Dynamic Voting for Consistent Primary Components[1]

Danny Dolev Idit Keidar Esti Yeger Lotem

Email: {dolev,idish,esti}@cs.huji.ac.il
Url: http://www.cs.huji.ac.il/{~dolev,~idish,~esti}

Institute of Computer Science
The Hebrew University of Jerusalem
Jerusalem, Israel

Technical Report CS96-7

June, 1996

[1]This work was supported by the United States - Israel Binational Science Foundation, Grant No. 92-00189

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 the system. In unreliable networks this can be problematic: Repeated failures may cause connected 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 [14], simulations [18], and empirical results [4] show that dynamic voting increases system availability, by increasing the probability that a primary component exists. In fact, the results in [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, many new applications (*e.g.* conferencing applications and interactive games) wish to allow users to freely join and leave, without restarting the entire system. 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 [16, 9], transaction management [15], 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.* [14, 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 offered by the dynamic voting paradigm, especially in failure-prone environments for which dynamic voting is most suitable. 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, and possibly even worse than the availability of static quorums. This approach is even more problematic in applications in which the set of participants is dynamic: one process that voluntarily leaves the system may cause all the other participants to block.

Unlike some previous protocols (*e.g.* the protocols implemented in ISIS and Horus [19]), 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.

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

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.* [8, 18, 9]) 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 the fact that this quorum is formed.
- 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 $\{c, d, e\}$.

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 are taken into account in later attempts to form a quorum. Some previously suggested protocols avoid inconsistencies by running Two Phase Commit ([14, 11]), 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 ability to make progress.

Our protocol uses ideas similar to those used in the majority based Three Phase Commit (3PC) [21, 15] protocols to allow a majority in the system to make progress. However, explicitly running the 3PC recovery protocol to resolve the status of past quorums before forming new ones would induce a high overhead that would make the protocol infeasible for use in practice, and also increase the chance of failure during the protocol. A similar idea was suggested in [17], where Chandra and Toueg's three phase consensus protocol [7] is employed. The status of past quorums is resolved before the installation of new ones. When a majority of the previous quorum reconnects, at least five communication rounds are needed in order to form a new quorum. In order to avoid such excessive communication, our protocol does not explicitly run a three phase recovery protocol for past ambiguous quorums; Their status 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.

Unfortunately, recording all ambiguous quorums is not feasible: The number of ambiguous quorums a process might need to record may be exponential, as demonstrated in Section 4.7. Furthermore, taking a huge number of quorums into consideration limits the possibility of progress in the system, and may cause the system to block. In Section 5 we present a simple "garbage collection" mechanism for reducing the number of quorums that a process needs to record to n, where n is the number of processes in the system, using local computation only; This mechanism does not require additional communication. Practically, the number of quorums a process may need to consider is expected to be very small. The resulting 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 on 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 small set of processes, or even 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. In our protocol, we incorporate a novel mechanism for providing this feature, in environments which allow new processes to join on the fly. Jajodia and Mutchler [13] suggest a similar idea in their hybrid algorithm. The hybrid algorithm combines dynamic voting in large quorums with static voting in quorums of size three, ruling out quorums consisting of a single process. Neither algorithm is strictly better than the other: There are situations in which our algorithm allows the system to make progress while the algorithm in [13] does not, and vice versa.

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 (majority) 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. 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.

3 The Model

We assume the existence of a *core* set of processes, $\mathcal{W}_0$, that is fixed and known to all the processes in $\mathcal{W}_0$. The set of all the processes that may run the protocol, is unknown to any of the processes in advance. 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 underlying asynchronous communication network. The system model allows for the following communication network changes: 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 [2]. Failures are detected using a (possibly unreliable) membership mechanism, as described in Section 3.1. Messages are not corrupted or spontaneously created. While no failures occur, processes communicate via reliable FIFO channels: Messages can be delayed arbitrarily, but are not lost[3].

3.1 Membership and Failure Detection

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. Dynamic voting protocols solve the latter problem, assuming a separate mechanism that solves the former. We assume a membership mechanism no stronger than those assumed in [8, 12, 18, 14, 9, 1].

Each process is equipped with an underlying membership module, *e.g.* [2, 3, 10]. When this 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. Membership changes are reported to the process via special *membership messages*. The membership reports do not necessarily reflect the network situation, nor is the membership reported atomically to all the processes. We only require that if a set of processes, $\mathcal{G}$, is connected and no subsequent failures occur for sufficiently long, then eventually, all the members of $\mathcal{G}$ receive the same membership message indicating that $\mathcal{G}$ is the current membership, and do not receive other membership messages while they are connected.

As shown in [6], it is impossible to reach agreement upon the current membership in an asynchronous system without failure detection. Consequently, agreement on the primary component in the network is also impossible. Therefore, any protocol for this problem must assume some (not necessarily perfect) failure detection mechanism. In our protocol, the failure detector is encapsulated in the membership module. The protocol we present is *correct* regardless of whether the membership mechanism is 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.

The membership module preserves causal dependencies between membership messages and the rest of the messages: If a process p sends a message m after receiving a membership message m', and m' is the last membership message that p received before sending m, then every process that receives m, receives m' before it. Note that this requirement is easily fulfilled, and is provided by typical membership protocols.

4 The Primary Component Protocol

We describe 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

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

[3] We assume that the underlying communication discovers the loss and either recovers it, or reports of a failure.

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.

The protocol we present resembles Three Phase Commit (3PC) protocols [20, 21, 15]. 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. In this 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, and 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 other members. If it is not an eligible quorum, the session is aborted. In the last step, the processes *form* the new quorum. If a process receives a membership message in the course of a session, it aborts the session and invokes a new session.

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

4.1 Dynamic Quorums

Originally, dynamic voting was implemented by always allowing a majority of the previous primary component to become the new primary component. *Dynamic linear voting*, first presented in [12], optimizes the above with the following mechanism to break ties between groups of equal size: A linear order, $\mathcal{L}$, is imposed on all the potential processes in the system, *e.g.* $\mathcal{L}$ can be the lexicographical order over an infinite name space. In case a quorum Q divides into two subsets of equal size, the subset containing the process with the highest rank in Q is chosen.

In order to avoid situations where almost all of the processes are connected but cannot form a quorum, because of the potential existence of a past surviving quorum at processes that are down, we provide the ability to impose a minimum quorum size on quorums allowed in the system. The minimum quorum size restriction, *Min_Quorum*, implies that every eligible quorum in the system must contain at least *Min_Quorum* members of the core group, $\mathcal{W}_0$. With this restriction, every quorum containing a subset of W_0 of size bigger than $n - Min\_Quorum$ is an eligible quorum, where n is the size of $\mathcal{W}_0$.

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. $|T \cap W_0| \geq Min\_Quorum$, 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
   - $|T \cap W_0| > n - Min\_Quorum$.

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 in the protocol we present here, the members of the core group, W_0, have a special status: every quorum in the system must contain at least *Min_Quorum* members of $\mathcal{W}_0$. This requirement restricts the availability if some members of $\mathcal{W}_0$ leave the system. In Section 6 we show how to relax this restriction, and require, instead, that a quorum will contain *Min_Quorum* processes.

4.2 Variables

Each process p maintains the following variables:

$Is\_Primary_p$ A boolean variable that is TRUE iff the current membership is the primary component in the system. If $p \in W_0$, then it is initialized to TRUE, and otherwise to FALSE.

$Session\_Number_p$ The current session number. This variable is initialized to 0, and is updated in the Attempt Step (Step 2) of the protocol.

