The Synchronization Cost of On-line Quorum Adaptation

Mark J. Bearden 
mbearden@ece.cmu.edu
Electrical and Computer Engineering Department
Carnegie Mellon University, Pittsburgh, Pennsylvania 15213-3890
http://www.ece.cmu.edu/afs/ece/usr/dsd/

Ronald P. Bianchini, Jr. 
rpb@ece.cmu.edu

Abstract

Quorum-based distributed mutual exclusion protocols require each processor in a distributed system to obtain permission from a quorum of processors before accessing a resource that cannot be concurrently shared. When processors fail, it is desirable to adapt the existing quorum assignment, rather than re-initializing the system after each failure or incurring the high overhead associated with masking failures. If quorums are adapted on-line, while the mutual exclusion protocol continues to operate, mutual exclusion is threatened, unless a global partial order is imposed on the steps taken by processors to adjust their quorum data structures. In this paper we characterize the coordination needed to guarantee correctness of mutual exclusion when quorums are adapted on-line. No assumptions are made about the method used to assign quorums or the maximum number of simultaneously faulty processors. A decentralized system model is assumed that provides no globally ordered communication mechanism. We present necessary and sufficient conditions for the synchronization needed to ensure mutual exclusion safety, and we establish bounds on the message passing cost of a distributed solution to on-line quorum adaptation.

Keywords: quorum system, mutual exclusion, synchronization, reconfiguration, fault-tolerance

1 Introduction

Mutual exclusion is the classic problem of coordinating access among several users to a shared critical resource such that no two users access the resource simultaneously [13]. A mutual exclusion protocol guarantees the following properties [24]: safety, or mutually exclusive access of the critical resource; fairness, indicating that any processor desiring access to the resource must eventually obtain access; and progress, meaning that the protocol must be free from deadlock.

In one distributed form of the mutual exclusion problem, users are networked processors that coordinate critical resource access by exchanging messages among themselves. In this paper we consider the class of solutions to distributed mutual exclusion that are quorum-based [2,15,16,26,34]. These protocols require each processor to request and obtain permission from a set of processors called a quorum before accessing the critical resource. Each processor is assigned a distinct quorum such that the intersection of any two processors’ quorums is not empty. This quorum intersection property ensures that any two competing processors must both obtain permission from some processor that is common to both their quorums. Thus it is possible to prevent two processors from accessing the critical section simultaneously. A quorum assignment is a set of quorums, one quorum for each processor participating in the mutual exclusion protocol. A quorum assignment is valid if and only if the quorums contained in the assignment satisfy the quorum intersection property. Many algorithms for generating valid quorum assignments have been suggested [2,8,9,10,21,26,29,31,37]. Sanders’ generalized form of the quorum-based mutual exclusion problem [34] gives a “generic” distributed mutual exclusion algorithm that solves mutual exclusion when used with any valid quorum assignment. We assume the use of Sanders’ algorithm, which is outlined here briefly.

Each processor that wishes to access the critical resource sends a request message to each processor in its quorum and then waits to receive grant messages in response. When a processor has received a grant from every member of its quorum, it can safely access the critical resource. When a processor gives up the critical resource, it sends a release message to all of the processors that previously sent it grant messages. Mutual exclusion is ensured by allowing each processor to grant permission to only one processor at a time; once a grant message has been sent by processor \( x \) to processor \( y \), processor \( x \) is “locked” such that \( x \) sends no other grant messages before it receives a release message from \( y \). To prevent deadlock from occurring when there are concurrent request messages from several processors, a logical ordering relation (such as a Lamport logical clock value [20]) is used to assign global precedence among competing requests. Full details of the mutual exclusion algorithm are given in [26,34].

