# Dynamic Voting for Consistent Primary Components*

**Esti Yeger Lotem**
IBM Haifa Research Laboratory
Israel
esti@vnet.ibm.com

**Idit Keidar**
Computer Science Institute
The Hebrew University of Jerusalem, Israel
idish@cs.huji.ac.il
http://www.cs.huji.ac.il/~idish

**Danny Dolev**
Computer Science Institute
The Hebrew University of Jerusalem, Israel
dolev@cs.huji.ac.il
http://www.cs.huji.ac.il/~dolev

**Abstract**

Distributed applications often use quorums in order to guarantee consistency. With emerging world-wide communication technology, many new applications (e.g., conferencing applications and interactive games) wish to allow users to freely join and leave, without restarting the entire system. The dynamic voting paradigm allows such systems to define quorums adaptively, accounting for the changes in the set of participants. Furthermore, dynamic voting was proven to be the most available paradigm for maintaining quorums in unreliable networks. However, the subtleties of implementing dynamic voting were not well understood; in fact, many of the suggested protocols may lead to inconsistencies in case of failures. Other protocols severely limit the availability in case failures occur during the protocol.

In this paper we present a robust and efficient dynamic voting protocol for unreliable asynchronous networks. The protocol consistently maintains the primary component in a distributed system. Our protocol allows the system to make progress in cases of repetitive failures in which previously suggested protocols block. The protocol is simple to implement, and its communication requirements are small.

## 1 Introduction

Numerous fault tolerant distributed systems, e.g., ISIS [5], use the primary component[1] paradigm to allow a subset of the processes to function when failures occur. A majority (or quorum) of the processes is usually chosen to be the primary component. In unreliable networks this is problematic: Repeated failures may cause majorities to further split up, leaving the system without a primary component. To overcome this problem, the *dynamic voting* paradigm was suggested.

The dynamic voting paradigm defines quorums adaptively: When a partition occurs, if a majority of the previous quorum is connected, a new and possibly smaller quorum is chosen. Thus, each newly formed quorum must contain a majority of the previous one, but not necessarily a majority of the sites. Stochastic models analysis [16], simulations [20], and empirical results [4] show that dynamic voting is more available than any other paradigm for maintaining a primary component.

Another important benefit of the dynamic voting paradigm is in its flexibility to support a dynamically changing set of processes. With emerging world-wide communication technology, new applications wish to allow users to freely join and leave. Using dynamic voting, such systems can dynamically account for the changes in the set of participants.

In this paper we present a robust and efficient protocol for maintaining a primary component using dynamic voting in an asynchronous environment, where processes and communication links may fail. By recording historical information, our protocol allows the system to make progress where previously suggested protocols either block or require a cold start of the entire system, or lead to inconsistencies. Our protocol's communication and memory requirements are small and it is simple to implement. It may be incorporated in many distributed applications that make progress in a primary component, e.g., replication algorithms [18, 11], transaction management [17], and even infrastructure systems like the ISIS toolkit [5].

If a failure occurs in the course of the protocol, some previously suggested protocols (e.g., [16, 1]) block until all the members of the last quorum become reconnected, while our protocol requires only a majority of the members that attempted to form the last quorum to become reconnected in order to make progress. Blocking until all the members reconnect significantly reduces the availability, especially in failure-prone environments (for which dynamic voting is most suitable) and in applications with a dynamic set of participants, where a waited upon process might have voluntarily left the system. Furthermore, the analyses of the availability of dynamic voting do not take the possibility of blocking into consideration, and therefore the actual availability of these protocols is lower than expected.

Unlike some previous protocols, e.g., the dynamic voting protocols implemented in ISIS and Horus[2] [21], our protocol recovers from situations in which the primary component was lost (e.g., when the primary component partitions into three minority groups) without requiring a cold start of the entire system.

*This work was supported by the United States - Israel Binational Science Foundation, Grant No. 92-00189 and by the Israeli Ministry of Science.

[1] A component is sometimes called a partition. In our terminology, a partition splits the network into several components.

[2] The [21] protocol was implemented in a previous version of Horus. The current membership protocol of Horus is majority based, and does not employ dynamic voting.

To appear in the *ACM Symp. on Prin. of Distributed Computing (PODC 97), August 1997.*

The challenge in designing consistent dynamic voting protocols is in coping with failures that occur while the processes are trying to form a new primary component (i.e., form a new quorum). Uncareful handling of such cases may lead to inconsistencies when there are different knowledge levels at different sites. When partitions occur, such knowledge differences are inevitable: Once a site detaches, it is impossible for other sites to know whether it received a specific message before its detachment, or not. Some past protocols (e.g., [9, 20, 11]) lead to inconsistent results in such cases, as demonstrated by the following typical scenario:

- The systems consists of five processes: $a, b, c, d$ and $e$. The system partitions into two components: $a, b, c$ and $d, e$.
- $a, b$ and $c$ try to form a new quorum. To this end, they exchange messages.
- $a$ and $b$ form the quorum $\{a, b, c\}$, assuming that process $c$ does so too. However, $c$ detaches before receiving the last message, and therefore is not aware of this quorum.
- $a$ and $b$ notice that $c$ detached, therefore form a new quorum $\{a, b\}$ which is a majority of $\{a, b, c\}$.
- Concurrently, $c$ connects with $d$ and $e$, and they form the quorum $\{c, d, e\}$ (a majority of $\{a, b, c, d, e\}$).
- The system now contains two live quorums, which may lead to inconsistencies.