$Last\_Primary_p$ The last primary component that process p formed (*i.e.* the last session that p ended successfully). It is a structure consisting of:

1. *Last_Primary*.M The membership of the session in which this primary component was formed. If $p \in W_0$, then it is initialized to $\mathcal{W}_0$, and otherwise to ∞. We extend the definition of the *Sub_Quorum* predicate so that $Sub\_Quorum(\infty, T)$ is FALSE for every set T.
2. *Last_Primary*.N The *Session_Number* of the session in which this primary component was formed. If $p \in W_0$, then it is initialized to 0, and otherwise to -1.

The structure is updated in Step 3 of the protocol.

$Ambiguous\_Sessions_p$ The set of (ambiguous) sessions process p attempted to form after p formed $Last\_Primary_p$. Each ambiguous session (or attempt) S in this set is a structure consisting of:

1. $S.M$ The membership of S.
2. $S.N$ The *Session_Number* of S.

This set is initially empty.

4.3 Notation

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. We denote by $i_{\mathcal{M}}(q)$ the index of process q in this list.
- *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 $\bigcup_{p \in \mathcal{M}}(A \in Ambiguous\_Sessions_p | A.N > Max\_Primary.N)$.

In order to simplify notations, we extend the definition of the *Sub_Quorum* predicate to attempts. For a pair of attempts $A1, A2$, $Sub\_Quorum(A1, A2)$ is defined to be: $Sub\_Quorum(A1.M, A2.M)$.

4.4 The Protocol

In each session, each process invokes the protocol described in Figure 1 upon receiving a membership message.

A session S of the protocol is identified by its membership, $S.M$, and session number, $S.N$. A *formed session* is a session that at least one of its members has formed. An *attempted session (attempt)* is a session that at least one of its members has attempted to form (*i.e.* recorded the session during the Attempt Step). Note that every formed session is in particular attempted. The initial primary component $(\mathcal{W}_0, 0)$ is considered both a formed and an attempted session. In Section 4.8 we claim that every formed session is uniquely identified by its session number.

1. Set *Is_Primary* to FALSE.
   Send your *Session_Number*, *Last_Primary*, and *Ambiguous_Sessions* to all the members of $\mathcal{M}$.
2. *Attempt Step:* Upon receiving such a message from all members of $\mathcal{M}$:
   - Compute *Max_Session*, *Max_Primary*, and *Max_Ambiguous_Sessions*.
   - **if** $(Sub\_Quorum(Max\_Primary.M, \mathcal{M})$ and
     $((\forall Attempt \in Max\_Ambiguous\_Sessions)\ \ Sub\_Quorum(Attempt.M, \mathcal{M})\ ))$
     then "attempt the session":
     - Set *Session_Number* to $Max\_Session + 1$.
     - Add $(\mathcal{M}, Session\_Number)$ to *Ambiguous_Sessions*.
       If *Ambiguous_Sessions* already contains an attempt with the same membership, overwrite it.
     - Send an attempt message to every member of $\mathcal{M}$.

     else abort this session.
3. *Form Step:* Upon receiving an attempt message from all members of $\mathcal{M}$ set:
   - *Last_Primary* to $(\mathcal{M}, Session\_Number)$, and
   - *Ambiguous_Sessions* to $\emptyset$, and
   - *Is_Primary* to TRUE.

Figure 1: A Session of the Protocol

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[4]. 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. If a process attempts to form two sessions with identical memberships it stores only the one with the greater session number, *i.e.* the later attempt.

The protocol presented here is symmetric; Every process performs exactly the same set of commands. It is straightforward to convert it to work in a centralized fashion by appointing a coordinator for each session. In every step the coordinator receives messages from all processes in a session, does local computation, and sends every process its decision. The centralized version requires less point to point messages. However, with hardware multicast capabilities, the symmetric version is more efficient.

4.5 The Typical Problematic Scenario

We now show how the suggested protocol overcomes the problematic scenario described in Section 1.

- The systems consists of five processes a, b, c, 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 sub-quorum of $\{a, b, c\}$.
- Concurrently c connects with d and e. $\{c, d, e\}$ is not a sub-quorum of $\{a, b, c\}$ that c records, therefore c, d, and e cannot form a new quorum.

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

4.6 Ruling Out a Trivial Approach

The scenario depicted above aroused the need to consider attempts to form a quorum, 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 following example demonstrates that this approach does not work:

- The core group $\mathcal{W}_0$ consists of processes a, b, c, d, e.
- For each process, *Last_Primary*$= (\mathcal{W}_0, 0)$, and $Ambiguous\_Sessions = \emptyset$.

[4] 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.

Session	a	b	c	d	e
S1=($\{a, b, c\}, 1$)	Form	Attempt	Attempt	-	-
S2=($\{b, c, d\}, 2$)	-	-	Attempt	Attempt	-
S3=($\{a, b\}, 2$)	Form	Form	-	-	-
S3'=($\{c, d, e\}, 3$)	-	-	Form	Form	Form

- In session $S1$, a forms $S1$. Processes b and c attempt to form $S1$ and detach before forming it.
- In session $S2$ (which is a sub-quorum of both $\mathcal{W}_0$ and $S1$), c and d attempt to form $S2$. b detaches before performing the attempt step. If now c records only the last attempt it made then c no longer takes $S1$ into account.
- Now it is possible to form concurrently two different primary components: Session $S3$, that consists of a and b, is a sub-quorum of $S1$ which is *Max_Primary* (hence a sub-quorum of $\mathcal{W}_0$ is not needed). Therefore it is a legal new quorum, and a and b form it successfully.

  Session $S3'$, that consists of c, d and e, is a sub-quorum of both $\mathcal{W}_0$ and $S2$. Therefore this session too is a legal new quorum, and c, d and e form it successfully.

Similarly, it is easy to show that recording only a constant number of the most recent attempts can lead to inconsistencies.

4.7 An Exponential Example

In this section we show that without any garbage collection mechanism, the number of attempts a process records concurrently can be exponential in the size of the initial configuration n.

- The core group $\mathcal{W}_0$ consists of processes $\{p_1, \ldots, p_n\}$.
- $Min\_Quorum < \lceil (n+1)/2 \rceil$.
- $\forall i$ $Last\_Primary_{p_i} = (\mathcal{W}_0, 0)$ and $Ambiguous\_Sessions_{p_i} = \emptyset$.

Let $\mathcal{G}$ be a subset of $\mathcal{W}_0$, such that $|\mathcal{G}| = \lceil (n+1)/2 \rceil$. The power set of $\mathcal{W}_0 \setminus \mathcal{G}$ consists of $2^{\lfloor n/2 \rfloor}$ groups: $G_1, \ldots, G_{2^{\lfloor n/2 \rfloor}}$. We now describe an execution comprised of sessions $S_1, S_2, \ldots, S_{2^{\lfloor n/2 \rfloor}}$, such that $\forall i$, $1 \leq i \leq 2^{\lfloor n/2 \rfloor}$ session S_i is conducted as follows:

- S_i's session number is i.
- S_i's membership is $\mathcal{G} \cup G_i$
- *Max_Primary* of S_i is $(\mathcal{W}_0, 0)$
- Conditions: $Sub\_Quorum(\mathcal{W}_0, S_i.M)$ holds, and $\forall j < i$ $Sub\_Quorum(S_j, S_i)$ holds too.
- Actions: p_1 appends S_i to $Ambiguous\_Sessions_{p_1}$. All other members of S_i detach before receiving p_1's attempt message, therefore do not append S_i to their *Ambiguous_Sessions* sets.

After this series of sessions process p_1 records $2^{\lfloor n/2 \rfloor}$ different sessions, which it has to consider in its future attempts to form a new quorum. Notice that the number of attempts a process records concurrently is at most 2^n, the size of the power set of $\mathcal{W}_0$, since every different membership appears only once in the process' *Ambiguous_Sessions* set.