1.1 Impact of Unavailable Processors

This paper focuses on the correctness of mutual exclusion when processors become unavailable. An unavailable processor is broadly defined as one that ceases to participate in the mutual exclusion protocol. This could be due to actual failure, or to voluntary reconfiguration activity such as removal from the system or a load balancing adjustment. To simplify discussion, the term “faulty” (“non-faulty”) is used interchangeably with “unavailable” (“available”).
Deadlock can result if a processor that has received a grant message fails before releasing the grant. The generic mutual exclusion algorithm is extended to prevent deadlock in [7,34,37]. Any locked processor that detects the failure of another processor must act as if it immediately receives any release messages it is waiting to receive from the failed processor. Since a processor must obtain grants from all of its quorum members before accessing the critical resource, faulty processors in quorums threaten fairness and/or progress. Safety is not directly threatened by failures, since a faulty processor will not grant permission to any processor that sends it a request. To ensure fairness and progress, the quorum assignment must mask the presence of faulty processors as in [3,7,25,31], or else the quorum assignment must be adapted so that faulty processors are removed from all quorums. Solutions that mask faulty processors incur higher overhead since a larger number of processors are required in the intersection of each two quorums in a valid quorum assignment. For example, if \( f \) or fewer processors can fail simultaneously, then each pair of quorums must contain at least \( 2f+1 \) processors in the intersection, and a processor may access the critical resource if grants are received from all but \( f \) members of the processors’ quorum [7]. Alternatively, failures may be masked by having each processor send requests to multiple quorums instead of just one quorum [3,25]. A processor can gain access to the critical resource when at least one of its quorums contains no faulty processors.

Rather than masking faulty processors, this work utilizes quorum adaptation, which allows smaller or fewer quorums in exchange for the occasional communication and delay needed to detect processor failures and adapt quorums. Each processor stores in local memory a list of the processors in its quorum. When a processor contained in one or more quorums fails, processors must make modifications to their local memories to remove references to the faulty processor. Even if the quorum assignment is valid before and after such modifications are made, it is possible for processors to update their local memories in an order such that the quorum intersection property is temporarily violated. Consider the example in Figure 1. Two intersecting quorums \( J \) and \( K \), shown in (a), exist at two processors (not shown) and form part of a valid quorum assignment. Initially, quorums \( J \) and \( K \) intersect since processor \( x \) is a member of both. Subsequently, \( x \) fails and is replaced with non-faulty processor \( y \) in both \( J \) and \( K \). The resulting state of \( J \) and \( K \), which also satisfies the quorum intersection property, is shown in (c). Suppose that as the processors transition from state (a) to state (c), the modifications to \( J \) and \( K \) are not made simultaneously. One possible sequence of modifications is replacing \( x \) with \( y \) in quorum \( K \) before \( J \) is modified, so that the quorums take on the arrangement shown in (b). The quorum assignment thus becomes temporarily invalid, and mutual exclusion safety is threatened if the mutual exclusion algorithm executes during the invalid state. Other possible sequences of transitions never cause \( J \) and \( K \) to have an empty intersection. For example, it is possible to transition from state (a) to (c) through the states shown in (d) and (e), adding \( y \) to both \( J \) and \( K \) before \( x \) is removed from either \( J \) or \( K \).

In this work, we address the distributed coordination problem of ordering the changes made by processors to the quorums such that the quorum assignment remains valid. The goal is to carry out quorum adaptation on-line, while the mutual exclusion algorithm continues to operate, even if no global synchronization is provided by the underlying system.

### 1.2 Related Research

A large body of research focuses on predicting the availability of non-adaptive quorum assignments, based on the assumption that processors in quorums fail with a given or measured probability. Such research has utilized theoretical models [2,3,30,31], empirical methods [1], and simulations [19,22]. These approaches do not deterministically guarantee that quorums will remain available (that they will satisfy progress and fairness) after processors fail. Algorithms that deterministically guarantee progress and fairness, by adapting quorum assignments when failures are detected, are proposed in [2,8,9,10,36,37]. These algorithms allow each processor to locally compute the list of processors in its own quorum, based upon changing local information about the availability of other processors. Valid quorum assignments are guaranteed as long as the number of simultaneously faulty processors is bounded by a number much smaller than the number of processors executing the protocol. Each of these deterministic quorum generation algorithms also restricts the “topology” of the quorum assignments and the minimum quorum size for a particular system size. Most of the quorum generation methods are based on a logical arrangement of the processors into a graphical or geometric topology, such as a tree or mesh. The quorum adaptation approach in [27,34] makes fewer restrictions on quorum topology but produces an adapted quorum assignment whose topology is strictly a function of the initial quorum assignment and the number of faulty processors. Fault-tolerant quorum construction using dynamic voting assignments [17,18,23] has also been investigated. All of these approaches place restrictions on the set of quorums that can result from quorum adaptation, and most require at least a majority of the processors to be non-faulty at any time in order to adapt the quorum assignment.

### 1.3 Definition of On-line Quorum Adaptation

The present research differs from previous work by investigating quorum adaptation without bounding the num-

![Figure 1. Quorum adaptation example.](image-url)
ber of simultaneously faulty processors or restricting the size or topology of quorums. We define the problem of on-line quorum adaptation (abbreviated OQA) as follows.

