A Single Phase Protocol for Total and Causal Ordering of Group Operations in Distributed Systems

Chengzheng Sun
School of Computing & Information Tech.
Griffith University
Brisbane, QLD 4111, Australia
Email: C.Sun@cit.gu.edu.au

Piyush Maheshwari
School of Computer Science & Engineering
The University of New South Wales
Sydney, NSW 2052, Australia
Email: piyush@cse.unsw.edu.au

Abstract

The total and causal ordering of group operations in distributed systems is at the core of solutions to many problems in distributed computing. Existing approaches are based on either a centralized coordinator to assign a sequence number for each operation or on a distributed two-phase protocol to determine the total ordering. This paper proposes a distributed single-phase protocol which determines the total ordering of group operations at the moment when operations are generated, without the need for a second phase of determining the sequence number. Therefore, the latency time can be kept in the range of $[0, d]$, where d is the maximum message propagation delay between any pair of sites in the system. Furthermore, in contrast to the arbitrary total ordering imposed by two-phase protocols, the total ordering imposed by the proposed protocol is consistent with the causal ordering among group operations, which is required by many distributed applications. With an optimized acknowledging scheme integrated, the number of acknowledgment (overhead) messages involved in each group operation can be kept in the range of $[0, N-1]$ in multicasting networks, or $[0, (N-1)^2]$ in point-to-point networks. Moreover, the proposed protocol is self-adaptive to the group operation traffic: the heavier the group operation communications traffic, the shorter the average latency time and the less overhead messages in the system. Both theoretical analysis and software simulation have been conducted to verify the correctness of the proposal protocol.

Keywords: total ordering, causal ordering, logical clocks, group operations, distributed protocols.

Journal of Computing and Information, Vol. 2, No. 1, 1996, pp. 219-237.
Special Issue: Proceedings of Eighth International Conference of Computing and Information (ICCI'96), University of Waterloo, Waterloo, Ontario, Canada, June 19-22, 1996.

1 Introduction

A *group* operation in a distributed system is an operation generated by one process but must be propagated and executed by all processes of the same group in the system. A group operation can be of any kind, for example, the delivery of a message to a group of processes, or an update operation on the multiple copies of a shared file. The total and causal ordering of group operations in distributed systems is at the core of solutions to many problems in distributed computing, such as consistency maintenance in distributed shared memory systems [10], parallel and fault tolerant computing based on group communications [2, 9], and concurrency control in groupware/CSCW (Computer Supported Cooperative Work) systems [5, 7, 13, 14]. The order in which group operations are processed at different processes is an important issue, not only because it is often necessary that particular ordering constraints must be obeyed for correctness, but also because meeting ordering requirements carries certain expenses, and protocols designed to guarantee a particular ordering can be expensive to implement. Existing approaches to the total and causal ordering of group operations in distributed systems can be classified into the following categories:

- Centralized coordinator-based protocols [9, 15]. In this approach, the total ordering of group operations is determined by using a central coordinator. First, a group operation is sent to the coordinator. Then, the coordinator assigns a sequence number to the group operation and multicast it to all processes in the same group. The main advantage of this approach is its simplicity and efficiency in terms of the number of messages involved in each group operation. Main problems with this approach are the single point of failure and potential performance bottleneck.

- Distributed two-phase protocols [1, 4, 6]. In this approach, the total ordering of group operations is achieved by executing a distributed two-phase protocol among all processes involved. In the first phase, the originator of a group operation multicasts the operation to all processes in the group, bearing a temporary identifier. After receiving a remote group operation, each site buffers it in an internal waiting queue and responds with a proposal for the operation's final identifier. In the second phase, the originator of the operation collects all the proposed identifiers, selects the largest one as the final agreed identifier, and then multicasts it. A buffered group operation becomes stable for execution when it has been assigned a final identifier and has reached the front of the waiting queue. This approach has no bottleneck or single point of failure for transmitting messages. But it requires two rounds of messages before the total ordering of a group operation can be decided, which is slow and expensive.

Journal of Computing and Information, Vol. 2, No. 1, 1996, pp. 219-237.

