our reconfiguration protocol observes at least one copy of every event on which it may depend. We prove that this property holds in the presence of communication partitions and site failures, as well as the associated reconfigurations performed by our protocol. In addition, we require all the transactions that run in a given partition to serialize as a block [2].

The *configuration state* that our protocol maintains consists primarily of a quorum assignment that specifies the initial and final quorums that should be used for every event. Each quorum is a set of sites whose consensus is sufficient to complete the event. Our protocol permits the front-ends to use different quorum assignments under different failure conditions. Whenever a reconfiguration is desired (for example, when a failure is detected) our protocol can be invoked to change the quorum assignments of some of the events to enhance their availability in the face of further failures. Not all quorum assignments will be acceptable, and we adopt the style of specifying the *constraints* on new quorum assignments. These constraints will typically be satisfiable by many different quorum assignments, and this freedom can be exploited to optimize whatever measures of performance or reliability are deemed appropriate by the system designer.

**Example 1 (Bank Account)** Consider a bank account that is replicated at four sites, and supports the operations *debit, credit,* and *balance*. Assume that the *debit* and *credit* operations are equivalent to the counter operations *decrement* and *increment*, respectively. The *balance* operation is a straightforward *read* of the account balance. To illustrate the possible reconfigurations that our protocol can perform, we will go through a simple scenario in which the network fails by splitting into two partitions, with two sites in each, and then one site fails in each partition.

Initially the configuration specifies, say, that any three sites of the four constitute a final quorum for each of the *debit* and *credit* operations, and that any two sites form an initial quorum that can run a *balance* operation. All the other quorums are empty. After the first failure, it becomes impossible to run *debits* or *credits* in either partition, but both partitions can still run *balance*s. After the

second failure—without reconfiguration—the *balance* operation will also become unavailable. Our protocol permits *shrinking the initial quorum* for the balance operation such that only one site is needed to constitute a quorum. This means that, after reconfiguration, the bank account will remain readable despite one more failure in each partition before it becomes completely unavailable. ∎

We adopt an approach to reconfiguration similar to El Abbadi *et al.*'s [3], in that we keep track, as part of the object configuration, of the set of representatives (called the *view*) assumed to be running and reachable. The configuration consists of the view and its corresponding quorum assignments. Whenever a change in quorum assignment is desired, or the view becomes no longer accurate, then a single protocol is invoked that installs a new configuration consisting of a new view and a new quorum assignment. The *configuration installer* has the responsibility of choosing a new quorum assignment that satisfies the constraints specified in section 4.

The view change part of our configuration installation protocol generalizes straightfowardly the view management protocol in [1], to handle arbitrary operations rather than just *read* and *write* ones. For simplicity, we describe the protocol as if there is only one object in the system. The configuration installer starts by broadcasting a message inquiring who is running and reachable. Every representative that receives this message replies with a message containing a copy of its current view, the object history and quorum assignments. The configuration installer merges the histories included in the replies, and chooses a new view and a new quorum assignment based on the information it has collected. Finally, the new configuration and the merged history are sent to all the representatives in the new view.

The configuration installation protocol runs as a single atomic transaction, using the underlying transaction system.

Our correctness condition stipulates that: *the initial quorum of an event must contain at least one copy of every preceding event that can affect it.* This condition is satisfied by General Quorum Consensus [11] in the absence of reconfiguration, so we need to show that it still holds after augmenting General Quorum Consensus with our reconfiguration protocol. We establish this property by distinguishing between events that occur in the same view and those that occur in different views. Observe that, for every preceding event that occurred in a view different from the current one, if the event's final quorum intersects the current view then it is known to *all* the representatives in the current view. The configuration installer ensures this by merging all the histories it collects and sending them to *every* representative in the new view.

If a quorum for a given event is present in the new view, the first rule stipulates that a new temporary quorum assignment can be used for the event, to be used only in the new view. Also, this rule constrains every new final quorum to contain some old final quorum. This means that initial quorums can shrink under this rule, but final quorums cannot. The second reconfiguration rule applies when the view has both a quorum and a co-quorum[2] for an event (*i.e.*, very roughly, a majority) of representatives from the previous configuration. In this case, the configuration installer is free to change the event's quorum assignment permanently. Of course, both rules also require that the new quorum assignment ensure that events that run in the new view observe each other if the depend on each other.

The configuration state is fully replicated, that is, each representative contains a complete copy of it. The configuration installer ensures that representatives in the same view always have identical

[2]This is defined rigorously below, in section 5.

copies of the configuration state. Each representative keeps two versions of the quorum assignment. One version is termed *current*, and the other *base*. The two versions are identical in the absence of failures, but may differ otherwise.

When a front-end receives an operation for execution, it uses the *current* quorum assignment for the event, obtained from any representative it can reach.[3] The only use of the *base* quorum assignment is during reconfiguration, when the configuration installer uses the most recent base quorum assignment it receives to decide which reconfiguration rule applies. Thus, the base version of the quorum assignment is used outside the view in which it is installed, while the current one is for use only within its view.

Having described the structure of the configuration state, we can restate the reconfiguration rules more precisely. The first rule applies if the view contains a base quorum for an event, in which case we say that the view *semi-covers* the event. In this case, the current quorum assignment for the event can change, under the condition that final quorums cannot shrink. If the view contains both a base quorum and a base co-quorum for an event, we say that the view *covers* the event, and the second rule applies, permitting the configuration installer to change both the base and current quorum assignments for the event. Both rules require the further constraint that conflicting events' quorums must intersect, as prescribed by regular General Quorum Consensus to guarantee correctness in the absence of reconfiguration.

The two reconfiguration rules, together with the configuration installation protocol, form the reconfiguration protocol.

# 3 Views and Events

The central correctness criterion for most replication protocols is *one copy serializability* [4]; that is, the execution of concurrent transactions on a set of replicated objects should be equivalent to some serial failure-free execution on a non-replicated database. We adopt this criterion, and reformulate it in terms of the information needed by the site executing a particular operation.

Our approach to reconfiguration requires each representative to maintain an approximation of the set of representatives with which it can communicate. We use the *view* construct to approximate the notion of a communication partition, following El Abbadi *et al.* in [1]. Note that we do not require a view to contain all sites that are connected and operational; only those that are representatives for the same object.