**Definition (OQA):** Given a set of processors that use a dynamic quorum assignment QA; and given that each processor has an input called QA-desired that takes on a succession of desired quorum assignment values; an on-line quorum adaptation protocol guarantees the following two properties:

- **OQA-Safety:** The value of QA always represents a valid quorum assignment;
- **OQA-Progress:** It is always true that the system will eventually reach a future state such that (a) QA has the same value as QA-desired at every processor and (b) no faulty processor is contained in any quorum of QA.

OQA-Safety requires that the quorum assignment QA must always be valid; though the quorums in QA may temporarily contain faulty processors. However, the processors’ values for QA-desired may be invalid quorum assignments, and processors may disagree with other processors about the value of QA-desired. OQA-Progress requires that inconsistent or invalid values for QA-desired persist only temporarily. Thus OQA-Progress requires the following assumption, that the value of QA-desired “stabilizes.”

**Definition (stabilizing quorum generation function):** A quorum generation function exists that provides each processor with a value for QA-desired. The function’s output is considered to be stable whenever each processor agrees with the other processors on a valid desired quorum assignment made up of quorums containing only non-faulty processors. It is always true that QA-desired will eventually remain stable for an interval of time with duration greater than \( a \), for any arbitrarily selected time constant \( a \). Informally, the values of QA-desired must eventually remain consistent, valid, and free of faulty processor members, long enough for an OQA protocol to ensure that QA “catches up” with QA-desired.

Section 3.3 describes how a stabilizing quorum generation function can be constructed from a system diagnosis service such as [5], that guarantees non-faulty processors will eventually identify all of the faulty processors in the system.

### 1.4 Approach and Contributions

A distributed solution to the OQA problem is investigated using a general distributed processing model that assumes processors may repeatedly fail and recover. No totally ordered membership or view-synchronous communication services [6,14] are available to globally order messages or notifications of processor failures. Although it is often argued that the view-synchronous communication model greatly simplifies the implementation of coordination problems like OQA, this model is implemented only at the cost of potentially unnecessary synchronization [11,32]. In pursuit of a solution that minimizes synchronization, we derive in Section 3.1 the minimum partial ordering of events required to solve OQA. These results are extended in Sections 3.2 and 3.3 to obtain lower and upper bounds on message passing needed to solve OQA in a “loosely synchronous” distributed system (defined in the next section). Section 4 gives conclusions and comments on the applicability of this research.

### 2 System Model

A system \( S \) is a countable set of processor elements. Each processor has its own private storage containing information about its own status and the status of other processors. Individual processors are allowed to leave and rejoin \( S \) any number of times, and processors may fail and recover while in \( S \). Processors communicate only by exchanging messages over a network that is logically fully connected. Communication in \( S \) is loosely synchronous: Processors may execute at different speeds, and message delivery times may be non-deterministic, but there exist known upper bounds on the relative processing speed of any two processors, and on the time needed to deliver a message.

Messages passed between non-faulty processors are reliably delivered unless the processors are separated by a communication network partition [33,35], such that multiple sets of non-faulty processors cannot communicate with those in different sets. If partitions occur, they exist only temporarily. Meanwhile the processors in at most one “primary” partition will execute the mutual exclusion protocol. Implementation of this characteristic is highly problematic in a real distributed system, but it must be assumed by a deterministically correct distributed mutual exclusion protocol [12].

#### 2.1 Mutual Exclusion Protocol Assumptions

Without loss of generality, there is one critical resource associated with \( S \). Some of the processors in \( S \) choose to execute a distributed quorum-based mutual exclusion protocol for sharing the resource. The mutual exclusion protocol is divided into three interacting modules that may be implemented independently of each other: They are (1) a permission algorithm, (2) a (stabilizing) quorum generation function, and (3) an OQA algorithm. The permission algorithm is the “generic” fault-tolerant mutual exclusion algorithm [34] already mentioned. For each processor \( p \) that executes the mutual exclusion protocol, \( Q^p \) denotes the quorum of \( p \) stored in \( p \)'s local memory. Each \( p \) also stores a flag called \( mutex^p \) which must equal \( \text{TRUE} \) when \( p \) is executing the permission algorithm and \( \text{FALSE} \) when \( p \) is not executing the permission algorithm. When \( p \) joins \( S \) or recovers from a failure, \( mutex^p \) initially equals \( \text{FALSE} \). The quorum intersection property must always be satisfied for quorum assignment \( QA = \{ Q^p \} \). \( Q^p \) is not required to satisfy the quorum intersection property with respect to other quorums if \( mutex^p \) equals \( \text{FALSE} \).