Our protocol overcomes the difficulty demonstrated in the scenario above by maintaining another level of knowledge. The protocol guarantees that if $a$ and $b$ succeed in forming $\{a, b, c\}$, then $c$ is aware of this possibility. From $c$'s point of view, the quorum $\{a, b, c\}$ is ambiguous: It might have or might have not been formed by $a$ and $b$. In general, every process records, along with the last quorum it formed, later quorums that it attempted to form but detached before actually forming them. These ambiguous quorums (or ambiguous attempts) are taken into account in later attempts to form a quorum. Some previously suggested protocols avoid inconsistencies by running Two Phase Commit ([16, 13]), or similar mechanisms ([1]) that cause processes to block when their latest quorum is ambiguous. These protocols do not record historical information, and therefore, in case of failures, must consider all possible histories. This imposes severe limitations on the system's progress.

In [19], a three phase consensus protocol [7] is employed in order to allow a majority to resolve ambiguous quorums. This protocol is similar to the majority based Three Phase Commit (3PC) [22, 17] protocols. This induces a high overhead that makes the protocol infeasible for use in practice: When a majority of the previous quorum reconnects, [19] requires at least five communication rounds in order to resolve the previous quorum and form a new one. Our protocol avoids such excessive communication by using pipelining: the status of past ambiguous quorums is resolved during the installation of new quorums. Thus, when a majority of the previous quorum reconnects, only two communication rounds are required in order to form a new quorum. This requires our protocol to record several ambiguous quorums in case failures cascade.

Unfortunately, recording all ambiguous quorums is not feasible: The number of ambiguous quorums a process might need to record may be exponential in the number of participating processes. In Section 7 we rule-out a simple mechanism that records only the latest ambiguous quorums; we demonstrate that in order to preserve consistency it may be necessary to consider an ambiguous quorum even if it is followed by an exponential number of ambiguous attempts. Taking a huge number of quorums into consideration limits the possibility of progress in the system, and may cause the system to block. An important contribution of our work is in providing a simple "garbage collection" mechanism for reducing the number of quorums that a process needs to record to at most $n$, where $n$ is the number of processes in the system. Practically, the number of quorums a process may need to consider is expected to be very small. Thus, our protocol achieves a good balance between the historical data it stores, the restrictions on the ability to make progress in the system and the number of communication rounds.

The main criticism of the dynamic voting paradigm is that there can be situations where almost all of the processes in the system are connected but cannot form a new quorum because of the potential existence of a past surviving quorum held by a single process. To prevent such situations, our protocol sets a lower bound, $x$, on the size of quorums. This way, every component containing more than $n - x$ members (where $n$ is the number of processes in the system) can always form a quorum, regardless of past events in the system. We developed a novel mechanism for providing this feature in environments that allow new processes to join on the fly.

The rest of this paper is organized as follows: In Section 2 we formally define the requirements of a primary component maintenance service. In Section 3 we describe the computation environment model. The primary component protocol is described in Section 4. In Section 5 we provide an intuition to the correctness of the protocol, and in Section 6 we evaluate its efficiency. In Section 7 we rule-out a trivial and efficient garbage collection mechanism. In Section 8 we present a novel mechanism that always allows connected components of a larger size than a user-defined threshold to form a primary component, in environments that allow new processes to join on the fly. Section 9 concludes the paper.

## 2 Problem Definition

In this paper we present a primary component maintenance service, that allows a group of processes to *form* a primary component in a consistent way. Such a service is required to impose a total order on all the primary components formed in the system. When using a static quorum system, the order is easily provided using the following property: "every two primary components intersect". Unfortunately, dynamic quorum systems do not possess this property. Instead, a total order on primary components is defined by extending the causal order on components that do intersect.

Formally: Let $P$ and $P'$ be two primary components. If $j \in P \cap P'$, and $j$ participates in both of these primary components, i.e., attempts to form both, then $j$ participates in one of $P$ and $P'$ before the other[3]. If $j$ participates in $P$ first, we denote the transitive closure of this relation by: $P \prec P'$. The requirement from a dynamic paradigm for maintaining primary components is that $\prec$ is a total order. Since a process is a member of at most one component at any given time, the total order on primary components implies that at any given time there is at most one live and

[3]A process does not participate in two quorums concurrently.

connected primary component in the system.

## 3 The Model

The initial primary component in the system consists of a *core* group of processes, $\mathcal{W}_0$, that is known to all the processes in $\mathcal{W}_0$. The core group is typically the initial configuration on which the system manager runs a protocol (e.g., the sites running a distributed database). The set of all the processes that may run the protocol is unknown to any of the processes in advance, and thus the configuration may change dynamically. Processes that do not belong to $\mathcal{W}_0$ are aware of the fact that they are not members of the core group.

The processes are connected by an asynchronous communication network. The system model allows for the following communication network changes: messages may be lost, failures may partition the network into disjoint components, and previously disjoint components may re-merge. Sites may crash and recover; recovered processes come up with their stable storage intact[4].

Maintaining the primary component is typically decoupled into two separate problems: first, determining the set of connected processes, and second, deciding whether a set of processes is the primary component. Like other dynamic voting protocols, our paper focuses on solving the latter problem, assuming a separate mechanism that solves the former.

Our dynamic voting protocol assumes a membership mechanism no stronger than those assumed in [9, 14, 20, 16, 11, 1]. Each process is equipped with an underlying membership module, e.g., [2, 3, 12, 8]. When the membership module senses failures or recoveries, it reports to the process of the new membership, i.e., the set of processes that are currently assumed to be connected. Furthermore, our protocol assumes that every message is received in the membership in which it was sent. This can be achieved either by refraining from sending messages while a membership change takes place, or by discarding old messages that arrive after a membership change. In order to discard messages from previous memberships, the protocol needs to provide a unique membership identifier (which will be added to all the messages sent in this membership). These requirements may be fulfilled by a simple and efficient membership protocol, e.g., the one round protocol in [8], which terminates after one communication round.