**Definition 1** *A* **view** $v$ *is a pair* $\langle id, reps \rangle$*, where* $v.id$ *is a unique ID, and* $v.reps$ *is the set of object representatives assumed to be connected and operational.*

*A view* $v$ *is said to* **observe** *another view* $w$*, if* $w.id < v.id$ *and* $w.reps \cap v.reps \neq \emptyset$*, that is, at least one representative was a member of both views. We denote this relation by* $w \rightharpoonup v$*. Furthermore, we use '*$\rightharpoonup^+$*' to denote the irreflexive transitive closure of '*$\rightharpoonup$*'. That is,* $v_0 \rightharpoonup^+ v_k$ *holds iff there exists a sequence of views (with increasing view ID's) such that* $v_0 \rightharpoonup v_1 \rightharpoonup v_2 \rightharpoonup \cdots \rightharpoonup v_k$*. Finally, we use '*$\rightharpoonup^*$*' to represent the reflexive transitive closure of '*$\rightharpoonup$*' (i.e.,* $w \rightharpoonup^* v$ *if* $w = v$ *or* $w \rightharpoonup^+ v$*).* ∎

[3] In general, the front-end should cache the configuration state.

Views are used in our scheme to control the serialization of events in a way that ensures that all the transactions that run in a particular view serialize as a block (similar to El Abbadi and Toueg's method in [2]). Every transaction must run in exactly one view; if a transaction invokes an operation at two representatives that do not belong to the same view, the transaction aborts. Transactions that run in different views serialize in the same order as view ID's[4].

The 'observes' relation between views formalized in definition 1 above will play a crucial rule in establishing the correctness of our protocol. Our arguments will rely on showing that enough information flows through the intersections between views. This flow of information will suffice for the overall computation to be equivalent to one in which views (and quorum assignments) do not change.

We assume that each event $e$ (execution of an operation) is assigned a view-specific serialization timestamp $t(e)$ by the concurrency control protocol in the view $v(e)$ in which it occurs. This timestamp should induce an ordering on the events that is consistent with the serialization order induced by the concurrency control protocol. Most concurrency control protocols can generate $t(e)$. For example, with standard two-phase locking, the timestamps $t(e)$ can be generated at commit time using Lamport clocks.[5] Static and hybrid atomic protocols also satisfy this assumption (see [25] for this classification of concurrency control protocols).

We augment the view-specific serialization timestamp $t(e)$ according to the following definition.

**Definition 2** *The* **serialization timestamp** $s(e)$ *of an event $e$ is given by $\langle v(e).id,\ t(e)\rangle$, where $v(e)$ is the view in which $e$ occurs, and $t(e)$ is the view-specific serialization timestamp assigned within the view by the concurrency control protocol. The relation '$<$' on serialization timestamps is that of lexical ordering, i.e., $s(e) < s(e')$ if $v(e).id < v(e').id$, or $v(e).id = v(e').id$ and $t(e) < t(e')$.* ∎

The next two definitions will be used to state the correctness condition that we consider fundamental to typed replicated data storage systems. Definition 3 encapsulates the semantics of the type, while definition 4 captures the behavior of the system.

**Definition 3** *The* **event dependency relation** *is a binary relation '$\rightarrow$' defined on the set of all possible events*[6]*. If $s(e) < s(e')$, we write $e \rightarrow e'$ whenever $e'$* needs to observe *the effects of $e$ in order for the execution of $e'$ to be consistent with the object's semantics.* ∎

To ensure that an event that needs to observe another actually does, the Quorum Consensus replication method requires that the initial and final quorums be chosen to satisfy the event dependency relation. However, we need a stronger condition to deal with the changing quorum assignments. The following definition captures the information flow between events that occur in potentially different views with potentially different quorum assignments.

**Definition 4** *If $s(e) < s(e')$, event $e'$ is said to* **observe** *event $e$ (denoted by $e \rightsquigarrow e'$) if and only if every initial quorum for $e'$ contains at least one copy of $e$.* ∎

[4] View ID's are chosen by the configuration installation protocol so that they induce a total order on views that is consistent with the 'observes' relation defined above. This causes transactions that serialize later to observe those that serialize earlier.

[5] To be precise, the timestamps should be read off the Lamport clock at the *lock point* of the transaction [4].

[6] This set is normally infinite.

Thus, a reconfiguration protocol can be considered correct if the quorum assignments it permits satisfy, for all events $e, e'$:

$$e \rightarrow e' \Rightarrow e \rightsquigarrow e'.$$

The statement of theorem 1 conincides with this correctness condition.

# 4 Reconfiguration during View Change

In this section we formally define the configuration state $G$ of an object, and the view change protocol that provides the context in which the configuration can change. $G$ consists of two versions of the quorum assignment for the object in question, and the views within which each applies. One of the two versions of the quorum assignment is termed the *current* version, and the other is called the *base* version. The current version is the one used by front-ends to decide which quorums to apply to a particular event, and the base version is used by the configuration installer to decide which reconfiguration rule(s) to apply. The configuration state is fully replicated in the sense that all representatives within a single view have (identical) copies of the configuration state.

Let $v$ be the current view for the object, and let $I(e), F(e)$ denote the classes (sets) of valid initial and final quorums, respectively that apply to event $e$ within that view. Similarly, let $\mathcal{I}(e), \mathcal{F}(e)$ denote the *base* quorum classes for event $e$ that were installed in view $\mathcal{V}(e)$. Then, the configuration $G$ can be stated as

$$\begin{aligned} G = \langle \quad v \quad &= \langle id, reps \rangle, \\ I \quad &: E \rightarrow 2^{2^S}, \\ F \quad &: E \rightarrow 2^{2^S}, \\ \mathcal{V} \quad &: E \rightarrow V, \\ \mathcal{I} \quad &: E \rightarrow 2^{2^S}, \\ \mathcal{F} \quad &: E \rightarrow 2^{2^S} \ \rangle, \end{aligned}$$

where $E$ is the set of the all possible events, $S$ is the set of all sites in the system, $V$ is the set of all views that will ever be installed in the system, and $2^A$ is the powerset of $A$ (hence, $2^{2^S}$ is the set of all possible quorums). We use functional notation to specify the quorum assignment for notational convenience only; implementations of our method need not support the same level of generality.

Implicit in the above definition of the configuration are the notions of an event quorum incorporating both an initial and a final quorum, and the set of all representatives for an object..