At each processor \( p \), the quorum generation function provides QA-desired\(^p \), \( p \)'s local view of the desired quorum assignment. As changes occur to the value of QA-desired\(^p \) at
each \( p \), the \textit{OQA algorithm} must update the value of the local quorum \( Q^p \) in a manner that satisfies the \textit{OQA-Safety} and \textit{OQA-Progress} properties. The OQA algorithm also determines when a processor that wishes to participate in mutual exclusion can safely set \( \text{mutex}^p \) equal to \text{TRUE}. The following definitions are used to discuss the operation of the OQA algorithm.

**Definition:** The \textit{mutual exclusion set}, or \( S_{\text{mutex}} \), is the set \( \{ x \mid \text{mutex}^x = \text{TRUE} \} \). These are the processors that “know” they may safely request access to the critical resource.

**Definition:** A \textit{quorum intersection set} \( R \) is a subset of \( S \) such that for any pair \( x,y \) of non-faulty processors in \( R \), \( Q^x \cap Q^y \neq \emptyset \). \( Q(I) \) denotes \( R \) is a quorum intersection set.

Formally, the \textit{OQA-Safety} property is expressed thus:

\[
\exists X \subseteq S \mid S_{\text{mutex}} \subseteq X \land Q(I(X)).
\]

**2.2 Expressing Evolution of System State**

The following notation defines the system state that is relevant to execution of the mutual exclusion protocol. The system’s \textit{configuration}, \( C \), is defined by the current values of \( Q^p \) and \text{mutex}^p for each \( p \) in \( S \), such that when \( S = \{a,...,z\} \), \( C = \{Q^a, \text{mutex}^a, ..., Q^z, \text{mutex}^z\} \). \( S \) acquires a sequence of states \((C_0, C_1, ..., C_\ell)\), \( C_i \neq C_{i+1} \), such that successive states capture each change made to \( C \). \( C_0 \) is an initial starting state.

Let \( Q^p \) and \text{mutex}^p denote the values of \( Q^p \) and \text{mutex}^p \) at state \( C_i \). Let \( S_{\text{mutex}}(C_i) \) and \( S_i \) represent the mutual exclusion set and the set of processors in \( S \), respectively, when \( S \) has state \( C_i \). \textit{OQA-Safety}(\( C_i \)) is satisfied if and only if \( Q(S_{\text{mutex}}(C_i)) \) is true at state \( C_i \).

Configuration \textit{event} \( E_{i+1} \) corresponds to the transition from state \( C_i \) to state \( C_{i+1} \). Four classes of configuration events—\textit{Start}, \textit{Stop}, \textit{Add}, and \textit{Del}—are defined in Table 1. Each unique event, other than a \textit{Stop} event, changes the value of only one data structure at a single processor. A subscript \( i \) is associated with an event (i.e., ‘Add\(_i\)\((Q^p, y)\)’) to indicate that the event corresponds to the transition between configuration states \( C_{i-1} \) and \( C_i \). An event that occurs at an unspecified transition is denoted without a subscript. For closure, initial transition \( E_0 \) implies the existence of a set of pseudo-events \( \{\text{Start}(0, K) \mid p \in S_{\text{mutex}}(C_0) \wedge Q^p = K\} \).

System execution is a discrete sequence of alternating configuration states and events. System actions that are not defined as events, such as sending messages and recovering from failures, may also occur at any time relative to configuration events. System execution \textit{intervals} are denoted using square brackets around the states that indicate closed endpoints of the interval. For example, the interval “\([C_1, ..., C_5]\)” consists of all states \( C_i \), \( 1 \leq i \leq 5 \) and includes all events \( E_j \), \( 1 \leq j \leq 5 \).

**3 Synchronization Cost of OQA**

This section presents the synchronization required to guarantee \textit{OQA}. First, we give the partial ordering of events in Table 1 that is necessary and sufficient for \textit{OQA-Safety}. It is shown that when a stabilizing quorum generation function is present, this partial ordering completely characterizes the synchronization cost of OQA.

**3.1 Partial Event Order due to OQA**

The following lemma identifies the events that must occur (and not occur), and in what order they must occur, to satisfy an assertion about the contents of some processor’s quorum, when that processor is in \( S_{\text{mutex}} \) at a particular state.