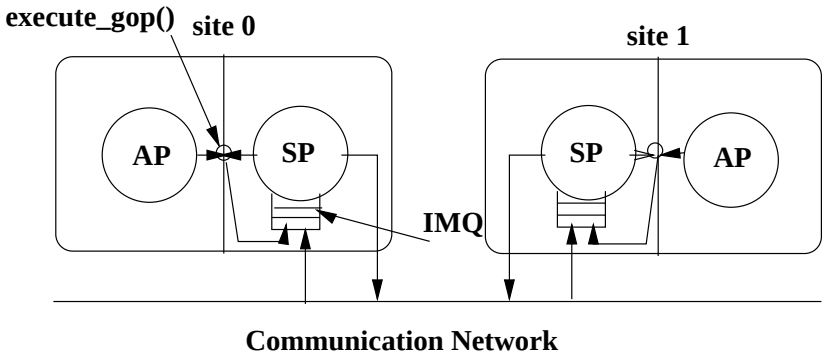

Figure 1: The system model

Moreover, the total ordering chosen by this approach is arbitrary and could potentially conflict with causality relationship among operations [1, 3].

In this paper, we propose a distributed *single-phase* protocol, which determines the global ordering of group operations at the moment when operations are generated, and stabilizes a group operation as soon as possible, without any delay caused by the need for a second phase of assigning a proper sequence number as in two-phase protocols. Moreover, in contrast to the arbitrary total ordering imposed by two-phase protocols, the total ordering imposed by the proposed protocol is consistent with the causal ordering among operations, which is required by many distributed applications [9, 10, 11].

The rest of the paper is organized as follows: First, a system model used for the design and discussion of our protocol is described in Section 2. Then a system of logical clocks is introduced, on which the total and causal ordering relation among group operations is defined in Section 3. After that, the basic protocol is proposed and discussed in Section 4. Next, an optimized version of the protocol is described in Section 5. Finally, the concluding remarks are given in Section 6.

2 The system model

Our system model for the design and discussion of the protocols is shown in Figure 1. It consists of N sites connected by a communication network. The N sites are identified by integers $0, 1, ..., N-1$, respectively. Each site is able to communicate with any other site in this system, though all sites may not be fully connected. The underlying communication subsystem is assumed to be reliable and to deliver messages between any pair of sites in the order as they were sent, which is referred as the FIFO property. There are one *application process* (AP) and one *server process* (SP), interfaced

Journal of Computing and Information, Vol. 2, No. 1, 1996, pp. 219-237.

by procedure `execute_gop()`, on each site in the system.

- The AP runs the application program. When AP needs to execute a group operation, it invokes the procedure `execute_gop()`.
- The `execute_gop()` procedure provides an interface between the AP and the SP. When it is invoked by the AP, it first constructs a message encapsulating the group operation, then inserts the message to the *head* of the *input message queue (IMQ)* associated with the local SP, finally returns to the AP after the group operation has been executed by the local SP.
- The SP receives, executes, and propagates group operations. It works in a loop of getting a message from its *IMQ* or blocking itself if there is no message in the *IMQ* until one arrives, executing the group operation encapsulated in the message if it is in the right order determined by the protocol, or putting it in an internal buffer if it is not, and multicasting the group operation to other SPs in the system if the operation was originated from the local site.

When a remote message arrives at a site, it is placed at the *tail* of the *IMQ* of the SP by the network interface. Since a local message from the local AP is always inserted to the head of *IMQ*, it will be processed by the SP as soon as it was generated. Furthermore, since the interface procedure `execute_gop()` does not return until the group operation has been executed by the local SP, there can be at most one outstanding local group operation in the *IMQ*.

3 Total and causal ordering based on logical clocks

Following Lamport [8], we first define a causal (partial) ordering relation among group operations in terms of operation generation and execution sequences, and a system of logical clocks. Then a total ordering relation among operations is defined by the combination of logical clocks and process identifiers.

Definition 1 *causal ordering relation "$\rightarrow$"*
Given two group operations a and b, generated at site i and j, respectively, then $a \rightarrow b$, *iff*:

1. $i = j$ and the generation of a *happened before* the generation of b, or
2. $i \neq j$ and the execution of a at site j *happened before* the execution of b, or
3. there exists a group operation x, such that $a \rightarrow x$ and $x \rightarrow b$.