**Notation 1** *The* **current quorum class**, $Q(e)$, *for event $e$ is the set of all quorums such that each is sufficient to complete $e$ correctly, i.e., the union of any pair of current initial and final quorums for $e$ is an element in $Q(e)$. Formally,*

$$Q(e) \stackrel{\text{def}}{=} \{(p \cup q) \mid \ p \in I(e),\ q \in F(e)\}.$$

*The base quorum set $\mathcal{Q}(e)$ is similarly derived from $\mathcal{I}(e)$ and $\mathcal{F}(e)$. The set $R$ of representatives is implicit in the quorum sets, and given by:* $R = \bigcup_e \bigcup_{q \in Q(e)} q$. ∎

The fundamental procedure in our protocol is that of an atomic view change, run as a transaction. During a view change, the installer of the new view also computes the new quorum assignment

based on the information it collects as part of the view change protocol. This strategy has several advantages. First, the messages used to choose the members of the view can be used to collect each representative's (among the new view's members) partial history and object configuration. These are merged, and then distributed to all the representatives in the new view so as to bring all the representatives, including those that are being decommissioned, up-to-date with respect to each other. This scheme possesses a second advantage: the reconfiguration procedure can use the information collected from the view members to evaluate the extent of reconfiguration that is legal for each operation.

We start by describing the configuration installation procedure for a new view, and then we characterize the constraints on reconfiguration. An implementation of our protocol would have the freedom of selecting any configuration that satisfies these constraints, perhaps optimizing such properties as availability. For clarity of presentation, the description below assumes that there is only one object in the system. In section 8 we discuss how to generalize our method to handle multiple objects.

### 4.1 Configuration Installation

The following procedure can be considered a generalization of the view management protocol in [1], to handle abstract data types and permit dynamic reconfiguration of the base quorum sets. The configuration installer will determine and install both the new quorum assignments and the new view together. Thus, it is always the case that any two representatives whose current views agree must also agree on the quorum assignments. Let $G.v$ denote the new view being installed (as part of the new configuration $G$). The installer and the other representatives run the following protocol:

1. **Begin-Transaction**.
2. The installer broadcasts "creating new view" message. Each representative $i$ that receives this message does the following (whether it is a current representative—$i \in G_i.R$—or not):
   (a) *Write-lock* its local history $h_i$ and its configuration $G_i$;
   (b) *Send* both $h_i$ and $G_i$ to the view installer.
3. Set $G.v.reps$ to be the set of representatives from whom the installer successfully receives a reply (containing $h_i$ and $G_i$) to its broadcast in the previous step.
4. Choose a unique (for the object) $G.v.id$ that is larger than all $G_i.v.id$.
5. Merge the partial histories received. Formally, $H_G := \bigcup_{i \in G.v.reps} h_i$.
6. Choose the new quorum assignments ($I, F, \mathcal{I}, \mathcal{F}$, and the corresponding base view $\mathcal{V}$, according to the rules provided in section 5.1 below).
7. Send the new configuration $G$ and the merged history $H_G$ to all the members of the new view (*i.e.*, $G.v.reps$). On receiving this message, every representative $i \in G.v.reps$ sets:
   $h_i := H_G$
   $G_i := G$.
8. **End-Transaction**.

Transactions that become active after the installation begins have to abort, since they read $h_i, G_i$ that are write-locked in step 2.a. When the configuration installation transaction is committed, all the locks on the local histories and configurations are released and processing can continue.

One of the criticisms that can be levelled at view management protocols in general is the absence of *locality* in the view change activity. However, our configuration installation protocol appears to be object-specific; so, is it immune from this problem? Not entirely, because a transaction that accesses two objects requires that their configurations have identical view ID's. There are three aspects to this problem: (1) locality of configuration information, (2) communication overhead, and (3) loss of concurrency due to reconfiguration triggered even though the quorum assignment need not change. In section 8 we discuss a solution to this problem. We eliminate the first aspect of the problem, releaving representatives of one object from having to store information about other objects. We can mitigate the second aspect of the problem, by reducing the message sizes needed to a very small size when a reconfiguration is triggered only to adopt a new view ID. Unfortunately, we lose some concurrency because we must lock the history and configuration to ensure the atomicity of the reconfiguration transaction. All three points are addressed in section 8 in the context of handling reconfiguration for multiple objects with different configurations.

# 5 Views and Quorums

The configuration installer needs to assess how much knowledge is available among the sites $G.v.reps$ of the new configuration $G$, about a given event $e$. We define two conditions of knowledge: the weaker condition is satisfied if $v.reps$ contains a quorum for the event. The stronger one obtains when $v.reps$ contains, very roughly, a majority of the object representatives. The rest of this section is devoted to defining these two conditions precisely.

Our protocol permits different representatives in the system to disagree about the configuration state of an object. This means that the representatives joining a new view (and configuration) may have different views. Under such circumstances, it helps to identify the representative that has the best information.

**Definition 5** *Let $G$ be a new configuration (with $G.v$ being the new view) that is being installed in place of $\{G_i \mid i \in G.v.reps\}$. Define $g(e)$, the representative that has the latest base quorums for event $e$ among those that are joining the new view, as follows:*

$$g : E \rightarrow G.v.reps$$

*such that*

$$G_g.\mathcal{V}(e).id = \max_{i \in G.v.reps} G_i.\mathcal{V}(e).id$$

*Whenever $e$ is understood from the context, we simply use $g$ to denote $g(e)$.* ∎

Thus, $g(e)$ knows the most recent base view[7] $\mathcal{V}(e)$ for event $e$ among the sites that have joined the new view $G.v$. Note that the installer of the new configuration $G$ (and its corresponding view $G.v$) discovers $g(e)$ during the installation of $G$ (more precisely, in step 3 of the configuration installation protocol given in section 4.1.

[7] A view $v$ is more recent than a view $w$ if $v.id > w.id$.

Now that we have gotten these notations out of the way, let us formally define the notions of a view containing a quorum or a majority of representatives for a given event. When the first condition is satisfied, we say that the view *semi-covers* the event. We say that the view *covers* the event in the case of the second (and stronger) condition. Our intention, as is evident in the next section, is to permit only views that semi-cover an event to change its current quorum assignment, and only views that cover an event to change its base quorum assignment.

**Definition 6** *A view $G.v$* **semi-covers** *an event $e$ if the following condition holds during the view's installation:*

$$G.v.reps \supseteq \text{ some } q \in G_g.\mathcal{Q}(e)$$

*where $\mathcal{Q}$ and $g$ are given in notation 1 and definition 5 earlier in this section.* ∎

**Definition 7** *A view $G.v$* **covers** *an event $e$ if $G.v$ semi-covers $e$ and:*

$$G.v.reps \supseteq \text{ some } \bar{q} \text{ such that } \forall q \in G_g.\mathcal{Q}(e) : \bar{q} \cap q \neq \emptyset.$$

*We say that $\bar{q}$ is a* **co-quorum** *for $e$ according to the base quorum assignment in $G_g$.* ∎

The above definition of the cover notion forces all the views that cover a particular event to occur *serially*, in the sense that only one view at a time can cover an operation, and at least one site in every covering view must be a member of the previous cover (this is formalized and proved in lemma 3).