**Lemma 1.** A processor \( p \) is in \( S_{\text{mutex}}(C_i) \) and has \( y \) in its quorum \( Q^p_i \) if and only if

\[
\exists E_{i_0}^p \leq i \leq E_{i_1}^p \mid \begin{align*}
E_{i_0}^p &\in \{\text{Add}_i(Q^p, y), \text{Start}(p, (K \mid y \in K))\} \\
E_{i_1}^p &\notin \{\text{Del}_i(Q^p, y), \text{Stop}(p)\}
\end{align*}
\]

**Proof:** The proof follows directly from the definitions given in Table 1.

**Lemma 2.** Given that \textit{OQA-Safety}(\( C_i \)) is true and that event \textit{Del}(\( Q^p, x \)) occurs, then \textit{OQA-Safety}(\( C_{i+1} \)) is true if and only if

\[
\forall q \mid q \in S_{\text{mutex}}(C_i) \land x \in Q^p_i \cap Q^q_i, \exists y, y \neq x \mid y \in Q^p_i \cap Q^q_i.
\]

**Proof (necessity):** Assume that \textit{OQA-Safety}(\( C_{i+1} \)) is true but that condition \( C_2 \) is not satisfied. It follows that there must be some \( q \) in \( S_{\text{mutex}}(C_i) \) such that \( Q^p \cap Q^q \) contains only \( x \) in state \( C_i \). Otherwise, \( C_2 \) is satisfied. But after \( x \) is removed from \( Q^p \) by the \textit{Del}(\( Q^p, x \)) event, \( Q^p \cap Q^q \) must be
empty in state $C_{i+1}$. Thus, any subset of $S_{i+1}$ containing $q$ and $p$ cannot be a quorum intersection set. But $q$ and $p$ are both still in $S_{mutex}(C_{i+1})$, and satisfying $OQA-Safety(C_{i+1})$ requires that they be contained in the same quorum intersection set. This is a contradiction.

(Sufficiency): Assume that condition C2 is met but that $OQA-Safety(C_{i+1})$ is not satisfied. Then there must be two processors $k$ and $m$, both in $S_{mutex}(C_{i+1})$, such that $Q^k_{i+1} \cap Q^m_{i+1}$ is empty. Otherwise, $Q(S_{mutex}(C_{i+1}))$ would be true and $OQA-Safety(C_{i+1})$ would be satisfied. Processors $k$ and $m$ must be in $S_{mutex}(C_i)$ since no stop event occurred at the transition from $C_i$ to $C_{i+1}$. Then $Q^k_i \cap Q^m_i$ is non-empty, since $OQA-Safety(C_i)$ is true. Either $k$ or $m$ must be processor $p$, and $x$ must be the only processor in $Q^k_i \cap Q^m_i$; otherwise $Q^k_{i+1} \cap Q^m_{i+1}$ would also be non-empty. But because condition C2 is true, there must also be a $y$ in $Q^k_i \cap Q^m_i$ that is distinct from $x$. The assumption leads to a contradiction; Q.E.D.

By substituting Lemma 1 into Lemma 2 we obtain Theorem 1, which establishes the minimum partial order of configuration events that guarantees $OQA-Safety$ when a single processor in $S_{mutex}$ removes a member from its quorum.

**Theorem 1:** Given that $OQA-Safety(C_i)$ is true and that event $Del_{i+1}(Q^p,x)$ occurs, then $OQA-Safety(C_{i+1})$ is true if and only if for all $q$ such that $q \in S_{mutex}(C_i)$ and $x \in Q^p_i \cap Q^q_i$, there exists processor $y$, $y \neq x$, such that

$$\forall q \in \{p,q\}, \exists y \in [p,q], \left(\exists E^y_g \quad \forall z \in \{y,q\}, E^y_g \in \{Add_g(Q^z_i,y), Start_g(z,(K \cap y \in K))\} \land \neg D^y_h \quad \forall z \in \{y,q\}, \exists E^y_g \quad \forall z \in \{y,q\}, E^y_g \in \{Del_h(Q^z_i,y), Stop_h(z)\} \right)$$

As previously illustrated in Figure 1, a Del event performed out of order with respect to other configuration events leads to a violation of the quorum intersection property. The event ordering given in Theorem 1 is illustrated on the timelines in Figure 2. Before a Del event is allowed to remove $x$ from $p$’s quorum, $p \in S_{mutex}$, there must have occurred an Add or Start event that added some $y$, distinct from $x$, to $p$’s quorum and to the quorum of $q$, for every other $q$ in $S_{mutex}$. The Add/Start events marked on the timelines are required to occur at any two processors $p$ and $q$ that are in $S_{mutex}$ before $p$ removes $x$ from $Q^p$, when $x$ is also in $Q^q$. An arrow from one event to another indicates that the first event must happen before the second event. Theorem 1 also explicitly requires the absence of events that would “undo” the effect of the required Add or Start events. During the intervals shown as broken timeline, there must be no Del$(Q^p, y)$ or Stop event at $p$ and similarly no Del$(Q^q, y)$ or Stop event at $q$. Next, consider the event ordering that preserves the OQA-Safety property when a Start event occurs.