As shown in [6], it is impossible to reach agreement upon the current membership in an asynchronous system. Hence, we do not assume that the membership reports accurately reflect the network situation, nor is the membership reported atomically to all the processes. The dynamic voting protocol we present is *correct* (i.e., guarantees a total order on primary components) regardless of whether the membership mechanism is live and accurate or not. The *liveness* of the protocol (its ability to form new primary components when the network situation changes) depends on the accuracy and liveness of this membership mechanism.

## 4 The Primary Component Protocol

We present a protocol for maintaining the primary component in an asynchronous system. Initially, the primary component in the system is the core group, $\mathcal{W}_0$. Whenever a membership change is reported, the notified members invoke a new session of the protocol, trying to *form* a new primary component. If they succeed, then at the end of the session they form a new primary component $P$, which persists until the next membership change. Each process independently invokes the protocol once it receives the membership message.

In this section, we present a simplified version of the protocol in which the members of the core group, $\mathcal{W}_0$, have a special status: every quorum in the system must contain a threshold of members from $\mathcal{W}_0$. In Section 8 we modify the protocol to eliminate this special status.

### 4.1 Dynamic Quorums

Our protocol uses dynamic voting to determine when a group of processes is an eligible quorum. Originally, dynamic voting was implemented by allowing a majority of the previous quorum to become the new quorum. *Dynamic linear voting* [14], optimizes this scheme by breaking ties between groups of equal size using a linear order, $\mathcal{L}$, imposed on all the potential processes in the system. We extend dynamic linear voting with another parameter: *Min_Quorum*, the minimum quorum size allowed in the system. Imposing a minimum quorum size allows large groups of processes to be eligible quorums regardless of the system's history.

We define a predicate $Sub\_Quorum(S,T)$, that is TRUE iff $T$ can become the new quorum in the system, given that the previous quorum was $S$. Formally, $Sub\_Quorum(S,T)$ is TRUE iff:

1. $|T \cap \mathcal{W}_0| \geq Min\_Quorum$, and
2. 
   - $|T \cap S| > |S|/2$, or
   - $|T \cap S| = |S|/2$ and $\exists p \in T \cap S$ such that $\forall q \in S \setminus T$ $\mathcal{L}(p) > \mathcal{L}(q)$, or
   - $|T \cap \mathcal{W}_0| > |\mathcal{W}_0| - Min\_Quorum$.

It is easy to see that the dynamic linear voting scheme has the following properties:

1. If $Sub\_Quorum(S,T)$ then $S \cap T \neq \emptyset$.
2. If $Sub\_Quorum(S,T)$ and $Sub\_Quorum(S,T')$ then $T \cap T' \neq \emptyset$.

Note that every quorum in the system must contain at least *Min_Quorum* members of $\mathcal{W}_0$. In Section 8 we relax this restriction, and require, instead, that a quorum will contain *Min_Quorum* processes.

### 4.2 Variables and Notation

The protocol is conducted in sessions, and the sessions are numbered. A session $S$ of the protocol is identified by its membership, $S.M$, and session number, $S.N$. Each process $p$ maintains the following variables:

$Is\_Primary_p$ is a boolean variable that is TRUE iff the current membership is the primary component in the system. If $p \in \mathcal{W}_0$, then it is initialized to TRUE, and otherwise to FALSE.

$Session\_Number_p$ is the current session number. This variable is initialized to 0.

[4] If the stable storage is destroyed because of a severe disk error, the protocol remains correct but its availability is reduced.

$Last\_Primary_p$ is the last primary component that process $p$ formed (i.e., the membership and *Session_Number* of the session in which the last primary component was formed). If $p \in \mathcal{W}_0$ then it is initialized to $(\mathcal{W}_0, 0)$ and otherwise to $(\infty, -1)$[5].

$Ambiguous\_Sessions_p$ is the set of ambiguous sessions that process $p$ attempted to form after $p$ participated in $Last\_Primary_p$. For each ambiguous session (or attempt) $S$ in this set, $p$ maintains an associative array $S.A$. For every $q \in S.M$: if $p$ knows that $q$ formed $S$ then $S.A(q) = 1$; if $p$ knows that $q$ did not form $S$ then $S.A(q) = -1$ and otherwise $S.A(q) = 0$. The set of ambiguous sessions is initially empty.

$Last\_Formed_p$ is an associative array. For each $q$ that $p$ participated in a session with, $Last\_Formed_p(q)$ is the last session (membership and number) that $p$ formed and $q$ was a member of. Initially, if $p, q \in \mathcal{W}_0$ then $Last\_Formed_p(q)$ is $(\mathcal{W}_0, 0)$. Otherwise, it is $(\infty, -1)$.

We use the following notation:

- $\mathcal{M}$ is the membership as reported in the membership message that invoked the current session of the protocol. The membership is a list of processes.
- $Max\_Session$ is: $\max_{p \in \mathcal{M}}(Session\_Number_p)$.
- $Max\_Primary$ is: $Last\_Primary_p$ *s.t.* $Last\_Primary_p.N = \max_{q \in \mathcal{M}}(Last\_Primary_q.N)$.
- $Max\_Ambiguous\_Sessions$ is: $\bigcup_{p \in \mathcal{M}}(A \in Ambiguous\_Sessions_p$ s.t. $A.N > Max\_Primary.N)$.