Both of the above two definitions are designed to permit the view installer to check them before the new view is committed. Based on which, if any, of these two properties holds for the view being installed, the installer decides which of the reconfiguration rules given in section 5.1 applies.

## 5.1 Quorum Assignment Constraints

Our method of reconfiguration does not specify the new quorum assignments, rather, it constrains it. This way the reconfiguration program can trade-off such things as performance versus availability more freely than if we make that decision *a priori*.

**Quorum intersection rules:** The fundamental constraint derives from the necessity that events that depend on each other observe each other. Formally, we write

$$(\mathbf{T_0}) \quad \forall e, e' : e \rightarrow e' \Rightarrow F(e),\ I(e') \text{ element-pair-wise-intersect},$$

and

$$(\mathcal{T}_\mathbf{0}) \quad \forall e, e' : e \rightarrow e' \Rightarrow \mathcal{F}(e),\ \mathcal{I}(e') \text{ element-pair-wise-intersect},$$

where,

**Notation 2** *We say that two quorum classes $F$ and $I$* **element-pair-wise-intersect** *if and only if: $\forall q \in F,\ \forall q' \in I : q \cap q' \neq \emptyset$.* ∎

The configuration installer decides which events are semi-covered by the new view, which are also covered and which are neither, according to definitions 6 and 7 in section 3 above. Based on this decision, the following rules circumscribe the permissible reconfigurations for each event. Note that the decision as to which rules apply relies on the base quorum assignment stored at site $g(e)$, the one that sends to view $G.v$'s installer the latest base configuration (largest $G_i.\mathcal{V}(e).id$) for event $e$.

**Semi-cover rules:** Constrain the changes in the current quorum assignment for event $e$. We first cover the possibility that $e$ is not supposed to run in the new view $G.v$ at all,

$$(\mathbf{T_1}) \quad e \text{ not semi-covered by } G.v \Rightarrow G.I(e) = G.F(e) = \emptyset.$$

Otherwise, we bound the size to which final quorums can shrink,

$$(\mathbf{T_2}) \quad \forall q \in G.F(e),\ \exists q' \in G_g.\mathcal{F}(e):\ q \supseteq q'$$

This means that a current final quorum cannot shrink smaller than all the latest base final quorums. We also restrict initial quorums of update and read events to be non-empty,

$$(\mathbf{T_3}) \quad \exists e' \rightarrow e \Rightarrow \forall q \in G.I(e),\ q \neq \emptyset.$$

This is regardless of whether $e'$ is semi-covered by $G.v$.

**Cover rules:** Limit the choice of new base quorums of event $e$. Sometimes the new view is not big enough to permit any changes:

$$(\mathcal{T}_1) \quad e \text{ not covered by } G.v \Rightarrow G.\mathcal{I}(e), G.\mathcal{F}(e), G.\mathcal{V}(e) = G_g.\mathcal{I}(e), G_g.\mathcal{F}(e), G_g.\mathcal{V}(e).$$

In this case, the base quorum assignment and view are copied from the latest available ones. Otherwise, the base view should become identical to the current one to reflect that the base quorum assignment was changed

$$(\mathcal{T}_2) \quad G.\mathcal{V}(e) = G.v.$$

The only other constraint is $\mathcal{T}_0$ above.

# 6 Protocol Invariants and Correctness

In this section, we establish the properties of the set $\Gamma$ of all configurations that take place during the lifetime of the object, and the corresponding set of all views $\Gamma.v$.

**Lemma 1** *$\langle \Gamma.v, \rightharpoonup^* \rangle$ is a partial order.*

**Proof:** Follows from the definition of $\rightharpoonup$ and its reflexive transitivie closure, $\rightharpoonup^*$. ∎

**Lemma 2** *Let $G$ be a new configuration that replaces $G_i$ at some site $i \in G.v.reps$. Then, for every event $e$:*

1. $G_i.v \rightharpoonup G.v$,
2. $G.\mathcal{V}(e) \rightharpoonup^* G.v$, *and*
3. $G_i.\mathcal{V}(e) \rightharpoonup^+ G.v$.

**Proof:**

1. From the protocol, $i \in G_i.v.reps$, $i \in G.v.reps$, and $G_i.v.id < G.v.id$. Part 1 follows immediately.
2. By induction on the set of all configurations $\Gamma$.

   *Basis.* The view corresponding to the initial configuration $A$ must have covered all events $e$. Hence, $\forall e : A.\mathcal{V}(e) = A.v$.

   *Step.* Configuration $G$ is installed, replacing $G_i$ at all sites $i \in G.v.reps$. If $G.v$ covers $e$, then $G.\mathcal{V}(e) = G.v$, and we are done. Otherwise, the protocol forces $G.\mathcal{V}(e) = G_g.\mathcal{V}(e)$. By the induction hypothesis, $G_g.\mathcal{V}(e) \rightharpoonup^* G_{g(e)}.v$. But, by part (1) of the lemma, $G_{g(e)}.v \rightharpoonup G.v$. Hence $G.\mathcal{V}(e) \rightharpoonup^* G.v$.
3. From (1) and (2) above, it follows that $G_i.\mathcal{V}(e) \rightharpoonup^+ G.v$. But, step 4 in the configuration installation protocol requires that $G.v.id$ be unique. Hence, $G_i.\mathcal{V}(e) \neq G.v$. Statement (3) of the lemma follows. ∎

**Lemma 3** *Let* $\Gamma(e) = \{G \mid G.v \text{ covers } e\}$. *Then* $\langle \Gamma(e).v, \rightharpoonup^* \rangle$ *is a total order. That is, there exists a total order* $G^{(0)}, G^{(1)}, G^{(2)}, \ldots$ *on* $\Gamma(e)$, *such that:*

$$G^{(0)}.v \rightharpoonup^+ G^{(1)}.v \rightharpoonup^+ G^{(2)}.v \rightharpoonup^+ \ldots$$

**Proof:** By induction on $|\Gamma(e)|$.

*Basis.* $|\Gamma(e)| = 1$. Trivial.