Journal of Computing and Information, Vol. 2, No. 1, 1996, pp. 219-237.

Definition 2 *The logical clock based timestamping scheme*
Each site maintains a local variable, called *logical clock (LC)*. All messages are timestamped by the value of their local LC at the time of their multicasting at the originating sites. Consider the logical clock LC_i at site i, $0 \leq i < N$:

1. Initially, $LC_i := 0$.
2. **Updating Rule 1 (UR1):** After receiving a local message with a group operation, LC_i is updated as follows: $LC_i := LC_i + 1$; and then the local group operation is multicast to all other sites.
3. **Updating Rule 2 (UR2):** After receiving a remote message m with a timestamp $LC(m)$, LC_i is updated as follows: $LC_i := max(LC_i, LC(m))$.

There are a few points worth mentioning:

- Since the generation of a local group operation by the AP is *immediately* (and indivisiblely) followed by the receiving of a local group operation by the SP according to the system model described in Section 2, **UR1** could equally be stated as: *After generating a local group operation, LC_i will be incremented by 1.*
- After receiving a remote message m (of any type), the clock is updated to $max(LC_i, LC(m))$ according to **UR2**, instead of $max(LC_i, LC(m)) + 1$, which was required by the rules of implementing logical clocks [8, 12]. This is because it is not necessary, for our purpose, to distinguish a sending event from the corresponding receiving event. We will show that maintaining logical clocks according to **UR1** and **UR2** is both correct (see the following theorem) and beneficial in saving the number of acknowledgement messages needed for implementing the total ordering of all group operations in the system (see the discussions in Section 4).
- There may be various types of messages sent out from each site, including the group operation messages and other control messages required by the protocols (see Section 4). All type of messages are timestamped, but only group operation messages from the same site are guaranteed to carry distinguishable timestamps among them. This is intentionally chosen because we are only interested in the order of group operations but not other types of messages in this system.

Theorem 1 *The logical clock based timestamping scheme possesses the fundamental monotony property, i.e., given two group operations a and b, if $a \rightarrow b$, then $LC(a) < LC(b)$, where $LC(a)$ is the timestamp of a, and $LC(b)$ is the timestamp of group operation b.*

Proof: According to Definition 1, there must be a sequence of group operations $a = O_1, O_2, ..., O_{m-1}, b = O_m$, generated at sites $s_1, s_2, ..., s_m$, such that for $i \in \{0, 1, ..., m-1\}$, $O_i \rightarrow O_{i+1}$ due to either of the two reasons:

1. $s_i = s_{i+1}$ and the generation of O_i happened before the generation of O_{i+1}, or
2. $s_i \neq s_{i+1}$ and the execution of O_i at s_{i+1} happened before the generation of O_{i+1}.

In the following, we will show that $LC(O_i) \leq LC(O_{i+1})$ in all the two cases above:

1. In the first case, since the logical clock is always incremented by 1 after generating a group operation according to **UR1** of Definition 2, $LC(O_{i+1})$ must be at least $LC(O_i) + 1$.
2. In the second case, since the execution of O_i at s_{i+1} happened before the generation of O_{i+1} at s_{i+1}, the receiving of O_i at s_{i+1} must also happen before the generation of O_{i+1}. After receiving the remote operation O_i, the logical clock at s_{i+1} will become $max(LC_{s_{i+1}}, LC(O_i))$ according to **UR2** in Definition 2. On the other hand, the logical clock is incremented by 1 after generating O_{i+1} according to **UR1** in Definition 2. Therefore, $LC(O_{i+1})$ must be at least $max(LC_{s_{i+1}}, LC(O_i)) + 1$.

From the above, we know that $LC(O_i) \leq LC(O_{i+1})$, for all $i \in \{0, 1, ..., m-1\}$, so $LC(a) < LC(b)$. □

Definition 3 *total ordering relation "⇒"*
Given two group operations a and b, with timestamps $LC(a)$ and $LC(b)$, originated from sites i and j, respectively, then $a \Rightarrow b$, *iff*:

1. $LC(a) < LC(b)$, or
2. $LC(a) = LC(b)$ and $i < j$.