In order to simplify notations, we extend the definition of the *Sub_Quorum* predicate to sessions. For a pair of sessions $S1, S2$, the predicate $Sub\_Quorum(S1, S2)$ is defined to be: $Sub\_Quorum(S1.M, S2.M)$.

### 4.3 The Protocol

Whenever a membership change is reported, the notified members invoke a new session of the protocol. Each session of the protocol is conducted in three steps: In the first step the connected processes exchange information about quorums in past sessions. In case the membership protocol involves message exchange among the members, this information can be piggybacked onto the membership protocol messages, thus no extra communication round is needed. The second step is the *attempt* step. Each process uses the information it received in the first step to make an independent decision whether the current membership is an eligible quorum. If it is, the member *attempts* to form the session: it computes the session number, records the session and sends an attempt message to the rest of the members. In the last step, the processes *form* the new quorum: They declare the session as a primary component, and no longer record preceding sessions.

If a process receives a membership message in the course of a session, it aborts the session and invokes a new session. Once the membership stabilizes, sessions are no longer aborted. If the expected messages fail to arrive from some of the members, then the primary component protocol is blocked until a membership change is reported.

[5]We extend the definition of the *Sub_Quorum* predicate so that $Sub\_Quorum(\infty, T)$ is FALSE for every set $T$.

Intuitively, the purpose of the attempt step is to guarantee that if a process $p$ forms a session $\mathcal{F}$, then all the other members of $\mathcal{F}$ recorded $\mathcal{F}$ as an ambiguous session. Thus, if some members of $\mathcal{F}$ detach before the last step, they will take $\mathcal{F}$ into account in future attempts to form a quorum.

In order to avoid recording an exponential number of ambiguous sessions, our protocol employs a "garbage collection" mechanism that reduces the number of ambiguous sessions recorded concurrently to the number of processes in the system. A process deletes ambiguous sessions when it *resolves* their status, i.e., discovers whether they were formed by any member or not. In order to resolve a session, a process needs to *learn* about the session status at other members. The rules for learning and resolving ambiguous sessions are described in Section 4.4. These rules are employed during the attempt step. The protocol is formally described in Figure 1.

In each step of the protocol, when a process changes any of its private variables, it must write the change to a stable storage before responding to the message that caused the change[6]. The primary component formed in Step 3 remains the primary component in the system until another membership change occurs. Notice that when a process forms a primary component, it no longer stores previous ambiguous sessions.

The protocol presented here is efficient: In each session of the protocol, each process sends two multicast messages, one of which may be piggybacked on a membership protocol message. The protocol is symmetric: processes multicast messages to all other processes. Such a protocol is efficient assuming a hardware broadcast/multicast mechanism. For networks in which efficient multicast is not available, it is straightforward to convert our protocol to work in a centralized fashion by appointing a coordinator for each session. The coordinator may be chosen deterministically, for example, the first member of the current membership (in lexicographical order). In the centralized version, each process sends two messages to the coordinator, and the coordinator multicasts two messages to the other processes.

### 4.4 Resolving Ambiguous Sessions

A process can delete ambiguous sessions upon resolving their status. The resolution is based on determining whether an ambiguous session was formed by one of its members: If an ambiguous session was not formed by any of its members, then it is safe to delete it from *Ambiguous_Sessions*. On the other hand, if an ambiguous session was formed by some member, then the other members *adopt* this session: They declare the session as a primary component, and no longer record ambiguous sessions that precede it. The resolution rules are described in Figure 2.

The resolution rules require a process $p$ to *learn* whether an ambiguous session that $p$ records was formed by one of its members. This is achieved by collecting the session status from other session members during the first step of future sessions of the protocol. Process $p$ applies the information it gathered to its *Ambiguous_Sessions* set during the attempt step. The learning rules are formally described in Figure 3.

[6]If the storage is destroyed because of a severe disk crash, the process may recover with its $Last\_Primary = (\infty, -1)$. This limits the availability, but does not affect the correctness.

1. Set *Is_Primary* to FALSE.
   Send your *Session_Number*, *Ambiguous_Sessions*, *Last_Primary* and *Last_Formed* to all the members of $\mathcal{M}$.

2. *Attempt step:* Upon receiving this information from all members of $\mathcal{M}$:
   - Update *Ambiguous_Sessions* according to the learning rules described in Figure 3.
   - Apply the resolution rules described in Figure 2.
   - Compute *Max_Session*, *Max_Primary*, and *Max_Ambiguous_Sessions*.
   - **if** $(Sub\_Quorum(Max\_Primary.M, \mathcal{M})$ and
     $((\forall S \in Max\_Ambiguous\_Sessions)\ \ Sub\_Quorum(S.M, \mathcal{M})))$
     **then** "attempt the session:"
     - Set *Session_Number* to *Max_Session+1*.
     - Append to *Ambiguous_Sessions* the session $S = (\mathcal{M}, Session\_Number)$, with $S.A(q) = 0$ for every $q \in S.M$ s.t. $q \neq p$, and $S.A(p) = -1$.
     - Send attempt message to every member of $\mathcal{M}$.

     **else** terminate this session with *Is_Primary*=FALSE.

3. *Form step:* Upon receiving an attempt message from all members of $\mathcal{M}$ set:
   - $Last\_Primary = (\mathcal{M}, Session\_Number)$, and
   - $Ambiguous\_Sessions = \emptyset$, and
   - *Is_Primary*=TRUE, and
   - $\forall q \in \mathcal{M}\ \ Last\_Formed_p(q) = Last\_Primary$.