*Step.* $|\Gamma(e)| = n + 1$, and a new (the $n+2$nd) configuration $G$, whose corresponding view $G.v$ covers $e$, gets installed. By the induction hypothesis,

$$G^{(0)}.v \rightharpoonup^+ G^{(1)}.v \rightharpoonup^+ G^{(2)}.v \rightharpoonup^+ G^{(n)}.v$$

We will show that $G^{(n)}.v \rightharpoonup^+ G.v$. Let $C.v = G_g.\mathcal{V}(e)$, that is, configuration $C$ was the one in which the base quorum assignment in $G_g$ for event $e$ was installed (see fig. 1.a). Since $G.v$ covers $e$, $\exists q_c, \bar{q}_c \subseteq G.v.reps$ such that

$$q_c \in C.\mathcal{Q}(e), \text{ and } \forall q \in C.\mathcal{Q}(e) : \bar{q}_c \cap q \neq \emptyset. \tag{1}$$

Now, by the third statement in lemma 2, $C.v = G_g.\mathcal{V}(e) \rightharpoonup^+ G.v$. So if we can show that $C = G^{(n)}$, then we are done. By definition, $C \in \Gamma(e)$. Assume, by way of contradiction, that $C.v = G^{(k)}.v$, $0 \leq k < n$ (see fig. 1.b). Therefore, $\exists G^{(k+1)} \in \Gamma(e)$ such that $G^{(k+1)}.v$ covers $e$, and

$$G^{(k)}.v \rightharpoonup^+ G^{(k+1)}.v \tag{2}$$

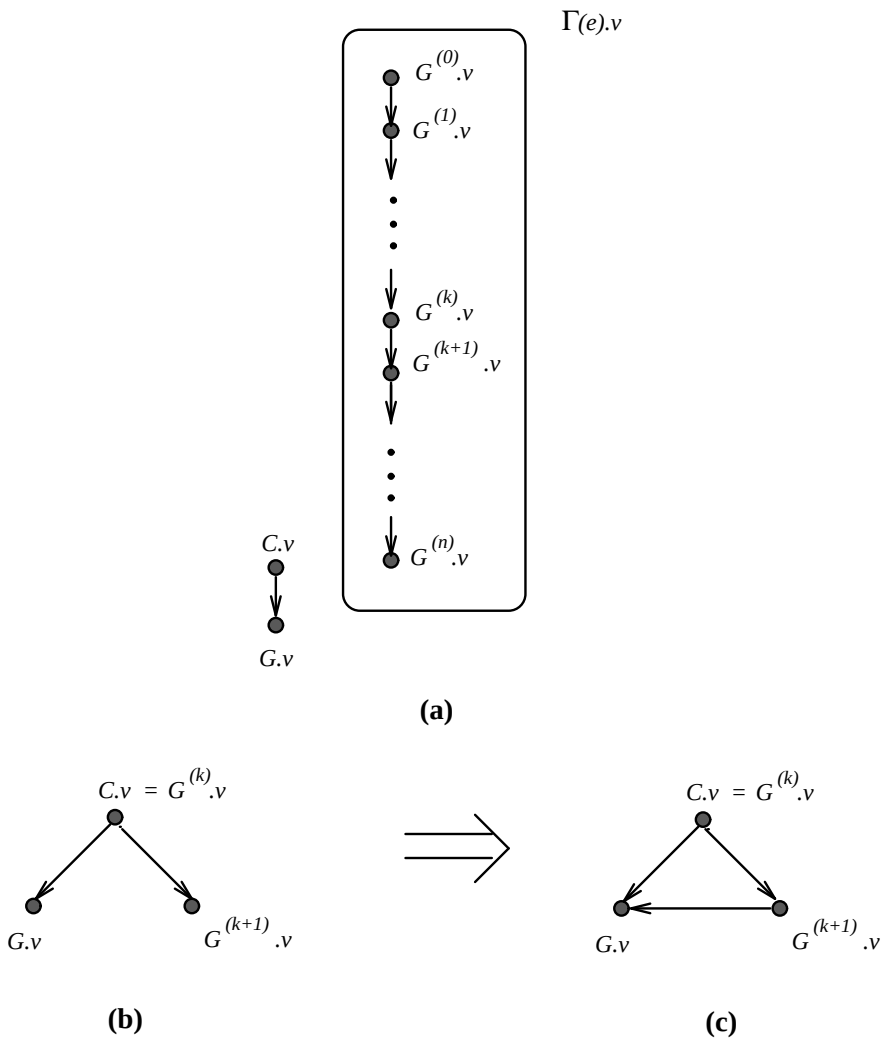


Figure 1: A new configuration $G$ is being installed.

In fig. (a) new view $G.v$ covers event $e$, and configuration $C$ contains the latest base quorum assignment for $e$ that is accessible to the installer of $G$. We Would like to show that $G^{(n)}.v \rightharpoonup^{+} G.v$. Figs. (b) and (c) illustrate the contradiction that establishes the induction step.

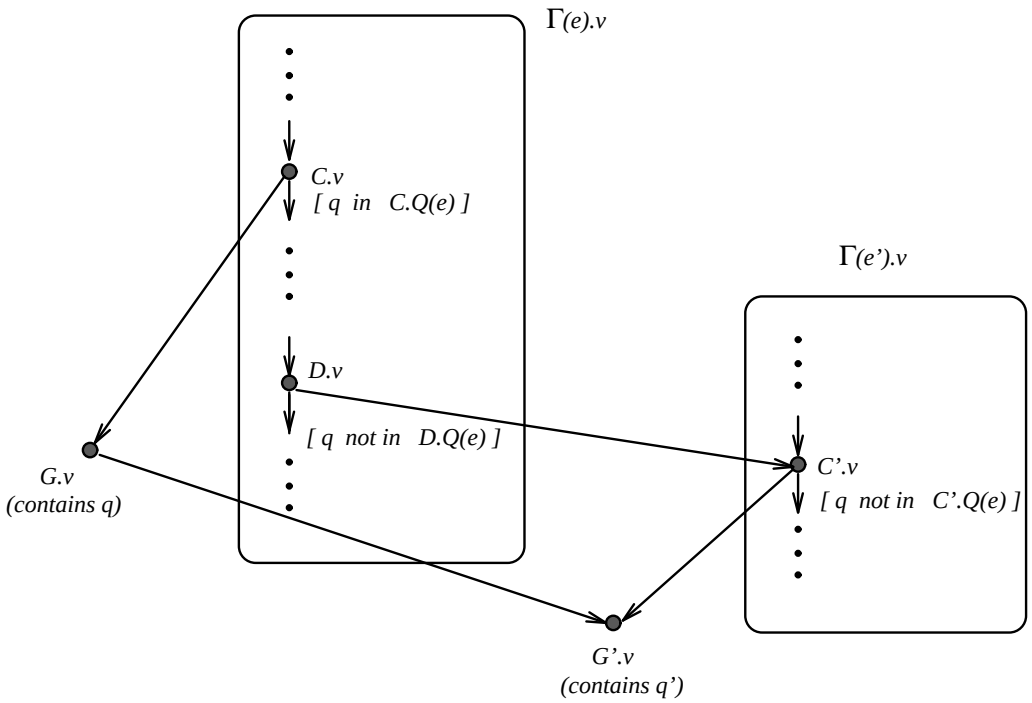


Figure 2: If $e'$ depends on $e$, then $G'.v$ must observe $G.v$.