**Lemma 3:** Given that $OQA-Safety(C_i)$ is satisfied and event $Start_{i+1}(p,K)$ occurs, then $OQA-Safety(C_{i+1})$ is true if and only if

$$\forall q \in S_{mutex}(C_i), \exists y \quad y \in K \cap Q^q_i.$$ 

The proof of Lemma 3, omitted here but given in [4], is similar to the proof of Lemma 2. Substituting Lemma 1 into Lemma 3 yields Theorem 2.

**Theorem 2:** Given that $OQA-Safety(C_i)$ is satisfied and event $Start_{i+1}(p,K)$ occurs, then $OQA-Safety(C_{i+1})$ is true if and only if for each $q$ such that $q \in S_{mutex}(C_i)$ there exists a processor $y$ such that $y$ is in $K$, and

$$\exists q \in S_{mutex}(C_i), \exists y \in K \cap Q^q_i.$$ 

Consider any event $E_i$, $i > 0$, in the interval $[C_{i-1}, C_i]$ such that $OQA-Safety(C_{i-1})$ is satisfied. If $E_i$ is either a Del or a Start event, Theorems 1 and 2 prove that the given event ordering is both necessary and sufficient for $OQA-Safety(C_i)$ to be satisfied. Consider the other two possible events types: Add or Stop. If $E_i$ is an Add event, $OQA-Safety(C_i)$ is obviously true whatever the respective ordering of $E_i$; since adding a processor to a quorum can never decrease the size of any existing quorum intersection set. If $E_i$ is a Stop event, $OQA-Safety(C_i)$ must be true whatever the ordering of $E_i$ with respect to other events. Assume otherwise; then $E_i$ removes $p$ from $S_{mutex}(C_{i-1})$, and $S_{mutex}(C_i)$ contains two processors $k$ and $m$ whose quorums contain no common processor. Since $E_i$ modified the quorum of no processor that remains in $S_{mutex}(C_i)$, the intersection of the quorums of $k$ and $m$ must have already been empty before the Stop event. This contradicts the assumption that $OQA-Safety(C_i)$ is satisfied.
Safety$(C_{i+1})$ is satisfied. By induction, OQA-Safety is satisfied for all configuration states that follow state $C_0$. Q.E.D.

The next theorem follows from Theorem 3 and the definition of a stabilizing quorum generation function (Section 1.3). The proof is given in [4].

**Theorem 4:** Given a system execution interval $[C_0...]$ such that OQA-Safety$(C_0)$ is satisfied, and given a stabilizing quorum generation function, the partial event ordering given by Theorems 1 and 2 is necessary and sufficient to satisfy both OQA-Safety and OQA-Progress for all $C_i$ $i>0$.

### 3.2 Lower Bounds on OQA Message Cost

This section considers the number of interprocessor messages that must be exchanged to preserve OQA-Safety under the loosely synchronous system model. Since the quorum generation function may temporarily present an arbitrary quorum assignment, it does not provide the ordering of configuration events needed to preserve OQA-Safety. Given that interprocessor communication is accomplished only by message passing (Section 2), then guaranteeing that any two events occur in a particular order at two different processors necessitates the sending of one or more messages. Otherwise, due to loose synchrony, the events could occur in the wrong order. Thus, two events are ordered by an OQA algorithm in a loosely synchronous system by Lamport’s happens before relationship [20]. Event $a$ happens before event $b$ if and only if (1) $a$ and $b$ are events at the same processor and $a$ comes before $b$ or (2) one processor executes event $a$, then sends a message to a second processor, which receives the message before executing event $b$. This relationship is transitively extended as follows. Event $a$ happens before event $b$ at a different processor if a sequence of messages called a message chain exists such that some processor executes event $a$ before sending the first message in the chain; the last message in the chain is received by processor $z$ before event $b$ occurs at $z$; and every $i$th message in the chain (other than the last message) is received at some processor $x_i$ before the next message in the chain is sent by $x_i$. If event $b$ is required to follow event $a$, $a$ and $b$ at different processors, then a message chain must exist that connects $a$ to $b$.