Since $a \rightarrow b$ implies $LC(a) < LC(b)$ (by Theorem 1), and $LC(a) < LC(b)$ implies $a \Rightarrow b$ (by Definition 3), $a \rightarrow b$ implies $a \Rightarrow b$ as well. Therefore, the total ordering relation "⇒" is consistent with the causal ordering relation "→."

In the above definition of the total ordering relation "⇒," when two group operations have the same timestamp, they are ordered according to their site identifiers, giving a higher priority to group operations originated from sites with smaller site identifiers. To be more fair, we could assign a dynamic identifier to each site by a function

$$f(sid, LC_{sid}) = (sid + LC_{sid}) mod N$$

where sid is the static site identifier, LC_{sid} is the logical clock at site sid, and N is the number of sites in the system. In this way, group operations can be ordered by their dynamic site identifiers when they possess the same timestamp. In the rest of the paper, we will use the static site identifiers to illustrate the protocols for the ease of explanation.

4 The basic protocol

In this section, the basic data structures and schemes in our protocol are defined.

4.1 Data structures

Definition 4 *Messages*
A message in the system is expressed as a tuple of <*type, sid, ts, op*>, where *type* represents the type of the message, *sid* is the sender's site identifier, *ts* is the timestamp, which is the sender's local logical clock value at the moment the message is sent out, and *op* is the group operation in case that the message encapsulates a group operation.

There are two types of messages in the system: one type is GOP message for encapsulating group operations, and the other type is ACK message for acknowledging local logical clock values. All messages in the system are multicast to all processes (except the sender). Given two GOP messages m_1 and m_2, $m_1 \Rightarrow m_2$ iff $m_1.op \Rightarrow m_2.op$.

Definition 5 *Pending Message Queue (PMQ)*
Each site maintains a queue of pending messages received but not yet executed. Messages in PMQ are sorted according to the total ordering relation "$\Rightarrow$", with $PMQ[i] \Rightarrow PMQ[j]$ if $i < j$. Only messages encapsulating group operations can be placed in PMQ.

Definition 6 *Logical Clock Vector (LCV)*
Each site k maintains a logical clock vector (LCV_k) of N elements, with one element ($LCV_k[k]$) being the local logical clock (LC_k) and others containing the estimated values of the logical clocks at other sites in the system. The LCV_k is maintained as follows:

1. Initially, $LCV_k[i] := 0$, $0 \leq i < N$.
2. After receiving a local GOP message, LCV_k is updated as follows: $LCV_k[k] := LCV_k[k] + 1$, according to **UR1** of Definition 2.

Journal of Computing and Information, Vol. 2, No. 1, 1996, pp. 219-237.

3. After receiving a remote message m of any type, LCV_k is updated as follows:

   - $LCV_k[m.sid] := m.ts$, and
   - $LCV_k[k] := max(LCV_k[k], m.ts)$, according to **UR2** of Definition 2.

4.2 Stability of messages

A group operation can be executed at a site only after it has become *stable*, which is defined below.

Definition 7 *Stable GOP messages*
Let m be a GOP message received by the SP at site k. m becomes *stable* at site k iff all GOP messages totally preceding m have been received at site k. Equivalently, m becomes *stable* at site k iff m totally precedes all forthcoming GOP messages to site k.

Definition 8 *The stability checking scheme*
Let pm be a pending message in PMQ at site k. pm is not stable for execution at site k until the following two conditions hold:

1. for $0 \leq i < pm.sid$, $pm.ts \leq LCV_k[i]$, and
2. for $pm.sid < i < N$, $pm.ts \leq LCV_k[i] + 1$.

Theorem 2 *The two conditions in Definition 8 are both necessary and sufficient for a pending message to become stable.*
Proof: Let pm be a pending message in PMQ at site k. According to **UR1** of Definition 2, we know that any forthcoming GOP message m from site $m.sid$ will carry a timestamp $m.ts \geq LCV_k[m.sid] + 1$.