4.8 Correctness

The full correctness proof of the protocol appears in Appendix A. In this section we describe the proof's outline. In order to show that the protocol is correct, we have to show that the transitive closure of the order between intersecting[5] formed sessions 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.

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, $Sub\_Quorum(\mathcal{F}_k, S)$ holds.

   Proved by induction on k, using the fact that all the members of a formed session have recorded it and take it into account.

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 $Sub\_Quorum(\mathcal{F}, S)$ holds.

   Proved by (sparse) induction on $\mathcal{F}.N$, using the above Claims.

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.

5 The Optimization

In this section we present an optimized version of the basic protocol, described in Section 4. We present a "garbage collection" mechanism that allows a process to reduce the number of ambiguous sessions it records concurrently from an exponential number to a linear number in the worst case.

The optimization (garbage collection) is achieved by local manipulations that each process performs on its *Ambiguous_Sessions* and *Last_Primary* data structures. The manipulations are performed in Step 2 of the protocol, according to the information exchanged in Step 1 of the protocol.

The underlying concept of the optimization is the resolution of ambiguous sessions: Each process tries to determine the status of each ambiguous session, *i.e.* whether the session was formed by one of its members or not. If an ambiguous session was not *formed* by any of its members, then

[5] By *intersecting sessions* we mean sessions with common members.

it is safe to delete it from *Ambiguous_Sessions*. In particular, if some process did not *attempt* to form the session, then it is safe to delete it. On the other hand, if the session was *formed* by some member, then the other members behave as if they also formed this session. The processes gather information that helps them resolve the status of ambiguous sessions.

The Resolution Rules:

- Process p sets $Last\_Primary_p$ to $\mathcal{F}$ during session S, where $S.N \geq \mathcal{F}.N$, when the following holds:

  1. $p \in \mathcal{F}.M$ and $Last\_Primary_p.N < \mathcal{F}.N$, and
  2. p learns that session $\mathcal{F}$ was formed by one of its members.

  If process p sets $Last\_Primary_p$ to $\mathcal{F}$ then for every process $q \in \mathcal{F}.M$ p sets $Last\_Formed_p(q)$ to $\mathcal{F}$.

- Process p deletes an ambiguous session S from $Ambiguous\_Sessions_p$ when one of the following holds:

  1. p learns that S was not formed by any of its members, or
  2. 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.

Figure 2: The Resolution Rules

The data structures used to gather information are described in Section 5.1 below, and the rules of how a process *learns* about the session status at other members are defined in Section 5.2. In Section 5.3 we show that at any given time, at each process, at most n ambiguous sessions remain unresolved. The resolution rules are specified in Figure 2. These rules are applied in Step 2 of the protocol described in Figure 3.

5.1 Data Structures

The optimized protocol maintains the following additional data structures in which it gathers information that help resolve past ambiguous sessions:

- $Ambiguous\_Sessions_p$, the set of ambiguous sessions process p attempted to form after p participated in $Last\_Primary_p$, is augmented with an array $S.A$ of size $S.M$, such that for every $q \in S.M$:

  - $S.A[i_{\mathcal{M}}(q)] = 1$ iff p knows that q formed S.
  - $S.A[i_{\mathcal{M}}(q)] = -1$ iff p knows that q did not form S.
  - $S.A[i_{\mathcal{M}}(q)] = 0$ otherwise.

  This array is updated in Step 2 of the optimized protocol, according to the learning rule described in the next section.

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

2. *Attempt step:* Upon receiving such a message from all members of $\mathcal{M}$:

   - Update *Ambiguous_Sessions* according to the learning rules.
   - Apply the resolution rules if possible.
   - 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 following ambiguous session structure, S:
       * $S.M = \mathcal{M}$,
       * $S.N = Session\_Number$,
       * $S.A[i_{\mathcal{M}}(q)] = 0$ for every $q \in S.M$ s.t. $q \neq p$, and $S.A[i_{\mathcal{M}}(p)] = -1$.
     - Send attempt message to every member of $\mathcal{M}$.

     else abort this session.

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 3: A Session of the Optimized Protocol Executed by Process p

- $Last\_Formed_p$ is an associative array. For each q that p participated in a session with, $Last\_Formed_p(q)$ is the last session that p formed and q was a member of. It is a structure consisting of:

  - The membership: $Last\_Formed_p(q).M$.
  - The session number: $Last\_Formed_p(q).N$.

  Initially, $Last\_Formed_p(q)$ contains F_0 for every process $q \in \mathcal{W}_0$. $Last\_Formed_p$ is updated in Steps 2 and 3 of the optimized protocol.

5.2 Gaining knowledge

In order to apply the optimization rules to an ambiguous session, a process p needs to *learn* whether the session 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 data structure in the second step of the optimized protocol. Formally, the rules of learning are the following:

Process p learns that process q established session S during a session S', where $S.N < S'.N$ and $p, q \in S.M \cap S'.M$, if during S':

- $Last\_Formed_q(p).N = S.N$, and
- p executes Step 2 of the optimized protocol.

Process p learns that process q did not form session S during a session S', where $S.N < S'.N$ and $p, q \in S.M \cap S'.M$, if during S':

- $Last\_Formed_q(p).N < S.N$, and
- p executes Step 2 of the optimized protocol.

A process p can deduce that a session S was not formed by any of its members in one of two ways: First, by learning as specified above from every member of S. Second, upon learning that another session member q does not consider S to be ambiguous although q did not form S or any other session with a session number greater than S; This can occur either because q did not even attempt to form S, or because q already learned that none of S members formed S. Formally:

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'
  - $Last\_Primary_q.N < S.N$ or ($Last\_Primary_q.N = S.N$ and $Last\_Primary_q.M \neq S.M$), and
  - $S \notin Ambiguous\_Sessions_q$, and
  - p executes Step 2 of the optimized protocol during S'.

5.3 Evaluating the Efficiency

In this section we evaluate the efficiency of the optimization: we show that a process records concurrently at most n ambiguous sessions, where n is the number of processes in the system. First, we prove that if a process p attempts to form[6] two ambiguous sessions with a process q, then during the later session p can learn q's status *w.r.t.* the former session. 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.

[6]Process p attempts to form a session S if p appends S to $Ambiguous\_Sessions_p$.

Lemma 1 *At each process p, the value of Session_Number is monotonously increasing.*

Proof: According to the protocol, *Session_Number* is chosen to be greater than the maximum of previous *Session_Number* each process records. $\Box$

Lemma 2 *Let p be a process and A_1, A_2 two ambiguous sessions, such that $A_1.N < A_2.N$ and $A_1, A_2 \in Ambiguous\_Sessions_p$. 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 *Last_Formed* lists from all session members and applied the learning rules before attempting to form A_2, as follows:

  1. If $Last\_Formed_q(p).N < A_1.N$ then p learned that q did not form A_1.
  2. If $Last\_Formed_q(p).N = A_1.N$ then p learned that q formed A_1.