Events $e, e'$ occur under configurations $G, G'$, respectively. Configuration $C$ contains the base quorum assignment used for $e$ in G. (((More...)))

Consequently, $\exists r_c, \bar{r}_c \subseteq G^{(k+1)}.v.reps$ such that

$$r_c \in C.\mathcal{Q}(e), \text{ and } \forall r \in C.\mathcal{Q}(e) : \bar{r}_c \cap r \neq \emptyset. \tag{3}$$

From equations 1 and 3, $\bar{r}_c \cap q_c \neq \emptyset$, implying that $G^{(k+1)}.v.reps \cap G.v.reps \neq \emptyset$. But, $G^{(k+1)}$ existed before $G$. Therefore, $\exists i \in G.v.reps$ s.t. $G_i = G^{(k+1)}$. By definition of $g(e)$, it follows that (see fig. 1.c):

$$G_g.\mathcal{V}(e).id = C.v.id = G^{(k)}.v.id > G^{(k+1)}.v.id,$$

which contradicts equation 2. ∎

**Lemma 4** *Let events $e$ and $e'$ occur under configurations $G$ and $G'$, respectively, and $e \rightarrow e'$. If $s(e) < s(e')$, then $G.v \rightharpoonup^* G'.v$.*

**Proof:** We have two cases. If $G = G'$, then $G.v = G'.v$ and we are done. Otherwise, $G \neq G'$. Assume, by way of contradiction, that $G.v \not\rightharpoonup^* G'.v$. But, $s(e) < s(e') \Rightarrow G.v.id < G'.v.id$. Consequently,

$$G.v.reps \cap G'.v.reps = \emptyset \tag{4}$$

For $e$ and $e'$ to have occurred under $G$ and $G'$, it must be true that they are semi-covered by $G.v$ and $G'.v$ respectively. Hence, (see fig. 2)

$$\exists q \in G_g.\mathcal{Q}(e) \quad \text{s.t.} \quad q \subseteq G.v.reps \tag{5}$$

$$\exists q' \in G'_{g'}.\mathcal{Q}(e') \quad \text{s.t.} \quad q' \subseteq G'.v.reps \tag{6}$$

From equation 4, it follows that

$$q \cap q' = \emptyset. \tag{7}$$

Let $C.v = G_g.\mathcal{V}(e)$, that is, configuration $C$ was the one in which the base quorum assignment in $G_g$ for event $e$ was first installed. Let $C'$ be defined similarly for $G'_{g'}$ and $e'$. Since $e \rightarrow e'$ the base quorum assignments chosen as part of $C$ and $C'$ must have observed rule $(\mathcal{T}_0)$, causing $C.\mathcal{F}(e), C.\mathcal{I}(e')$ to element-pair-wise-intersect (similarly for $C'.\mathcal{F}(e), C'.\mathcal{I}(e')$). This, together with equations 5, 6, and 7, implies that

$$q \notin C'.\mathcal{Q}(e), \text{ and} \tag{8}$$

$$q' \notin C.\mathcal{Q}(e'). \tag{9}$$

Without loss of generality, assume that $C.v.id < C'.v.id$. We claim that $C.v \rightharpoonup^* C'.v$, and we prove it in the rest of this paragraph. If not, then there must exist some least common ancestor $E.v \in \Gamma(e).v$ of both $C.v$ and $C'.v$ (*i.e.*, $E.v$ has the largest view ID among such common ancestors). Consequently, $C'$ and $E$ must have the same base quorum assignment for event $e$. Consider the first configuration $H.v \in \Gamma(e).v$ that follows $E.v$ in the total order of lemma 3. In order for $H.v$ to cover $e$, there must exist some base quorum $r \in E.\mathcal{Q}(e)$ that is a subset of $H.v.reps$. By rule $(\mathcal{T}_0)$ every $e'$ base quorum in $C'.\mathcal{Q}(e')$ must intersect $r$. We can safely assume that at least one such base quorum $r'$ for $e'$ is a subset of $C'.v.reps$[8]. Therefore, $H.v.reps$ intersects $C'.v.reps$. Knowing that $H.v.id \leq C.v.id < C'.v.id$, we must conclude that $H.v \rightharpoonup^* C'.v$, which contradicts our assumption that $E$ is the least common ancestor in $\Gamma(e).v$ of $C.v$ and $C'.v$.

By equation 9 and reconfiguration rule $(\mathcal{T}_1)$, there must exist a configuration $D$ such that $D.v$ covers $e$ and

$$C.v \rightharpoonup^* D.v \rightharpoonup^* C'.v, \text{ and}$$

$$q \notin D.\mathcal{Q}(e)$$

We take $D$ to have the smallest $D.v.id$ consistent with the above two properties. Hence, $q \in D_d.\mathcal{Q}(e)$.

Since $D.v$ covers $e$, then $D.v.reps$ must contain a co-quorum $\bar{q} \in C.\mathcal{Q}(e)$ such that $\bar{q} \cap q \neq \emptyset$, causing $G.v.reps \cap D.v.reps \ni$ some representative $k$.

Consider the relation $\rightharpoonup$ between $G.v$ and $D.v$. If $D.v \rightharpoonup G.v$, then $D.v.id$ must be $> C.v.id$. But,

$$\begin{aligned} G_k.\mathcal{V}(e).id &= D.v.id \text{ , for some } k \in G.v.reps \\ &> C.v.id \text{ , because } D.v \rightharpoonup^* C.v \\ &= G_g.\mathcal{V}(e).id \text{ , by definition of } C \end{aligned}$$

which contradicts the definition of $g(e)$ as the representative that sends the configuration with the largest $\mathcal{V}(e).id$ to the installer of $G$.