A lower bound on the message passing cost of an OQA algorithm is given using the following definition of a round of quorum adaptation, called an $\alpha$-step.

**Definition (\alpha-step):** An $\alpha$-step is a transition from a valid quorum assignment to a successive valid quorum assignment. During an $\alpha$-step, a set of $Add$ and $Del$ configuration events affect the removal of exactly one processor $r$ from each of the $M$ quorums that $r$ appears in. Before its removal, $r$ is the only processor in the intersection of any two quorums that contain $r$. No $Start$ or $Stop$ events occur during an $\alpha$-step.

As an illustration, consider a system of four processors $\{a,b,c,d\}$ and an $\alpha$-step that causes a transition from quorum assignment $\{Q^a\{x,b\}, Q^b\{x,c\}, Q^c\{x,d\}, Q^d\{x,a\}\}$ to a second quorum assignment $\{Q^a\{y,b\}, Q^b\{y,c\}, Q^c\{y,d\}, Q^d\{y,a\}\}$.

In this example $M$ equals four. The given $\alpha$-step replaces processor $x$ in each quorum with $y$. The four events \{$Del(Q^a\{x\}) \mid p \in \{a,b,c,d\}\$} must occur to satisfy OQA-Progress. Theorem 1 requires that the $Del(Q^a\{x\})$ event must not occur until after $Add/Start$ events have added some processor $p_{ar}\cap b$, distinct from $x$, to each $Q^a$ and to $Q^b$. Likewise, $Add/Start$ events must occur that add $p_{ar}\cap c$, distinct from $x$, to $Q^c$ and to $Q^d$, and $p_{ar}\cap d$ to both $Q^d$ and $Q^f$, before the $Del(Q^a\{x\})$ event happens. The minimum number of events results when $p_{ar}\cap b = p_{ar}\cap c = p_{ar}\cap d = y$, since the quorum assignment that must result contains $y$ in every quorum. The same argument follows for the other Del events: Each $Del(Q^a\{x\})$ must be preceded by $Add/Start$ events that occur at each of the $M$-1 processors other than $p$. Thus $M(M-1)$ ordering dependencies result from the $\alpha$-step. To satisfy these dependencies, at least $M(M-1)$ message chains must exist, one from each processor to every other processor in the set of $M$ processors whose quorums initially contain $x$.

Figure 4 illustrates two ways of constructing a message chain between every ordered pair among the four processors. Each arrow represents a single message. In Figure 4(a), the $2M$-2 messages are sent and delivered in the order indicated by the numeric labels. Figure 4(b) shows one message corresponding to each of the $M(M-1)$ dependencies. In both cases, it is assumed that all of the required $Add/Start$ events occur prior to the sending of any messages, and the Del events occur after all messages are received. The example of 4(a) requires that messages are complete-history messages, containing information about all of the events that the sender learned about through previously received messages. The example of 4(b) may be accomplished using single-event messages that contain information only about the occurrence of a single configuration event. The examples serve to illustrate the lower bounds on message passing that are stated in the following theorem.

**Theorem 5:** Given a solution to OQA in a loosely synchronous system, and that each processor is contained in $M$ quorums, then to complete an $\alpha$-step it is necessary to send at least $2M-2$ complete-history point-to-point messages, at least $M(M-1)$ single-event point-to-point messages, or at least $M$ broadcast messages.

**Proof:** By Theorem 1, an $\alpha$-step requires $M(M-1)$ event ordering dependencies between distinct ordered pairs of processors, as discussed above. If single-event point-to-point messages are sent, at least one message per dependency is required. If messages are broadcasts, at least one message must be sent by each of the $M$ processors. It remains to be shown that $2M-2$ complete-history point-to-point messages or at least $M$ broadcast messages are necessary. Assume that exactly $M-1$ messages are sent by

![Figure 4. Message passing for adaptation step](image-url)
the $M$ processors. Then at least one processor $p_x$ sends no message, and therefore no message chain can be established from $p_x$ to any other processor. Thus if only $M-1$ messages are sent, at most one processor ($p_y$) can establish a message chain from all other processors to itself. Consider a set $T$ of complete-history messages that establish the required set of message chains, one chain from each processor to every other processor. Consider set $U$ that consists of the first $M-1$ messages delivered in $T$. The messages in $U$ can establish all required message chains to at most one of the processors. This leaves $M-1$ other processors that must each receive one message not in $U$ to establish the remaining required message chains. Thus there must be at least $2M-2$ messages in $T$. Q.E.D.