  Notice that the case $Last\_Formed_q(p).N > A_1.N$ is impossible because otherwise, process q formed $Last\_Formed_q(p)$, and therefore, process p attempted to form it. This implies, according to Lemma 1, that $A_1.N < Last\_Formed_q(p).N < A_2.N$, 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 $A_2 \in Ambiguous\_Sessions_p$, p received $Last\_Formed_q(p)$ during session A_2, and learned as follows:

  1. If $Last\_Formed_q(p).N \leq A_1.N$ then, as in the base case, p learned whether q formed A_1.
  2. Otherwise, $Last\_Formed_q(p).N > A_1.N$. Hence, as above, 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 process p learned whether q formed A_1 before attempting to form F_i, and hence before attempting to form A_2. $\Box$

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, in contradiction, that p records concurrently $n - Min\_Quorum + 2$ ambiguous sessions, $A_1, \ldots, A_{n-Min\_Quorum+2}$, such that $(\forall 1 \leq i < n - Min\_Quorum + 2) A_i.N < A_{i+1}.N$. (By Lemma 1 the order requirement is always fulfilled). 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 $A_j \in \mathit{Ambiguous\_Sessions}_p$ 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 $A_1, \ldots, A_{n-\mathit{Min\_Quorum}+1}$, there are at least $n - \mathit{Min\_Quorum} + 1$ members that are not members of $A_{n-\mathit{Min\_Quorum}+2}$. Therefore $|A_{n-\mathit{Min\_Quorum}+2}.M| < \mathit{Min\_Quorum}$, and hence $A_{n-\mathit{Min\_Quorum}+2}$ is not a legal session and hence is not recorded by any process, a contradiction. □

6 Dynamically Changing Quorum Requirements

In the protocols presented so far, the members of the core group, W_0, have a special status: Every quorum in the system must contain at least *Min_Quorum* members of $\mathcal{W}_0$. This requirement restricts the availability if some members of $\mathcal{W}_0$ leave the system. In this section we show how to relax this restriction, and thus increase the flexibility to dynamically change the set of participants.

The purpose of the *Min_Quorum* requirement is to prevent a set of processes smaller than *Min_Quorum* from blocking the rest of the system, *i.e.* to always allow a group of more than $n - \mathit{Min\_Quorum}$ processes to make progress. In Section 4.1, n was fixed to be the size of the core group of processes, $\mathcal{W}_0$. We now show how to provide this feature while allowing n to increase dynamically, thus eliminating the special status of the members of W_0.

Allowing n to change dynamically is subtle because the truth value of the *Sub_Quorum* predicate changes with time. For example, $\mathit{Sub\_Quorum}(S, T)$ may be initially TRUE because T contains more than $|W_0| - \mathit{Min\_Quorum}$ members of W_0, but later, as the set of participants increases, $\mathit{Sub\_Quorum}(S, T)$ 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" as described below:

Every process maintains two new variables:

$\mathcal{W}$ 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; they can be incorporated in either the basic version of the protocol (cf. Section 4) or in the optimized one (cf. Section 5). At the beginning of each step in a session S of the protocol, every process p executes the following operations:

1. 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 S is an eligible quorum, regardless of past quorums, *i.e.* for every session S' that p records, $Sub\_Quorum(S', S)$ is TRUE.

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 Appendix A.

References

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

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

[3] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. Fast Message Ordering and Membership using a Logical Token-Passing Ring. In *International Conference on Distributed Computing Systems*, number 13, pages 551–560, May 1993.

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

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

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

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

[8] D. Davcev and W. Burkhard. Consistency and Recovery Control for Replicated Files. In *ACM Symp. on Operating Systems Principles*, number 10, pages 87–96, 1985.

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

[10] P. D. Ezhilchelvan, A. Macedo, and S. K. Shrivastava. Newtop: a Fault Tolerant Group Communication Protocol. In *International Conference on Distributed Computing Systems*, number 15, pages 296–306. IEEE Computer Society Press, June 1995.

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

[12] S. Jajodia. Managing Replicated Files in Partitioned Distributed Database Systems. In *IEEE Int'l. Conf. on Data Engineering*, number 3, pages 412–418, 1987.

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

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

[15] I. Keidar and D. Dolev. Increasing the Resilience of Atomic Commit, at No Additional Cost. In *ACM Symp. on Prin. of Database Systems (PODS)*, pages 245–254, May 1995. Previous version available as Technical Report CS94-18, The Hebrew University, Jerusalem, Isreal.

[16] I. Keidar and D. Dolev. Efficient Message Ordering in Dynamic Networks. In *ACM Symp. on Prin. of Distributed Computing (PODC)*, number 15, pages 68–76, May 1996.

[17] C. Malloth and A. Schiper. 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).

[18] J. Paris and D. Long. Efficient Dynamic Voting Algorithms. *Proceedings 13th Int'l. Conf. on Very Large Data Bases*, pages 268–275, 1988.

[19] A. M. Ricciardi and K. P. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *ACM Symp. on Prin. of Distributed Computing (PODC)*, pages 341–352, August 1991.

[20] D. Skeen. Nonblocking Commit Protocols. In *SIGMOD Intl. Conf. Management of Data*, pages 133–142, 1981.

[21] D. Skeen. A Quorum-Based Commit Protocol. In *Berkeley Workshop on Distributed Data Management and Computer Networks*, number 6, pages 69–80, Feb. 1982.

A Correctness Proof of the Protocols

The basic requirement from a dynamic paradigm for maintaining a primary component is to impose a total order on all the primary components formed in the system. Total order on primary components is defined by extending the causal order on components that intersect. This requirement is formally postulated in Section 2. We now prove the correctness of our protocols, *i.e.* that the total order requirement is fulfilled. The proof of the basic protocol is detailed in Section A.1. The proof of the optimized protocol closely resembles the proof of the basic protocol, hence only the differences are discussed in Section A.2. Section A.3 contains the correctness proof of the protocol that allows the quorum requirements to change dynamically.

A.1 Proof of the Basic Protocol

Notations and Definitions

- The variables: *Session_Number*, *Max_Session*, *Max_Ambiguous_Sessions*, and *Max_Primary* computed during a session S, are denoted: *Session_Number*(S), *Max_Session*(S), *Max_Ambiguous_Sessions*(S), and *Max_Primary*(S).
- $Quorums\_List(S)$ is defined as: $Max\_Ambiguous\_Sessions(S) \cup Max\_Primary(S)$.
- The initial primary component $(\mathcal{W}_0, 0)$ is denoted: F_0.
- In general, we shall denote a session by S, and a formed session by $\mathcal{F}$.

Lemmas

Lemma 3 *Whenever a process p modifies the value of* $Session\_Number_p$*, the value is incremented.*

Proof: Process p modifies the value of $Session\_Number_p$ at most once during a session S, where $p \in S.M$, in Step 2 of the protocol, and never modifies $Session\_Number_p$ outside a session. In addition, p is a member of at most one session at a time, since upon noticing a membership change p immediately ends the current session it was a member of, and joins the new session at Step 1. When p modifies the value of $Session\_Number_p$ it sets it to $Max\_Session(S) + 1$, and writes the result to stable storage. The value of $Max\_Session(S)$ is computed as $\max_{q \in S.M}(Session\_Number_q)$, hence $Max\_Session(S) \geq Session\_Number_p$. Consequently, whenever p modifies the value of $Session\_Number_p$ it increments it at least by 1. $\Box$

Lemma 4 *If two members* p, q *of a session* S *attempt to form* S*, then* p *and* q *increment their* *Session_Number during session* S*, and* $Session\_Number_p = Session\_Number_q$ *upon ending the session.*

Proof: A process p attempts to form session S in Step 2 of the protocol, only if p received $Session\_Number_r$ from all members r of S during Step 1 of the protocol. During Step 2 of session S p increments $Session\_Number_p$ to $Max\_Session(S)+1$. Since $Max\_Session(S)$ is computed over the same set of values at all sites q that also attempt to form S, $Session\_Number_p = Session\_Number_q$ upon ending the session. $\Box$

Lemma 5 *If a member p of a formed session $\mathcal{F}$ sets $Last\_Primary_p$ to $\mathcal{F}$ during session $\mathcal{F}$, then all members q of $\mathcal{F}$ appended $\mathcal{F}$ to $Ambiguous\_Sessions_q$ during this session.*