Otherwise, we must have $G.v \rightharpoonup D.v$, because $D.v.reps \supseteq \bar{q}$ and $G.v.reps \supseteq q$, and we know that $q \cap \bar{q} \neq \emptyset$. It follows immediately that $G.v \rightharpoonup^* G'.v$ (see fig. 2) which contradicts our original assumption. ∎

[8] For this not to happen, the installer of $C'$ must delete the all the existing base quorums that are subsets of the new set of representatives, without adding new base quorums that are. We bar this behavior, although the reconfiguration rules do not.

**Theorem 1** *For all events* $e, e' :\ e \rightarrow e' \Rightarrow e \rightsquigarrow e'$.

**Proof:** Let $G$ and $G'$ be the configurations under which $e$ and $e'$ occurred, respectively. By lemma 4, we have only two cases:

1. $G.v = G'.v$. In this case, the lemma follows trivially from rule $(T_0)$, which ensures that $G.F(e)$ and $G.I(e')$ element-pair-wise-intersect, satisfying definition 4 of the '$\rightsquigarrow$' relation.

2. $G.v \rightarrow^{+} G'.v$. From the definition of '$\rightarrow^{+}$', there exists a sequence of views $G.v \rightarrow \cdots \rightarrow G'.v$. Consider the longest such sequence. Let $D.v$ be the first view after $G.v$ in this sequence, that semi-covers $e'$. $D.v$ exists since $G'.v$ semi-covers $e'$. From definition 6 of the semi-cover, and because there is no view that covers $e'$ between $G.v$ and $D.v$, we have:

$$D.v.reps \supseteq \text{ some } q_d \in D_d.\mathcal{I}(e') = G.\mathcal{I}(e').$$

   But, the cover rule $(\mathcal{T}_0)$ requires that $G.\mathcal{I}(e')$ and $G.\mathcal{F}(e)$ element-pair-wise-intersect. This, together with semi-cover rule $(T_2)$, causes $G.\mathcal{I}(e')$ and $G.F(e)$ to element-pair-wise-intersect too. Therefore,

$$\forall q \in G.F(e) :\ D.v.reps \cap q \neq \emptyset. \tag{10}$$

   That is, regardless of the final quorum used for $e$ under $G$, the installer of $D$ will obtain at least one copy of $e$, and distribute it to all the representatives under $D$, according to the configuration installation protocol. Thereafter, at least one copy of $e$ will be copied to the representatives of each subsequent view in the sequence up to and including the members of $G'.v.reps$. The semi-cover rule $(T_3)$ ensures that $q' \neq \emptyset$, $\forall q' \in G'.I(e')$. Intuitively, this means that any initial quorum for $e'$ will contain at least one copy of $e$, and we are done. Formally, we have satisfied definition 4 for $e \rightsquigarrow e'$. ∎

# 7 Related Work

In some sense, the reconfiguration method we present in this paper forms a generalization of El Abbadi *et al.*'s [1] view management protocol. We generalize view management in several ways: we are able to handle arbitrary abstract data type operations, while El Abbadi *et al.*'s deals only with read and write operations. We permit the specification of quorums as sets, rather than weights, which gives the system administrator more freedom in quorum assignments. Most importantly, we achieve higher availability because we can change all the aspects of the configuration, while [1] can reconfigure only a part of it. In particular, we permit changing the set of representatives used to replicate the data object. At the same time, our protocol was influenced by Herlihy's reconfiguration method [13], and behaves in a dual manner by permitting initial quorums to shrink instead of final quorums. Other techniques to reconfigure replicated data can be found in [7, 16, 21].

El Abbadi, Skeen and Cristian [1] give a view management protocol that permits the approximate tracking of communication partitions. Whenever a failure or recovery is detected, a new view that better approximates the current failure situation can be installed. Their view installation protocol runs as a transaction that requires consensus between all the sites in the view before that view can be installed. In a later paper [3], El Abbadi and Toueg relaxed that requirement and gave a cleaner set of rules to govern the choice of read and write quorums within each view. The base quorum sets

in our scheme generalize their *accessibility thresholds* for use with abstract data types, but differ in that base quorum sets are dynamically changeable, thus permitting higher availability.

In [13], Herlihy proposed a reconfiguration method in which a sequence of configurations is computed when a majority of sites are available, to be used later when failures occur. A transaction that fails to collect a quorum using a particular level, aborts and restarts at a higher level. Transactions serialize by the level number of the configurations they use. The availability of blind-write operations (and operations with relatively small initial quorums) improve at higher levels because final quorums can shrink. On the other hand, the initial quorums cannot shrink, since they are required to intersect the final quorums at all the lower levels. The main disadvantage of Herlihy's scheme is that the sequence of configurations must be computed before the failures occur, and that can be hard to calculate.

Our scheme can be considered a dual of Herlihy's because we favour operations that have empty or small final quorums. We permit initial quorums, rather than final quorums, to shrink. We feel that it makes more sense, when failures happen, to favour operations that read the state of data objects, because they allow us to take emergency measures (*e.g.*, become a non-replicated system) to continue to support the object in the face of further failures. At the same time, we provide a sound framework to permit blind-write transactions maximum availability under extreme failure conditions, by sacrificing external consistency [10].

It is worth mentioning that Goldman, in [7], gives a complete formal development, based on I/O automata, of a reconfigurable version of Gifford's weighted voting replication algorithm [6]. More recently, Oki [21] gives a highly optimized primary copy method with reconfiguration, and Hwang [16] presents a reconfiguration protocol for Liskov and Ladin's [20] high availability, non-atomic, replication method.

# 8 Discussion

In this paper, we presented a reconfiguration method for the General Quorum Consensus replication algorithm. Our method generalized, and improved on, previous work in crucial respects. In addition to the comparative advantages of our method that we described in the previous section, we obtain the following extra benefits essentially for free.

As we mentioned in the introduction to this paper, it is possible to treat *site recovery from failure as a special case of reconfiguration*. This can be done when enough sites remain available after a failure to be able to read the complete state of the data object. In this case, if we reconfigure the object to include the recovering site as a *new* representative, then that site will be initialized with information that is available entirely among the other representatives. Traditional recovery techniques recover the state of a repaired site by logging the site's activities on a stable storage device that does not lose its memory when a failure happens. In the systems where the level of desired recoverability and the failure modes are such that only mild failures occur, our approach to recovery obviates the need for stable storage devices.