Figure 1: A Session of the Protocol Executed by Process $p$

**The Resolution Rules:**

**Adoption** If $p \in \mathcal{F}.M$ and $Last\_Primary_p.N < \mathcal{F}.N$, and $p$ learns that session $\mathcal{F}$ was formed by one of its members then:
Process $p$ sets $Last\_Primary_p$ to $\mathcal{F}$ and $\forall q \in \mathcal{F}.M$ $p$ sets $Last\_Formed_p(q) = \mathcal{F}$.

**Deletion** If $p$ learns that an ambiguous session $S$ was not formed by any of its members, or if $p$ learns that a session $\mathcal{F}$, where $\mathcal{F}.N \geq S.N$ and $p \in \mathcal{F}.M$, was formed by one of its members, then:
Process $p$ deletes $S$ from $Ambiguous\_Sessions_p$.

Figure 2: The Resolution Rules

## 5 Correctness of the Protocol

In this section, we provide an intuition to the correctness of the protocol. In Section 5.1 we show how our protocol overcomes the typical problematic scenario described in Section 1.

In Section 5.2 we outline the protocol's correctness proof. The detailed correctness proof appears in the full paper [10]. In the correctness proof, we do not rely on the accuracy of the underlying membership service; the protocol provides a total order on primary components even when the membership does not correctly reflect the network situation.

### 5.1 Overcoming The Typical Problematic Scenario

We now demonstrate that our protocol overcomes the problematic scenario of Section 1.

- The systems consists of five processes $a, b, c, d, e$. The system partitions into two components: $a, b, c$ and $d, e$.
- $a, b$ and $c$ try to form a new quorum. To this end, they exchange messages.
- $a$ and $b$ form the quorum $\{a, b, c\}$, assuming process $c$ does so too. However, $c$ detaches before receiving the last message, and therefore does not form this quorum. Yet, $c$ records this session in $Ambiguous\_Sessions_c$.
- $a$ and $b$ notice that $c$ detached, therefore form a new quorum $\{a, b\}$ which is a majority of $\{a, b, c\}$.
- Concurrently $c$ connects with $d$ and $e$. $\{c, d, e\}$ is not a majority of $\{a, b, c\}$ that $c$ records, therefore $c, d$, and $e$ cannot form a new quorum.

The system contains only one live quorum $\{a, b\}$.

### 5.2 Correctness Proof Outline

In order to show that the protocol is correct, we have to show that the transitive closure of the order between intersecting formed sessions[7] is a total order. The correctness proof is based on the following claims:

1. Every two intersecting formed sessions have different session numbers.
2. If two attempted sessions have a common attempt in $Max\_Ambiguous\_Sessions \cup Max\_Primary$ then these sessions intersect.

[7] A *formed session* is a session that at least one of its members has formed. The initial primary component $(\mathcal{W}_0, 0)$ is also considered a formed session.

**Process $p$ learns the status of process $q$ w.r.t. session $S$** during a session $S'$, where $S.N < S'.N$ and $p, q \in S.M \cap S'.M$, if during $S'$ $p$ executes the attempt step. Process $p$ learns accordingly:

- If $\mathit{Last\_Formed}_q(p).N = S.N$ then $p$ learns that process $q$ formed session $S$.
- If $\mathit{Last\_Formed}_q(p).N < S.N$ then $p$ learns that process $q$ did not form session $S$.

**Process $p$ learns that session $S$ was not formed by any of its members** if:

- $p$ doesn't form $S$, and learns from all the other session members that they did not form session $S$ either, or
- There exist a session $S'$ and a process $q$, where $S.N < S'.N$ and $p, q \in S.M \cap S'.M$, such that during $S'$, $q$ does not consider $S$ to be ambiguous or formed. Formally:
  - $\mathit{Last\_Primary}_q.N < S.N$ or ($\mathit{Last\_Primary}_q.N = S.N$ and $\mathit{Last\_Primary}_q.M \neq S.M$), and
  - $S \notin \mathit{Ambiguous\_Sessions}_q$, and
  - $p$ executes the attempt step of the protocol during $S'$.

Figure 3: Learning Rules

3. Let $S$ be a formed session, and let *Max_Primary* computed in $S$ be $\mathcal{F}$. Let $F_0, F_1, ..., F_k$ be a sequence of formed sessions (ordered by session numbers) s.t. $\mathcal{F} = F_0$, and for every $0 < i \leq k$, $\mathcal{F}.N < F_i.N < S.N$, and $F_{i-1}$ and $F_i$ intersect. Then, $\mathit{Sub\_Quorum}(\mathcal{F}_k, S)$ holds.

4. Let $S$ be an attempted session, and let $\mathcal{F}$ be a formed session, s.t. $\mathcal{F}.N$ is the maximal value among formed sessions with a sessions number smaller than $S.N$. Then, $\mathcal{F}$ is the only formed session with this session number, and $\mathit{Sub\_Quorum}(\mathcal{F}, S)$ holds.

The third and fourth claims are proved by induction. It is derived from the fourth claim that every formed session has a unique session number, and that two successive formed sessions intersect. We conclude that the transitive closure of the order between intersecting formed sessions is a total order.

## 6 Evaluating the Efficiency

Our protocol assumes a simple underlying membership protocol, which may be conducted in one communication round (e.g., [8]). Each session of our protocol is conducted in two communication rounds, one of which may be conducted by piggybacking information on the messages sent by the membership protocol. Thus, in each session of the symmetric protocol, a total of $2n$ messages are multicast by all the processes, where $n$ is the number of processes participating in this session. In the centralized version of the protocol, a total of $4n$ point to point messages are sent. Once the membership stabilizes, our protocol terminates within one session, in which it resolves past ambiguous quorums and also forms a new quorum (if possible).