Proof: Process p sets $Last\_Primary_p$ to $\mathcal{F}$ during session $\mathcal{F}$ in Step 3 of the protocol, only if p received attempt messages from all members q of $\mathcal{F}$ indicating that they successfully executed Step 2 of the protocol during this session, *i.e.* appended $\mathcal{F}$ to $Ambiguous\_Sessions_q$. $\square$

Corollary 1 *During every formed session F every member p of F increments $Session\_Number_p$ to $F.N$.*

Proof: Immediate from Lemmas 4 and 5. $\square$

Lemma 6 *If two formed sessions, F_1, F_2, intersect then $F_1.N \neq F_2.N$.*

Proof: Let process $p \in (F_1.M \wedge F_2.M)$. By Corollary 1, p increments $Session\_Number_p$ in both F_1, F_2, *w.l.o.g.* p participates in F_1 first. Hence $F_1.N < F_2.N$. $\square$

Lemma 7 *Let A be an attempt, such that there is no formed session F fulfilling $F_0.N < F.N < A.N$. Then A and F_0 intersect, and $A.N > F_0.N$.*

Proof: For every process p in the system, $Session\_Number_p$ is initialized to zero. By Lemma 3 $A.N > 0$, *i.e.* $A.N > F_0.N$.

Since A is an attempt, there exists a formed session F s.t. $F \in Quorums\_List(A)$, and $Sub\_Quorum(F, A)$ is TRUE. If $F = F_0$ then, by the Sub_Quorum requirements, A and F_0 intersect. Otherwise, $F \neq F_0$: Since F is in particular an attempt, $F.N > F_0.N$. Moreover, there exists a process p s.t. $p \in F.M \cap A.M$. By Corollary 1 and Lemma 3, in Step 1 of session A $Session\_Number_p \geq F.N$. Thus $Session\_Number(A) > F.N > F_0.N$, in contradiction to the definition of A. Hence $F = F_0$ and A and F_0 intersect. $\square$

Lemma 8 *If $Last\_Primary_p = \mathcal{F}$ then for every member q of $\mathcal{F}$ either*

- *$\exists A \in Ambiguous\_Sessions_q$, such that $A.M = \mathcal{F}.M$ and $A.N \geq \mathcal{F}.N$, OR*
- *$Last\_Primary_q = \mathcal{F}$, OR*
- *$Last\_Primary_q.N > \mathcal{F}.N$.*

Proof: By Lemma 5 all members q of $\mathcal{F}$ appended $\mathcal{F}$ to $Ambiguous\_Sessions_q$. Process q erases $\mathcal{F}$ from $Ambiguous\_Sessions_q$ in two cases:

- When q forms a session $\mathcal{F}'$ after q appended $\mathcal{F}$ to $Ambiguous\_Sessions_q$. If $\mathcal{F}' = \mathcal{F}$, then $Last\_Primary_q = \mathcal{F}$. Otherwise, q participated in $\mathcal{F}'$ after it ended session $\mathcal{F}$. By Corollary 1, $Session\_Number_q$ is incremented during session $\mathcal{F}'$, hence $\mathcal{F}'.N > \mathcal{F}.N$.

- When q appends to $Ambiguous\_Sessions_q$ a session A, such that $A.M = \mathcal{F}.M$ and $A.N > \mathcal{F}.N$.

Process q erases $\mathcal{F}$ from $Last\_Primary_q$ in case q forms at least one session, $\mathcal{F}'$, after forming $\mathcal{F}$. By Corollary 1 $\mathcal{F}'.N > \mathcal{F}.N$. $\square$

Lemma 9 *For every sequence of sessions $\mathcal{F}_1, \ldots, \mathcal{F}_k$ and a formed session S, such that*

- $\forall i \;\; \mathcal{F}_i$ *is a formed session, and*
- $\forall i < k \;\; \mathcal{F}_i.N < \mathcal{F}_{i+1}.N$, *and*
- $\forall i < k \;\; Sub\_Quorum(\mathcal{F}_i, \mathcal{F}_{i+1})$ *is* TRUE, *and*
- $S.N > \mathcal{F}_k.N$

if $\mathcal{F}_1 \in Quorums\_List(S)$ *then* $Sub\_Quorum(\mathcal{F}_k, S)$ *is* TRUE.

Proof : By induction on k.

- **Base case,** $k = 1$ Since S is a formed session and $\mathcal{F}_1 \in Quorums\_List(S)$ then, according to the protocol, $Sub\_Quorum(\mathcal{F}_1, S)$ is TRUE.
- **General case,** $k > 1$ By the induction hypothesis, $Sub\_Quorum(\mathcal{F}_{k-1}, S)$ is TRUE. By the sequence definition, $Sub\_Quorum(\mathcal{F}_{k-1}, \mathcal{F}_k)$ is also TRUE. Therefore, by the properties of Sub_Quorum, $\mathcal{F}_k.M \wedge S.M \neq \emptyset$.

  Let $p \in \mathcal{F}_k.M \wedge S.M$. Since by our assumption, $S.N > \mathcal{F}_k.N$, and S is a formed session, p participated in $\mathcal{F}_k.N$ before participating in $S.N$. By Lemma 8, when S begins, there are two possibilities:

  1. p has $Last\_Primary_p.N \geq \mathcal{F}_k.N$. This case is not possible since $\mathcal{F}_1 \in Quorums\_List(S)$, therefore when session S begins, p has $Last\_Primary_p.N \leq \mathcal{F}_1.N < \mathcal{F}_k.N$.
  2. $\exists \mathcal{F}' \in Ambiguous\_Sessions_p$ such that $\mathcal{F}'.M = \mathcal{F}_k.M$ and $\mathcal{F}'.N \geq \mathcal{F}_k.N$. This implies that $Sub\_Quorum(\mathcal{F}', S)$ is TRUE, and therefore $Sub\_Quorum(\mathcal{F}_k, S)$ is also TRUE.

$\square$

Lemma 10 *Let A be an attempt other than F_0. Let $\mathcal{F}$ be a formed session such that $\mathcal{F}.N = max(F.N | F$ is a formed session and $F.N < A.N)$. Then the value of $\mathcal{F}.N$ is unique among formed sessions, and $Sub\_Quorum(\mathcal{F}, A)$ is* TRUE.

Proof: The proof is by induction on $\mathcal{F}.N$.

- **Base case** $\mathcal{F}.N = 0$. By Lemma 3, and since for every process p $Session\_Number_p$ is initialized to zero, $\mathcal{F} = F_0$. By Lemma 7, the value of $F_0.N$ is unique among attempts, and A and F_0 intersect.

It remains to show that $Sub\_Quorum(F_0, A)$ is TRUE. $\forall p \in \mathcal{W}_0$ the initial value of $Last\_Primary_p$ is F_0. By Lemma 8 either $F_0 \in Quorums\_List(A)$ and hence $Sub\_Quorum(F_0, A)$ is TRUE, or else there exist a process p and a formed session $\mathcal{F}'$ such that: $p \in \mathcal{F}'.M \cap A.M$, and during A $Last\_Primary_p = \mathcal{F}'$, and $\mathcal{F}'.N > F_0.N$. By Lemma 3 $A.N > \mathcal{F}'.N$, in contradiction to $F_0.N$ being maximal. Therefore, $\mathcal{F}'$ does not exist and hence $Sub\_Quorum(F_0, A)$ is TRUE.