We have presented our protocol as if there were only one object in the system. Our method can support multiple objects with multiple configurations (those with the same configuration can be treated as a single object), without destroying the locality of our configuration installation protocol. Whenever a reconfiguration is triggered, the objects that need new quorum assignments are reconfigured, either using separate reconfiguration transactions if they need not have the same

view ID, or using one transaction otherwise. We first observe that the only piece of configuration information that may span objects that have otherwise different configurations is the view ID ($G.v.id$). Thus, the configuration information is purely local.

Reconfiguration locality can be destroyed only if an object is forced to reconfigure even though it does not need a new quorum assignment. The only case where this can happen is when a transaction accesses two objects whose configuration have different view ID's. In this case, we perform the reconfiguration of the two objects as if they were two independent reconfigurations, except that the same view ID is selected, and they run as two parts of the same transaction. We can do better if we observe that if all the representatives in a view join a new view, and the new configuration does not have a new quorum assignment, then the representatives' local histories need not be merged. Most non-locally triggered reconfigurations can be expected to fit this requirement. Under such circumstances, two objects can be made to have the same view ID by running a degenerate form of the configuration installation transaction that updates only the view ID's. Thus, we can reduce the messages sizes in such reconfigurations, mitigating their overhead. Unfortunately, this does not remove the need to lock the local histories and configurations for the two objects on behalf of the transaction that is intalling the new view ID.

One can conceivably take an entirely different approach to reconfiguration. We could use a specialized non-atomic highly available server (such as the one in [20]) to store the configuration, and reconfigure by accessing this server. We found our approach to be more fruitful at the present stage because the atomicity of the reconfiguration transactions allowed us to state and prove much stronger properties of the protocol than we could have with a configuration server that does not support atomic reconfigurations.

## 9 Acknowledgements

The author would like to thank Mei Hsu and Bill Weihl for invaluable advice during the development of this work, and Jie Wang for careful checking of the proofs. Any errors that remain are, of course, solely the author's responsibility.

## References

[1] A. El Abbadi, D. Skeen, and F. Cristian. An efficient, fault-tolerant protocol for replicated data management. In *Proc. 4th ACM Symp. on the Principles of Database Systems*, Portland, Oregon, 1985.

[2] A. El Abbadi and S. Toueg. The group paradigm for concurrency control protocols. *IEEE Trans. on Knowledge and Data Engineering*, 1(3), Sep. 1989. Also available as technical report TRCS89-19, Dept. of Computer Science, Univ. of California, Santa Barbara.

[3] A. El Abbadi and S. Toueg. Maintaining availability in partitioned replicated databases. *ACM Trans. on Database Systems*, 14(2):264–290, June 1989. Also available in Proc. 5th ACM Symp. on the Principles of Database Systems, Cambridge, Mass., Mar. 1986.

[4] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, Massachusetts, 1987.

[5] A. Chan and R. Gray. Implementing distributed read-only transactions. *IEEE Trans. on Software Engineering*, SE-11(2):205–212, Feb. 1985.

[6] D.K. Gifford. Weighted voting for replicated data. In *Proc. 7th ACM Symp. on Operating System Principles*, Dec. 1979.

[7] K.J. Goldman. Data replication in nested transaction systems. Master's thesis, M.I.T., Lab. for Computer Science, May 1987.

[8] J.M. Gray. Notes on database operating systems. In R. Bayer, R.M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*. Lecture Notes in Computer Science, vol. 60, Springer-Verlag, Heidelberg, 1978.

[9] A.A. Heddaya, M. Hsu, and W.E. Weihl. Two phase gossip: Managing distributed event histories. *Information Sciences*, 49(1,2,3):35–57, Oct./Nov./Dec. 1989. Special issue on databases.

[10] Abdelsalam Heddaya. View-based reconfiguration of replicated data. Technical Report BU-CS-90-001, Boston University, Computer Science Dept., 1990.

[11] M.P. Herlihy. *Replication Methods for Abstract Data Types*. PhD thesis, M.I.T., Lab. for Computer Science, May 1984.

[12] M.P. Herlihy. A quorum-consensus replication method. *ACM Trans. on Computer Systems*, 4(1):32–53, Feb. 1986.

[13] M.P. Herlihy. Dynamic quorum adjustment for partitioned data. *ACM Trans. on Database Systems*, 12(2):170–194, June 1987.

[14] M. Hsu and A. Chan. Partitioned two-phase locking. *ACM Trans. on Database Systems*, 11(4):431–446, Dec. 1986.

[15] M. Hsu and S.E. Madnick. Shifting timestamps for concurrency control in an information heirarchy. *Information Processing Letters*, 27(6):291–297, May 1988.

[16] D.J-H. Hwang. Constructing a highly-available location service for a distributed environment. Master's thesis, M.I.T., Lab. for Computer Science, Nov. 1987.

[17] S. Jajodia and Mutchler. Dynamic voting. In *Proc. ACM SIGMOD International Conf. on Management of Data*, May 1987.

[18] B. Liskov. Distributed programming in Argus. *Comm. ACM*, 31(3):300–313, Mar. 1988.

[19] B. Liskov and *et al.* Argus reference manual. Technical Report MIT/LCS/TR-400, M.I.T., Lab. for Computer Science, 1987.

[20] B. Liskov and R. Ladin. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proc. 5th ACM Symp. on Principles of Distributed Computing, Calgary, Canada*, Calgary, Alberta, Canada, Aug. 1986.

[21] B.M. Oki. *Viewstamped replication for highly available distributed systems*. PhD thesis, M.I.T., Lab. for Computer Science, Aug. 1988.

[22] S. Sarin, C. Kaufman, and J. Somers. Using history information to process delayed database updates. In *Proc. 12th Internat. Conf. on Very Large Databases*, Aug. 1986.

[23] A.Z. Spector, J. Butcher, D.S. Daniels, D. Duchamp, J.L. Eppinger, C.E. Fineman, A. Heddaya, and P.M. Schwarz. Support for distributed transactions in the TABS prototype. *IEEE Trans. on Software Engineering*, SE-11(6):520–530, June 1985. Also available in Proc. 4th IEEE Symp. on Reliability in Distributed Software and Database Systems, Silver Spring, Maryland, Oct. 1984.

[24] A.Z. Spector and K.R. Swedlow (eds.). Guide to the Camelot distributed transaction facility. Technical report, Computer Science Dept., Carnegie Mellon University, May 1988. New edition to be published as a book.

[25] W.E. Weihl. *Specification and Implementation of Atomic Data Types.* PhD thesis, M.I.T., Lab. for Computer Science, 1984.

[26] W.E. Weihl. Distributed version management for read-only actions. *IEEE Trans. on Software Engineering*, SE-13(1), Jan. 1987.