In this section we prove that a process records concurrently at most $n$ ambiguous sessions in the worst case. In practice, the number of ambiguous sessions is expected to be very small, since whenever a new quorum is successfully formed, all the ambiguous sessions are discarded.

We now prove that if a process $p$ attempts to form two ambiguous sessions with a process $q$, then during the later session $p$ can learn $q$'s status w.r.t. the former session. Note that after $p$ learns a session's status as recorded by every session member, $p$ can resolve the status of a session. Therefore, in case $p$ cannot resolve a session's status, there is at least one session member with which $p$ does not share a later attempt. This property linearly bounds the number of unresolved ambiguous sessions a process records concurrently.

**Lemma 1** *At each process, the value of Session_Number is increased whenever the process attempts to form a session.*

**Proof**: Immediate from the protocol. □

**Lemma 2** *Let $p$ be a process and $A_1, A_2$ two ambiguous sessions, such that $A_1.N < A_2.N$ and both $A_1$ and $A_2$ are in $\mathit{Ambiguous\_Sessions}_p$. If there exists a process $q$ such that $q \in A_1.M \cap A_2.M$, then $p$ learned whether $q$ formed session $A_1$ before $p$ attempted to form session $A_2$.*

**Proof**: By induction on the difference $A_2.N - A_1.N$.

- Base case: $A_2.N - A_1.N = 1$. According to the protocol, a process attempts to form a session in Step 2 of the protocol, after the process received the *Last_Formed* arrays from all session members and applied the learning rules as follows:
  1. If $\mathit{Last\_Formed}_q(p).N < A_1.N$ then $p$ learned that $q$ did not form $A_1$.
  2. If $\mathit{Last\_Formed}_q(p).N = A_1.N$ then $p$ learned that $q$ formed $A_1$.

  Notice that $\mathit{Last\_Formed}_q(p).N > A_1.N$ is impossible; Otherwise, process $q$ formed $\mathit{Last\_Formed}_q(p)$, and process $p$ attempted to form it. Hence, by Lemma 1, $A_1.N < \mathit{Last\_Formed}_q(p).N < A_2.N$, in contradiction with *Session_Number* being an integer.

- General case: We assume the induction hypothesis holds for $A_2.N - A_1.N < k$, and prove for $A_2.N - A_1.N = k$. Since $A_2 \in \mathit{Ambiguous\_Sessions}_p$, $p$ received $\mathit{Last\_Formed}_q(p)$ during session $A_2$, and learned as follows:
  1. If $\mathit{Last\_Formed}_q(p).N \leq A_1.N$ then, as in the base case, $p$ learned whether $q$ formed $A_1$.
  2. Otherwise, $\mathit{Last\_Formed}_q(p).N > A_1.N$. Hence, there exists a formed session $F_i$ such that:
     - $A_1.N < F_i.N < A_2.N$, and

- $p, q \in F_i.M$, and
- $q$ formed $F_i$.

According to the protocol, since q formed $F_i$, $F_i \in Ambiguous\_Sessions_p$ upon ending $F_i$. Moreover, $F_i.N - A_1.N < k$. Hence from the induction hypothesis $p$ learned whether $q$ formed $A_1$ before attempting to form $F_i$, hence before attempting to form $A_2$. □

**Theorem 1** *Process $p$ records concurrently at most $n - Min\_Quorum + 1$ ambiguous sessions, where $n$ is the number of processes that participate in an execution of the protocol.*

**Proof**: Assume to the contrary that $p$ concurrently records in $Ambiguous\_Sessions_p$ $n - Min\_Quorum + 2$ ambiguous sessions, $A_1, \ldots, A_{n-Min\_Quorum+2}$, such that $(\forall 1 \leq i < n - Min\_Quorum + 2)$ $A_i.N < A_{i+1}.N$[8]. Since $A_i$ is still ambiguous, $p$ did not learn whether some member of $A_i$ formed it or not. Hence, by Lemma 2, there is at least one member of $A_i$ that is not a member of any session $A_j \in Ambiguous\_Sessions_p$ for $j > i$. Consequently, for each $i$, there are at least $i$ processes that do not participate in any session $A_j$ where $j > i$. In particular, after recording sessions $A_1, \ldots, A_{n-Min\_Quorum+1}$, there are at least $n - Min\_Quorum + 1$ members that are not members of $A_{n-Min\_Quorum+2}$. Therefore $A_{n-Min\_Quorum+2}.M$ consists of less than $Min\_Quorum$ members, and $A_{n-Min\_Quorum+2}$ is not a legal session, a contradiction. □

## 7 Ruling Out a Trivial Approach

The the typical problematic scenario depicted in Section 1 raised the need to consider attempts to form a session, even though they didn't succeed. Still it might seem that it is enough that each member records only the last attempt it failed to form, instead of recording a list of attempts. The example below demonstrates that this approach does not work. In fact, we show that in order to preserve consistency it may be necessary to consider an attempt even if it is followed by an exponential number of attempts.

Consider the following execution:

- The core group $\mathcal{W}_0$ consists of processes $p_1, \ldots, p_{4k}$.
- For each process, $Last\_Primary = (\mathcal{W}_0, 0)$, and $Ambiguous\_Sessions = \emptyset$.
- In session $S_1 = (\{p_1, \ldots, p_{2k+1}\}, 1)$, $p_{2k+1}$ forms session $S_1$, while $p_1, \ldots, p_{2k}$ attempt to form $S_1$ and detach before actually forming it.
- Let $\mathcal{G} = \mathcal{W}_0 \setminus S_1.M$. Denote: $\Lambda = \binom{2k-1}{k}$. There are $\Lambda$ different sub-sets of $\mathcal{G}$ of size $k$, denoted $G_1, \ldots, G_\Lambda$,. Note that $\Lambda$ is exponential in $|\mathcal{W}_0|$.