- **General case** $\mathcal{F}.N > 0$

  Let F^* be a formed session such that $\mathcal{F}^*.N = max(F.N | F$ is a formed session and $F.N < \mathcal{F}.N)$. Then, by the induction hypothesis, the value of $\mathcal{F}^*.N$ is unique among formed sessions, and $Sub\_Quorum(\mathcal{F}^*, \mathcal{F})$ is TRUE. Assume, for the sake of contradiction, that $\mathcal{F}.N$ is not unique among formed sessions, then there exists a formed session F' such that $F'.N = \mathcal{F}.N$. By the induction hypothesis $Sub\_Quorum(F^*, F')$ is also TRUE. By the properties of Sub_Quorum, $\mathcal{F}$ and F' intersect, hence by Lemma 6 $F'.N \neq \mathcal{F}.N$, a contradiction. Therefore $\mathcal{F}.N$ is unique among formed sessions.

  Let $Max\_Primary(A) = \mathcal{F}'$, if $\mathcal{F}' = \mathcal{F}$, then $Sub\_Quorum(\mathcal{F}, A)$ is TRUE and we are done. Otherwise, by the definition of $\mathcal{F}$ and the uniqueness of $\mathcal{F}.N$ among formed sessions, $\mathcal{F}'.N < \mathcal{F}.N$. By the induction hypothesis there exists a unique sequence of sessions, $\mathcal{F}_1, \ldots, \mathcal{F}_k$ such that:

  - $\mathcal{F}_1 = \mathcal{F}'$, and
  - $\mathcal{F}_k = \mathcal{F}$, and
  - $\forall i \;\; \mathcal{F}_i$ is a formed session, and
  - $\forall i < k \;\; \mathcal{F}_i.N < \mathcal{F}_{i+1}.N$, and
  - $\forall i < k \;\; Sub\_Quorum(\mathcal{F}_i, \mathcal{F}_{i+1})$ is TRUE, and
  - $A.N > \mathcal{F}_k.N$

  Furthermore, $\mathcal{F}' \in Quorums\_List(A)$, hence by Lemma 9 $Sub\_Quorum(\mathcal{F}, A)$ is TRUE.

□

Theorem 2 *The transitive closure of the causal order between intersecting formed sessions, denoted $\prec$, is a total order.*

Proof: By Lemma 10, each formed session has a unique session number, hence we can define a total order on formed sessions as follows: $F < F'$ if $F.N < F'.N$. We now show that $F < F'$ iff $F \prec F'$, implying that $\prec$ is a total on formed sessions. If $F \prec F'$, then by Lemma 3 $F.N < F'.N$. If $F.N < F'.N$, then by Lemma 10, there exists a unique sequence of sessions, $\mathcal{F}_1, \ldots, \mathcal{F}_k$ such that:

- $\mathcal{F}_1 = \mathcal{F}$, and
- $\mathcal{F}_k = \mathcal{F}'$, and
- $\forall i \;\; \mathcal{F}_i$ is a formed session, and

- $\forall i < k \;\; \mathcal{F}_i.N < \mathcal{F}_{i+1}.N$, and
- $\forall i < k \;\; Sub\_Quorum(\mathcal{F}_i, \mathcal{F}_{i+1})$ is TRUE, and therefore, by the properties of *Sub_Quorum*, $\mathcal{F}_i \cap \mathcal{F}_{i+1} \neq \emptyset$.

Therefore, $F \prec F'$. □

A.2 Proof of the Optimized Protocol

The basic protocol and the optimized protocol closely resemble one another. There are two main differences between the two:

1. In the optimized protocol, a process can delete an ambiguous session upon discovering that none of the members formed the session. Notice that in the correctness proof of the basic protocol, we only reason about formed sessions. Sessions that are not formed by any member are not mentioned, and therefore do not affect the correctness.

2. In the optimized protocol a process can form session $\mathcal{F}$ during a later session S. There are two issues to consider:

   - In the proof of the basic protocol, the proof of the properties of formed sessions relies on the fact that a formed session $\mathcal{F}$ was formed by one of its members during $\mathcal{F}$. In Lemma 11 below, we show that if some process forms $\mathcal{F}$ in a later session S, then there exists at least one member of $\mathcal{F}$ that formed $\mathcal{F}$ during $\mathcal{F}$. Hence the lemmas regarding formed sessions remain correct for the optimized protocol.
   - When a process p forms $\mathcal{F}$, p deletes from its *Quorums_List* all the sessions A s.t. $A.N < \mathcal{F}.N$. Since p is a member of each such A and also of $\mathcal{F}$, p participated in the attempt to form F *after* participating in every such session A. Since some member of $\mathcal{F}$ formed $\mathcal{F}$ during $\mathcal{F}$, the conditions for forming $\mathcal{F}$ were fulfilled during session $\mathcal{F}$ (*i.e.* all the members have sent attempt messages), and therefore it is safe for p to delete these sessions (as would have happened had p formed $\mathcal{F}$ during $\mathcal{F}$).

Due to the above, the proof of the basic protocol is also correct for the optimized protocol. It remains to prove the following:

Lemma 11 *Process p sets $Last\_Primary_p$ to a session $\mathcal{F}$ during a session S, if either $\mathcal{F} = S$, or there exists a process q such that q set $Last\_Primary_q$ to $\mathcal{F}$ during $\mathcal{F}$.*

Proof: According to the protocol, a process p sets $Last\_Primary_p$ to $\mathcal{F}$ in one of two cases:

1. In Step 3 of the protocol.
2. In Step 2 of the protocol, upon learning that a process r set $Last\_Formed_r(p)$ to $\mathcal{F}$.

In the first case p forms $\mathcal{F}$ during $\mathcal{F}$. Henceforth we assume p set $Last\_Primary_p$ to $\mathcal{F}$ during Step 2 of the protocol in session S. Note it implies that during Step 1 of session S $Ambiguous\_Sessions_p$ contains a session A such that $A.M = \mathcal{F}.M$ and $A.N = \mathcal{F}.N$, *i.e.* p attempted to form session $\mathcal{F}$.

According to the protocol, a process q changes $Last\_Formed_q(r)$ to F upon setting $Last\_Primary_q$ to a session F such that $q, r \in \mathcal{F}.M$. To the contrary, assume that none of $\mathcal{F}$ members formed $\mathcal{F}$ during $\mathcal{F}$. Therefore, at the end of session $\mathcal{F}$, $\forall q, r \in \mathcal{F}.M$ $Last\_Primary_q \neq \mathcal{F}$, and $Last\_Formed_q(r) \neq \mathcal{F}$. Hence no process $p \in \mathcal{F}.M$ can learn that a process $r \in \mathcal{F}.M$ set its $Last\_Formed_r(p)$ to $\mathcal{F}$ during in Step 2 of a session, unless there exists a member q of $\mathcal{F}$ such that:

- q formed a session $\mathcal{F}'$ during session $\mathcal{F}'$, in which q participated after it ended session $\mathcal{F}$, and
- $\mathcal{F}'.M = \mathcal{F}.M$ and $\mathcal{F}'.N = \mathcal{F}.N$.