- In case that $0 \leq m.sid < pm.sid$, we have $pm.ts \leq LCV_k[m.sid]$ from the first condition in Definition 8, which implies $pm.ts < m.ts$. Therefore, $pm \Rightarrow m$ according to Definition 3.
- In case that $pm.sid < m.sid < N$, we have $pm.ts \leq LCV_k[m.sid] + 1$ from the second condition in Definition 8, which implies $pm.ts \leq m.ts$. Moreover, since $pm.sid < m.sid$, we obtain $pm \Rightarrow m$ according to Definition 3.

Therefore, if the two conditions in Definition 8 hold, pm must totally precede any forthcoming GOP message from any site, so it must have become stable according to Definition 7.

Conversely, suppose pm has become stable at site k, then all GOP messages totally preceding pm must have been received at site k according to Definition 7. The following proves that the two conditions must be hold:

- If, by contradiction, there exists an i, $0 \leq i < pm.sid$, such that $LCV_k[i] < pm.ts$, then it is possible for a forthcoming GOP message m from site i to carry a timestamp $m.ts = LCV_k[i] + 1 \leq pm.ts$. Since $i < pm.sid$, we always have $m \Rightarrow pm.op$, which is in contradiction with the precondition.

- If, by contradiction, there exists an i, $pm.sid < i < N$, such that $LCV_k[i] + 1 < pm.ts$, then it is possible for a forthcoming GOP message m from site i to carry a timestamp $m.ts = LCV_k[i] + 1 < pm.ts$, so $m \Rightarrow pm.op$, which is also in contradiction with the precondition.

Therefore, if pm has become stable, then the two conditions must hold. □

4.3 Specification and analysis of the protocol

In the basic protocol, the SP multicasts an ACK message each time a remote GOP message is received. The top level control structure of the basic protocol (of the SP at any site) is specified in Protocol 1.

Protocol 1 The SP at site k starts by initializing itself: $LCV_k[i] := 0$, for all $i \in \{0, 1, ..., N-1\}$. Then it goes into an infinitive loop, with each iteration containing the following steps:

1. Get a message m from *IMQ*.
2. If the message is a local one, i.e., $m.sid = k$, then do the following:
   (a) Update the local logical clock according to **UR1** of Definition 2: $LCV_k[k] := LCV_k[k] + 1$.
   (b) Multicast a GOP message $m\_out =< GOP,\ k,\ LCV_k[k],\ m.op >$.
   (c) Insert the GOP message m_out into the proper position in *PMQ*.
3. If the message is from a remote site, i.e., $m.sid \neq k$, then do the following:
   (a) Record the incoming message timestamp: $LCV_k[m.sid] := m.ts$.
   (b) If the incoming message is of type GOP, then do the following:
      i. Update the local logical clock according to **UR2** of Definition 2: $LCV_k[k] := max(LCV_k[k], m.ts)$.
      ii. Multicast an ACK message $m\_out =< ACK,\ k,\ LCV_k[k],\ \phi >$.
      iii. Insert the incoming GOP message m into the proper position in *PMQ*.

4. Check to see whether there are any pending messages in PMQ which have become stable according to the two conditions in Theorem 2. If so, execute them in the order determined by the total ordering relationship "$\Rightarrow$".

The following theorem establishes the correctness of Protocol 1.

Theorem 3 *Under Protocol 1, the following is ensured:*

1. *Safety: Given two GOP messages m_1 and m_2, if $m_1 \Rightarrow m_2$, m_1 will be executed before m_2 at all sites.*

2. *Liveness: Given a GOP message m originated, the execution of the operation in m will never be delayed indefinitely.*

Proof:

1. Safety: Suppose, by contradiction, that m_2 becomes stable and executed at site k before site k has received m_1. Based on Protocol 1, when m_2 becomes stable at site k, site k must have received a message m from site $m_1.sid$ such that either $m_2.ts \leq m.ts (m.ts = LCV_k[m_1.sid])$ if $m_2.sid < m_1.sid$, or $m_2.ts \leq m.ts + 1$ if $m_1.sid < m_2.sid$ according to Theorem 2. From **UR1** of Definition 2 and the FIFO property of the underlying communication system, we have $m.ts + 1 \leq m_1.ts$. Then, it must be either that $m_2.ts \leq m_1.ts$ if $m_2.sid < m_1.sid$, or that $m_2.ts < m_1.ts$ if $m_1.sid < m_2.sid$, which implies that $m_2 \Rightarrow m_1$ – contradiction with the precondition that $m_1 \Rightarrow m_2$.