The execution continues with sessions $S_2, \ldots, S_\Lambda$ s.t. $\forall i$, $2 \leq i \leq \Lambda$, session $S_i$ is conducted as follows:

- $S_i$'s session number is i.
- $S_i$'s membership is $\{p_1, \ldots, p_{k+1}\} \cup G_{i-1}$.
- $Max\_Primary$ of $S_i$ is $(\mathcal{W}_0, 0)$.
- Conditions: $Sub\_Quorum(\mathcal{W}_0, S_i.M)$ is TRUE, and $\forall j < i$ $Sub\_Quorum(S_j, S_i)$ is TRUE.
- Actions: $p_1, \ldots, p_k$ append $S_i$ to their $Ambiguous\_Sessions$ sets. All other members of $S_i$ detach before performing the Attempt Step, therefore do not append $S_i$ to their $Ambiguous\_Sessions$ sets.

If $p_1, \ldots, p_k$ delete $S_1$ from their $Ambiguous\_Sessions$ sets after $\Lambda$ sessions then two primary components may be formed concurrently, as shown below.

- Session $S_{\Lambda+1}$ whose membership consists of $\{p_{k+1}, \ldots, p_{2k+1}\}$ is a sub-quorum of $S_1$, which is $Max\_Primary$ according to $p_{2k+1}$ (hence a sub-quorum of $\mathcal{W}_0$ is not needed). Therefore it is a legal new quorum, and all its members form it successfully.
- Session $S'_{\Lambda+1}$ whose membership consists of $\mathcal{W}_0 \setminus S_{k+2}.M$ is a sub-quorum of both $(\mathcal{W}_0, 0)$ and of sessions $S_2$ to $S_\Lambda$. Therefore this session is also a legal new quorum, and all its members form it successfully.

Hence, a trivial garbage collection mechanism that deletes only a number of least recent sessions cannot reduce the number of recorded sessions to sub-exponential, while preserving consistency. Our protocol uses a sophisticated garbage collection mechanism that reduces the number of recorded sessions to be at most linear in the number of processes.

## 8 Dynamically Changing Quorum Requirements

The definition of $Sub\_Quorum$ presented in Section 4.1 requires every quorum to contain at least $Min\_Quorum$ members of $\mathcal{W}_0$, in order to always allow a group of more than $|\mathcal{W}_0| - Min\_Quorum$ members of $\mathcal{W}_0$ to make progress. This requirement restricts the availability if some members of $\mathcal{W}_0$ leave the system. In this section we eliminate the special status of the members of $\mathcal{W}_0$: we always allow a group of more than $n - Min\_Quorum$ processes to make progress, where $n$ is the "current" number of processes in the system. We present a novel mechanism for providing this feature, in environments which allow new processes to join on the fly.

Allowing $n$ to change dynamically is subtle because the truth value of the $Sub\_Quorum$ predicate changes with time. For example, $Sub\_Quorum(S, T)$ may be initially TRUE because $T$ contains more than $|\mathcal{W}_0| - Min\_Quorum$ members of $\mathcal{W}_0$, but later, as the set of participants increases, the truth value of $Sub\_Quorum(S, T)$ may become FALSE. Therefore, $n$ must be increased with care, and new processes may not immediately be taken into account. New processes are inserted into the "set of participants" using the two new variables described below:

$\mathcal{W}$ is the set of participants taken into account in the new $Min\_Quorum$ requirement. $\mathcal{W}$ is initialized to $\mathcal{W}_0$, and new processes are inserted into this group when they participate in a formed session.

$\mathcal{A}$ is the set of processes that have not been admitted into $\mathcal{W}$ yet. $\mathcal{A}$ is initialized to the empty set if $p \in \mathcal{W}_0$, and otherwise to contain $p$ itself.

These variables are used to evaluate the $Sub\_Quorum$ predicate. Below we describe how these variables are maintained in the course of the primary component protocol (cf. Section 4). At the beginning of each step in a session $S$ of the protocol, every process $p$ executes the following operations:

[8] By Lemma 1, the order requirement is always fulfilled.

1. In the first step, $p$ sends $\mathcal{W}_p$ and $\mathcal{A}_p$ to every member of $S$.
2. *The Attempt Step:* Upon receiving responses from every member of $S$, $p$ updates $\mathcal{W}_p$ and $\mathcal{A}_p$ as follows:
   - Set $\mathcal{W}_p$ to $\bigcup_{q \in S.M} \mathcal{W}_q$.
   - Set $\mathcal{A}_p$ to $(\bigcup_{q \in S.M} \mathcal{A}_q) \setminus \mathcal{W}_p$.

   The *Min_Quorum* requirement is evaluated as follows:
   - $S$ is an eligible quorum only if $|S.M \cap W_p| \geq Min\_Quorum$.
   - If $|S.M \cap (\mathcal{W}_p \cup \mathcal{A}_p)| > |\mathcal{W}_p \cup \mathcal{A}_p| - Min\_Quorum$, then for every session $S'$ that $p$ records, the truth value of $Sub\_Quorum(S', S)$ is TRUE, regardless of past quorums.
3. *The Form Step:* Upon receiving an attempt message from every member of $S$:
   - Set $\mathcal{W}_p$ to $\mathcal{W}_p \cup (\mathcal{A}_p \cap S.M)$.
   - Set $\mathcal{A}_p$ to $\mathcal{A}_p \setminus S.M$.