Since $\mathcal{F}'$ is a formed session, from Lemma 5 p attempted to form $\mathcal{F}'$. Since p attempted to form $\mathcal{F}$ too, then from Lemma 3 and Lemma 4 $\mathcal{F}'.N > \mathcal{F}.N$, in contradiction. Hence $\mathcal{F}$ was formed by one of its members during session $\mathcal{F}$. □

A.3 Proving the Correctness of the Dynamically Changing Quorum System

Lemma 12 *At each process p, $\mathcal{W}_p$ and $\mathcal{W}_p \cup \mathcal{A}_p$ are monotonically increasing.*

Proof: According to the protocol, processes are never removed from $\mathcal{W}_p$. Processes are removed from $\mathcal{A}_p$ only after they were added to $\mathcal{W}_p$. □

Lemma 13 *All the members that attempt to form a session S, set their $\mathcal{W}$ and $\mathcal{A}$ variables to the same value in the Attempt Step.*

Proof: Processes attempt to form a session after receiving information messages from all other members of the session. They compute $\mathcal{W}$ and $\mathcal{A}$ using only the information in these messages. All the members receive the same set of messages, and therefore the result of the computation is the same. □

Henceforth we denote by $\mathcal{W}(S)$ and $\mathcal{A}(S)$ the values of $\mathcal{W}$ and $\mathcal{A}$ computed in the Attempt Step during session S. We denote by $\mathcal{WA}(S)$ the union $\mathcal{W}(S) \cup \mathcal{A}(S)$.

Lemma 14 *If a process p forms a session S, then at the end of the session $\mathcal{W}_p \cup \mathcal{A}_p = \mathcal{WA}(S)$.*

Proof: Upon forming a session, process p may only move members from $\mathcal{A}_p$ to $\mathcal{W}_p$, and this is the only change made to these variables after p attempted the session. Hence, the union remains the same, and, by Lemma 13, is equal to $\mathcal{WA}(S)$. □

Lemma 15 *If two formed sessions $\mathcal{F}_i, \mathcal{F}_j$ intersect and $\mathcal{F}_i.N < \mathcal{F}_j.N$, then $\mathcal{WA}(\mathcal{F}_i) \subseteq \mathcal{WA}(\mathcal{F}_j)$.*

Proof: There exists a process p s.t. $p \in \mathcal{F}_i.M \cap \mathcal{F}_j.M$. From Lemmas 13 and 14, at the end of session $\mathcal{F}_i$, $\mathcal{W}_p \cup \mathcal{A}_p = \mathcal{WA}(\mathcal{F}_i)$, while at the end of session $\mathcal{F}_j$, $\mathcal{W}_p \cup \mathcal{A}_p = \mathcal{WA}(\mathcal{F}_j)$. From Lemma 12, $\mathcal{WA}(\mathcal{F}_i) \subseteq \mathcal{WA}(\mathcal{F}_j)$. □

Lemma 16 *If for two formed sessions $\mathcal{F}_i, \mathcal{F}_j$, $F_i \in Quorums\_List(F_j)$, then $\mathcal{F}_i$ and $\mathcal{F}_j$ intersect.*

Proof: Since $\mathcal{F}_i$ in *Quorums_List*(F_j), there is a member p of $\mathcal{F}_j$ s.t. during $\mathcal{F}_j$, $\mathcal{F}_i$ is in $Last\_Primary_p \cup Ambiguous\_Sessions_p$. Since a process records only sessions in which it participated, process p participated in session $\mathcal{F}_i$, implying $\mathcal{F}_i$ and $\mathcal{F}_j$ intersect.□

Lemma 17 *For every process p and for every process $q \in \mathcal{W}_p$, q has participated in some formed session $\mathcal{F}$.*

Proof: For every process p, $\mathcal{W}_p$ is initialized to $\mathcal{W}_0 = F_0.M$. $\mathcal{W}_p$ incorporates a process $q \notin \mathcal{W}_0$ during a session S in one of two cases:

- During the Attempt Step, if there exists a member r of S such that $q \in \mathcal{W}_r$.
- During the Form Step, if $q \in S.M$.

Hence if q is not a member of any formed session yet, then there is no process p such that $q \in \mathcal{W}_p$.
□

Definition 1 **Formed Sequence** *is a set of formed sessions $\{\mathcal{F}_1, \ldots, \mathcal{F}_k\}$, such that*

- $\mathcal{F}_1 = F_0$*, and*
- $\forall i < k \ \ \mathcal{F}_i.N < \mathcal{F}_{i+1}.N$*, and*
- $\forall i < k \ \ \mathcal{F}_i \in$ *Quorums_List*$(\mathcal{F}_{i+1})$*, and*
- *for every formed session $\mathcal{F}'$, such that $\mathcal{F}'.N < \mathcal{F}_k.N$, $\mathcal{F}'$ is in the formed sequence.*

$\mathcal{F}_i$ denotes the i'th session in the sequence.

Lemma 18 *Let a formed sequence, denoted $\mathcal{FS}$, be of length k. Then for every formed session $\mathcal{F}_i \in \mathcal{FS}$, where $i < k$, the following is* TRUE*:*

- $\mathcal{WA}(\mathcal{F}_i) \subseteq \mathcal{WA}(\mathcal{F}_{i+1})$*, and*
- $\mathcal{W}(\mathcal{F}_{i+1}) \subseteq \mathcal{WA}(\mathcal{F}_i)$*.*

Proof: By the definition of $\mathcal{FS}$, $\mathcal{F}_i \in$ *Quorums_List*$(\mathcal{F}_{i+1})$. By Lemma 16 $\mathcal{F}_i$ and $\mathcal{F}_{i+1}$ intersect, and by Lemma 15 $\mathcal{WA}(\mathcal{F}_i) \subseteq \mathcal{WA}(\mathcal{F}_{i+1})$.

In addition, every formed session in the system that is formed before session $\mathcal{F}_i$ is attempted, denoted $\mathcal{F}_j$, fulfills: $\mathcal{F}_j \in \mathcal{FS}$ and $j < i$. Hence, by Lemma 17, $\mathcal{W}(\mathcal{F}_{i+1}) \subseteq \bigcup_{\mathcal{F}_j \in \mathcal{FS}, j \leq i} \mathcal{F}_j.M$. Since for every formed session $\mathcal{F}$, $\mathcal{F}.M \subseteq \mathcal{W}(\mathcal{F}) \cup \mathcal{A}(\mathcal{F})$, then by the first part of this lemma, $\mathcal{W}(\mathcal{F}_{i+1}) \subseteq \mathcal{WA}(\mathcal{F}_i)$.□

Lemma 19 *Let S be a formed session. Let a formed sequence, $\mathcal{FS}$, fulfill the following* w.r.t. *S:*

- *$\mathcal{FS}$ contains every formed session $\mathcal{F}'$, where $\mathcal{F}'.N < S.N$, and*
- *$\mathcal{FS}$ is of length k, and*

- $\mathcal{F}_k.N < S.N$.

Then $\mathcal{F}_k \in$ *Quorums_List*(S).

Proof : By induction on k.

- **Base case,** $k = 1$ From the definition of S and $\mathcal{FS}$, $\mathcal{F}_1 \in$ *Quorums_List*(S).
- **General case,** $k > 1$

  By Lemma 17 and the definition of $\mathcal{FS}$, $\mathcal{W}(S) \subseteq \bigcup_{\mathcal{F}_j \in \mathcal{FS}} \mathcal{F}_j.M$. Let $\mathcal{F}_i$ be a formed session with the minimal index in $\mathcal{FS}$, s.t. $S.M \cap \mathcal{W}(S) \subseteq \mathcal{F}_i.M$.

  The minimality of $\mathcal{F}_i$ implies that there exists a process $p \in S.M \cap \mathcal{F}_i.M$. Since $\mathcal{F}_i$ is a formed session and $\mathcal{F}_i.N < S.M$, by Lemma 8 during S there exists a formed session F in $Ambiguous\_Sessions_p \cup Last\_Primary_p$ s.t. $F.N \geq \mathcal{F}_i.N$. Since $Session\_Number_p$ is monotonically increasing, $F.N < S.N$, and hence $F \in \mathcal{FS}$. All this implies that there exists a formed session $\mathcal{F}_j \in Quorums\_List(S) \cap \mathcal{FS}$ s.t. $j \geq i$. If $j = k$ then $\mathcal{F}_k \in Quorums\_List(S)$, and we are done. Otherwise, $j < k$.