2. Liveness: After receiving a remote GOP message m at site k, $0 \leq k \neq m.sid < N$, this site will multicast an ACK message m_k, according to Protocol 1. Since m_k is multicast after receiving m at site k, it must be that $m.ts \leq m_k.ts$, according to **UR2** of Definition 2. Since the network is reliable, every site will eventually have $m.ts \leq LCV[k]$ for all $k \in \{0, 1, ..., N-1\}$. Therefore, m is ensured to become stable at all sites according to Theorem 2. □

Protocol 1 has the property of minimizing the *latency time* – the time expired from the moment a process receives a group operation until the moment it is sure that this operation is stable for execution. This is because:

- The total ordering of group operations is determined at the moment when operations are generated, without the need for a second phase of determining the total ordering as in two-phase protocols.

- The logical clocks' updating rules, especially **UR2** of Definition 2, are specially defined, so that the values of all sites' logical clocks are frequently the same (fully synchronized), and the stability checking scheme is able to *predict* the stability of a GOP message m without having to wait for the ACK messages from those sites with identifiers larger than $m.sid$.

Given a GOP message m arrived at site k, according to the second condition of the stability checking scheme, if $m.sid < i$, even when $m.ts = LCV_k[i] + 1$, it can be ensured that m has become stable with respect to site i. Therefore, it is possible for m to become stable upon its arrival without any delay, i.e., zero latency time. For one case, if, at the time of multicasting m, all sites' logical clocks had the same value and $m.sid$ is the smallest identifier[1], then m is ensured to be stable even at the moment of its multicasting. For another case, if, before the time of multicasting m, every other site i has multicasts a message m_i, with $m_i.ts \geq m.ts + 1$ if $i < m.sid$, or $m_i.ts \geq m.ts$ if $i > m.sid$, then m will become stable upon its arrival at remote sites without any delay. Generally, the latency time for a GOP message is in the range of $[0, d]$ inclusive, where d is the maximum message propagation delay between any pair of sites in the system. Whereas in two-phase protocols, the latency time is always at least $2d$ (see [3, 4]).

However, the basic protocol suffers from the large number of messages needed for stabilizing one group operation. Since for each received GOP message, a site multicast an ACK message in the basic protocol, the total number of messages involved in one group operation is N in multicasting networks, or $N \times (N-1)$ in point-to-point networks. For two-phase protocols, a total number of $(N+1)$ messages per group operation is needed for multicasting networks, or $3 \times (N-1)$ in point-to-point networks.

4.4 An example

An example is shown in Figure 2. In the figure, $G(sid, ts)$ represents a message of type GOP generated at site sid with a timestamp ts, $A(sid, ts)$ denotes a message of type ACK generated at site sid with a timestamp ts. **Site 0:**

- When $LCV = [1, 0, 0]$, a GOP message $G(0, 1)$ is multicast. $G(0, 1)$ is executed immediately at the local site since its timestamp 1 satisfies the stable conditions, i.e., $1 \leq LCV[i] + 1$, for $i \in \{1, 2\}$.

- After receiving $G(1, 1)$ from site 1, LCV becomes $[1, 1, 0]$. $G(1, 1)$ is executed immediately since its timestamp 1 satisfies the stable conditions, i.e.,

[1] If a dynamic identifier scheme is used as suggested in Section 3, then a different site could become the smallest at a different logical time.