This mechanism allows the system to adjust the quorum requirements in the protocol to the dynamically changing set of process. We prove the correctness of the resulting protocol in the full paper [10].

Jajodia and Mutchler [15] suggest a similar idea in their hybrid algorithm. The hybrid approach combines dynamic voting in large quorums with static voting in quorums of size three, ruling out quorums consisting of a single process. Neither approach is strictly better than the other: There are situations in which our approach allows the system to make progress while the hybrid approach does not, and vice versa. However, the hybrid algorithm of [15] applies the hybrid approach to the algorithm of [16] and uses two phase commit to avoid inconsistencies.

## 9 Conclusions

We presented a dynamic voting protocol for consistently maintaining primary components in an asynchronous failure-prone system. Our protocol is more available than previously suggested protocols, in that it allows progress in more cases. The protocol is efficient and does not require a cold start of the system in order to recover from severe system failures.

Our protocol always allows connected components of a larger size than a user-defined threshold to form a primary component. We developed a novel mechanism for providing this feature in environments that allow new processes to join on the fly.

## References


[1] Amir, Y. *Replication Using Group Communication Over a Dynamic Network.* PhD thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1995.

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

[3] Amir, Y., Moser, L. E., Melliar-Smith, P. M., Agarwal, D. A., and Ciarfella, P. Fast Message Ordering and Membership using a Logical Token-Passing Ring. In *International Conference on Distributed Computing Systems* (May 1993), no. 13, pp. 551–560.

[4] Amir, Y., and Wool, A. Evaluating Quorum Systems over the Internet. In *The Fault-Tolerant Computing Symposium(FTCS)* (June 1996), pp. 26–35.

[5] Birman, K., and Van Renesse, R. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.

[6] Chandra, T., Hadzilacos, V., Toueg, S., and Charron-Bost, B. On the Impossibility of Group Membership. In *ACM Symp. on Prin. of Distributed Computing (PODC)* (May 1996), pp. 322–330.

[7] Chandra, T. D., and Toueg, S. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of ACM 43*, 2 (Mar. 1996), 225–267.

[8] Cristian, F., and Schmuck, F. Agreeing on Process Group Membership in Asynchronous Distributed Systems. Tech. Rep. CSE95-428, Department of Conmputer Science and Engineering, University of California, San Diego, 1995.

[9] Davcev, D., and Burkhard, W. Consistency and Recovery Control for Replicated Files. In *ACM Symp. on Operating Systems Principles* (1985), no. 10, pp. 87–96.

[10] Dolev, D., Keidar, I., and Yeger Lotem, E. Dynamic Voting for Consistent Primary Components. TR 96-7, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, June 1996.

[11] El Abbadi, A., and Dani, S. A Dynamic Accessibility Protocol for Replicated Databases. *Data and Knowledge Engineering*, 6 (1991), 319–332.

[12] Ezhilchelvan, P. D., Macedo, A., and Shrivastava, S. K. Newtop: a Fault Tolerant Group Communication Protocol. In *International Conference on Distributed Computing Systems* (June 1995), no. 15, IEEE Computer Society Press, pp. 296–306.

[13] Herlihy, M. A Quorum-Consensus Replication Method for Abstract Data Types. *ACM Trans. Comp. Syst. 4*, 1 (Feb. 1986), 32–53.

[14] Jajodia, S. Managing Replicated Files in Partitioned Distributed Database Systems. In *IEEE Int'l. Conf. on Data Engineering* (1987), no. 3, pp. 412–418.

[15] Jajodia, S., and Mutchler, D. A Hybrid Replica Control Algorithm Combining Static and Dynamic Voting. *IEEE Transactions on Knowledge and Data Engineering 1*, 4 (Dec. 1989).

[16] Jajodia, S., and Mutchler, D. Dynamic Voting Algorithms for Maintaining the Consistency of a Replicated Database. *ACM Trans. Database Systems 15*, 2 (1990), 230–280.

[17] KEIDAR, I., AND DOLEV, D. Increasing the Resilience of Distributed and Replicated Database Systems. *The Journal of Computer and System Sciences (JCSS) special issue with selected papers from PODS 1995*. To Appear. Available in: `http://cs.huji.ac.il/ transis/ /Abstracts/jcss.html`. Previous version in the 1995 ACM-SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), May 1995, pages 245-254.

[18] KEIDAR, I., AND DOLEV, D. Efficient Message Ordering in Dynamic Networks. In *ACM Symp. on Prin. of Distributed Computing (PODC)* (May 1996), no. 15, pp. 68–76.

[19] MALLOTH, C., AND SCHIPER, A. View Synchronous Communication in large scale networks. In *Proceedings 2nd Open Workshop of the ESPRIT project BROADCAST (number 6360)* (July 1995 (also available as a Technical Report Nr. 94/84 at Ecole Polytechnique Fédérale de Lausanne (Switzerland), October 1994)).

[20] PARIS, J., AND LONG, D. Efficient Dynamic Voting Algorithms. *Proceedings 13th Int'l. Conf. on Very Large Data Bases* (1988), 268–275.

[21] RICCIARDI, A. M., AND BIRMAN, K. P. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *ACM Symp. on Prin. of Distributed Computing (PODC)* (August 1991), pp. 341–352.

[22] SKEEN, D. A Quorum-Based Commit Protocol. In *Berkeley Workshop on Distributed Data Management and Computer Networks* (Feb. 1982), no. 6, pp. 69–80.