  Since $\mathcal{F}_j \in Quorums\_List(S)$, then $Sub\_Quorum(\mathcal{F}_j, S)$ is TRUE. From the definition of $\mathcal{FS}$, $Sub\_Quorum(\mathcal{F}_j, \mathcal{F}_{j+1})$ is also TRUE during $\mathcal{F}_{j+1}$. From Lemma 18, $\mathcal{W}(\mathcal{F}_{j+1}) \subseteq \mathcal{WA}(\mathcal{F}_j)$, and $\mathcal{WA}(\mathcal{F}_j) \subseteq \mathcal{WA}(\mathcal{F}_{j+1})$. From Lemma 15 $\mathcal{WA}(\mathcal{F}_j) \subseteq \mathcal{WA}(S)$. There are two cases:

  - Both $\mathcal{F}_{j+1}$ and S contain a majority of $\mathcal{F}_j$. Hence, $\mathcal{F}_{j+1}$ and S intersect.
  - One of $\mathcal{F}_{j+1}, S$ does not contain a majority of $\mathcal{F}_j$, *w.l.o.g.* it is S. Both $\mathcal{F}_{j+1}$ and S contain at least *Min_Quorum* members of $\mathcal{WA}(\mathcal{F}_j)$. Moreover, $|S.M \cap \mathcal{WA}(\mathcal{F}_j)| > |\mathcal{WA}(\mathcal{F}_j)| - Min\_Quorum$. Thus $|S.M \cap \mathcal{WA}(\mathcal{F}_j)| + |\mathcal{F}_{j+1} \cap \mathcal{WA}(\mathcal{F}_j)| > |\mathcal{WA}(\mathcal{F}_j)|$. Hence, $\mathcal{F}_{j+1}$ and S intersect.

  In both cases, $\mathcal{F}_{j+1}$ and S intersect. Let process q be in $\mathcal{F}_{j+1}.M \cap S.M$. By Lemma 8, during S one of the following cases exists:

  - $\mathcal{F}_{j+1} \in Ambiguous\_Sessions_q \cup Last\_Primary_q$, and therefore $\mathcal{F}_{j+1} \in Quorums\_List(S)$.
  - $Last\_Primary_q = F'$ where $\mathcal{F}'.N > \mathcal{F}_{j+1}.N$. This is impossible since $\mathcal{F}_j \in Quorums\_List(S)$, and therefore when S begins $Last\_Primary_q.N \leq \mathcal{F}_j.N < F'.N$, a contradiction.

  If $F_{j+1} = \mathcal{F}_k$ then we are done. Otherwise, the above arguments can be iterated by replacing $\mathcal{F}_j$ with $\mathcal{F}_{j+1}$. Since in every iteration we prove that $\mathcal{F}_j \in Quorums\_List(S) \cap \mathcal{FS}(S)$ implies that $\mathcal{F}_{j+1} \in Quorums\_List(S) \cap \mathcal{FS}(S)$, and since $\mathcal{FS}(S)$ is finite, we are bound to reach $\mathcal{F}_{j+1} = \mathcal{F}_k$, yielding $\mathcal{F}_k \in Quorums\_List(S)$.

□

Lemma 20 *Let A be an attempt other than F_0. Let $\mathcal{F}$ be a formed session such that $\mathcal{F}.N = max(F.N | F$ is a formed session and $F.N < A.N)$. Then the value of $\mathcal{F}.N$ is unique among formed sessions, and $\mathcal{F} \in Quorums\_List(A)$.*

Proof: The proof is by induction on $\mathcal{F}.N$.

- **Base case** $\mathcal{F}.N = 0$. The proof is identical to the proof of the base case in Lemma 10, specified in Section A.1.

- **General case** $\mathcal{F}.N > 0$

  Let F^* be a formed session such that $\mathcal{F}^*.N = max(F.N|F$ is a formed session and $F.N < \mathcal{F}.N)$. Then, by the induction hypothesis, the value of $\mathcal{F}^*.N$ is unique among formed sessions, and $\mathcal{F}^* \in$ *Quorums_List*$(\mathcal{F})$. Moreover, there exists a formed sequence of length k, denoted $\mathcal{FS}_\mathcal{F}$, s.t. $\mathcal{F}_k = \mathcal{F}$.

  Assume, for the sake of contradiction, that $\mathcal{F}.N$ is not unique among formed sessions, then there exists a formed session $\mathcal{F}'$ such that $\mathcal{F}'.N = \mathcal{F}.N$. This implies that $\mathcal{F}^* \in$ *Quorums_List*$(\mathcal{F}')$, and that there exists a formed sequence of length k, denoted $\mathcal{FS}_{\mathcal{F}'}$, s.t. $\mathcal{F}_k = \mathcal{F}'$.

  Since both $\mathcal{F}, \mathcal{F}'$ are formed sessions, according to the protocol, *Sub_Quorum*$(\mathcal{F}^*, \mathcal{F})$ and *Sub_Quorum*$(\mathcal{F}^*, \mathcal{F}')$ are TRUE. By Lemma 18, $\mathcal{W}(\mathcal{F}) \cup \mathcal{W}(\mathcal{F}') \subseteq \mathcal{WA}(\mathcal{F}^*)$, and $\mathcal{WA}(\mathcal{F}^*) \subseteq \mathcal{WA}(\mathcal{F}) \cap \mathcal{WA}(\mathcal{F}')$. Now there are two possibilities:

  1. Both $\mathcal{F}, \mathcal{F}'$ contain a majority of $\mathcal{F}^*$. Hence $\mathcal{F}$ and $\mathcal{F}'$ intersect.
  2. One of $\mathcal{F}, \mathcal{F}'$ does not contain a majority of $\mathcal{F}^*$, *w.l.o.g.* it is $\mathcal{F}$. Both $\mathcal{F}, \mathcal{F}'$ contain *Min_Quorum* members of $\mathcal{WA}(\mathcal{F}^*)$. Moreover, $|\mathcal{F}.M \cap \mathcal{WA}(\mathcal{F}^*)| > |\mathcal{WA}(\mathcal{F}^*)| -$ *Min_Quorum*. By adding the two equations we get $|\mathcal{F}.M \cap \mathcal{WA}(\mathcal{F}^*)| + |\mathcal{F}'.M \cap \mathcal{WA}(\mathcal{F}^*)| > |\mathcal{WA}(\mathcal{F}^*)|$. Hence $\mathcal{F}$ and $\mathcal{F}'$ intersect.

  Since $\mathcal{F}$ and $\mathcal{F}'$ intersect, by Lemma 6, $\mathcal{F}'.N \neq \mathcal{F}.N$, a contradiction. Therefore $\mathcal{F}.N$ is unique among formed sessions.

  The existence of $\mathcal{FS}_\mathcal{F}$ implies, by Lemma 19, that $\mathcal{F} \in$ *Quorums_List*(S).

□

Theorem 3 *The transitive closure of $\prec$ on formed sessions is a total order.*

Proof: Lemma 20 implies that there exists a formed sequence $\mathcal{FS}$ containing all formed sessions in the system. For every two successive formed sessions, $\mathcal{F}_i, \mathcal{F}_{i+1} \in \mathcal{FS}$, $\mathcal{F}_i \in$ *Quorums_List*$(\mathcal{F}_{i+1})$, thus $\mathcal{F}_i$ and $\mathcal{F}_{i+1}$ intersect. Hence if $\mathcal{F}.N < \mathcal{F}'.N$, then $F \prec F'$. □