Theorem 6 gives a lower bound on the message passing cost of preserving OQA-Safety when a processor $p$ sets $mutex^p$ to TRUE. The proof, given in [4], is similar to that of Theorem 5 and follows directly from the order requirements of Theorem 2.

Theorem 6: Given a solution to OQA in a loosely synchronous system and $N$ processors executing the mutual exclusion permission algorithm. If a processor executes a Start event, then at least $N$ interprocessor messages must be sent.

3.3 Upper Bounds on OQA Message Cost

Two algorithms that provide upper bounds on the cost of solving the OQA problem are briefly mentioned. In [25] Lynch and Shvartsman give an on-line protocol for adapting a quorum assignment that is used to provide atomic ordering of reads/writes to a shared register, where each processor utilizes a set of intersecting quorums rather than a single quorum. Their three-phase algorithm assumes that a single processor acts as a reconfiguration controller and utilizes the intersection property of the existing quorum assignment to safely propagate a new quorum assignment. No particular mechanism is specified for determining new configurations. Message losses and processor failures are tolerated, as long as the processors in at least one quorum, along with the reconfiguration controller, experience no failures during execution of the protocol. No ordered or reliable broadcast is assumed. Installing a new quorum assignment requires $6n$ or fewer point-to-point messages and has best case latency of $6d$, where $d$ is the time needed to deliver a message. Message count can be reduced to $3n+3$ if hardware-assisted broadcast is available.

In [4] a fully decentralized OQA algorithm called QADAPT is presented. All processors participate equally, and up to $n-1$ simultaneous processor failures are allowed. Reliable but unordered message delivery is assumed between each pair of non-faulty processors. A stabilizing quorum generation function is constructed as follows. The system diagnosis algorithm implemented in [5] allows each non-faulty processor to identify the other non-faulty processors in the system. Each processor also executes a quorum construction algorithm ([2,8,26,29]), using the list of non-faulty processors as input, to determine a value for the desired quorum assignment. The desired quorum assignment values stabilize when all non-faulty processors agree on the correct list of non-faulty processors. The analysis in [5] shows that this must occur within a bounded time $L$ after the most recent failure. The assumption is made that there always eventually exists an interval longer than $L$ during which no processors fail or recover. Given a balanced quorum assignment [26] with all processors in $M=\sqrt{n}$ quorums, QADAPT requires between $1.5(n-\sqrt{n})$ and $2(n-\sqrt{n})$ messages to execute an $\alpha$-step (Section 3.2). Determining the new desired quorum assignment requires an additional $2n$ to $3n$ messages generated by the diagnosis algorithm. Best case latency for an $\alpha$-step is $2d$. Worst case message count for an $\alpha$-step reduces to $2\sqrt{n}$ if unordered reliable broadcast is available.

4 Conclusions

This paper formally defines on-line quorum adaptation (OQA) as the problem of coordinating modifications to quorums such that the quorum intersection property required for mutual exclusion safety is continuously preserved. The distributed mutual exclusion problem is conceptualized as three interacting modules that may be implemented independently of each other: (1) a Sanders-style “generic” permission algorithm, (2) an OQA algorithm, and (3) a quorum generation function. The OQA algorithm provides the partial ordering of events needed to safely install a new quorum assignment, allowing use of any quorum generation function that eventually stabilizes. This modular definition makes OQA potentially applicable to the wide variety of problems—other than mutual exclusion—in which quorums are utilized, such as those that maintain shared data consistency [3,16,17,25]. The OQA problem definition imposes no assumptions about the size or topology of quorums or the maximum number of simultaneously faulty processors.

A characterization is given of the system event ordering imposed by any solution to the OQA problem. The characterization is extended to show theoretical bounds on OQA message passing cost when no globally ordered communication mechanism is available. If OQA is solved for $n$ processors and balanced quorums [26], $O(n)$ point-to-point messages are necessary and sufficient for a simple round of quorum adaptation called an $\alpha$-step. If hardware-supported broadcast is assumed, then $O(\sqrt{n})$ messages are necessary and sufficient for an $\alpha$-step. A lower bound of $2\sqrt{n}$-2 messages is shown for the cost of an $\alpha$-step, if point-to-point messages with unbounded information content are assumed. Future work includes investigation of the applicability of this work to distributed coordination problems other than mutual exclusion, and further investigation of the relationship between the required partial ordering of events and the costs of OQA implementation.

Acknowledgments: The authors wish to thank Mark Stahl and Nina Banerjee for their valuable comments on this work.
References