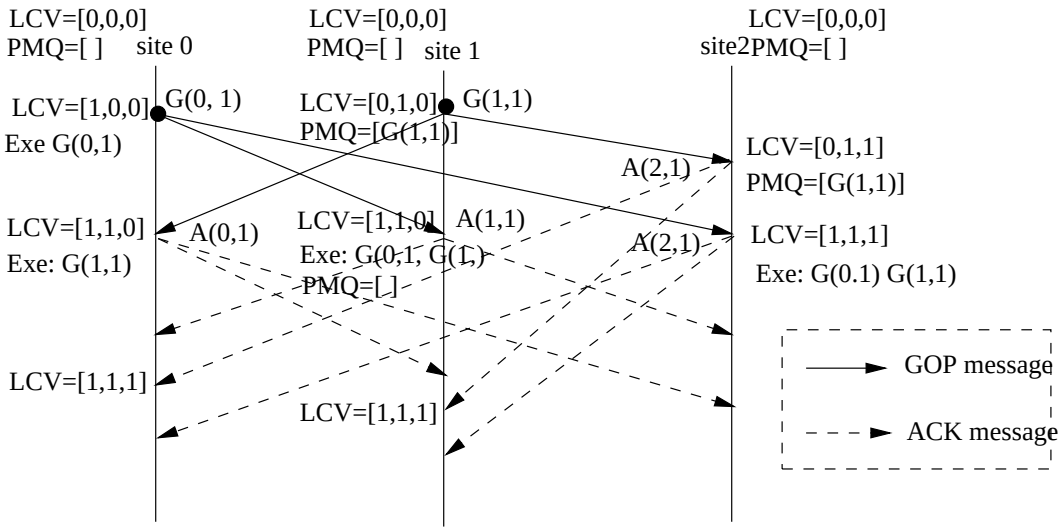

Figure 2: Example for the basic protocol

$1 \leq LCV[0](=1)$, and $1 \leq LCV[2](=0)+1$. Then an ACK message $A(0,1)$ is multicast.

- After receiving $A(1,1)$ from site 1, no change is made to LCV.
- After receiving $A(2,1)$ from site 2, LCV becomes $[1,1,1]$.
- After receiving another $A(2,1)$ from site 2, no change is made to LCV.

Site 1:

- When $LCV = [0,1,0]$, a GOP message $G(1,1)$ is multicast. Then, $G(1,1)$ is put into the PMQ, i.e., $PMQ = [G(1,1)]$, since its timestamp 1 does not satisfy the stable conditions.
- After receiving $G(0,1)$ from site 0, LCV becomes $[1,1,0]$. Then, $G(0,1)$ is executed immediately since its timestamp 1 satisfies the stable conditions, i.e., $1 \leq LCV[i]+1$, for $i \in \{1,2\}$. Next, $G(1,1)$ is executed. So, PMQ becomes empty. Finally, an ACK message $A(1,1)$ is multicast.
- After receiving $A(0,1)$ from site 0, no change is made to LCV.
- After receiving $A(2,1)$ from site 2, LCV becomes $[1,1,1]$.
- After receiving another $A(2,1)$ from site 2, no change is made to LCV.

Site 2:

- After receiving $G(1,1)$ from site 1, LCV becomes $[0,1,1]$. Then, $G(1,1)$ is put put into the PMQ, i.e., $PMQ = [G(1,1)]$, since its timestamp 1 does not satisfy the stable conditions, i.e., $1 \leq LCV[0](= 0)$. Finally, an ACK message $A(2,1)$ is multicast.

- After receiving $G(0,1)$ from site 1, LCV becomes $[1,1,1]$. Then, $G(0,1)$ is immediately executed, followed by the execution of $G(1,1)$ since they satisfied the stable conditions. Finally, an ACK message $A(2,1)$ is multicast.

- After receiving $A(1,1)$ from site 1, no change is made to LCV.

- After receiving $A(0,1)$ from site 0, no change is made to LCV.

As can be seen from this example, the latency time for GOP messages is indeed very short: zero for $G(0,1)$ at all sites, and zero for $G(1,1)$ at site 0, but less than d at sites 1 and 2. Moreover, it can also be observed that none of the four ACK messages in this example plays any role in stabilizing the two GOP messages, so all of them are in fact unnecessary and could be avoided. In the next section, an optimized acknowledging scheme will be discussed.

5 Optimizations

The basic protocol has the property of a minimum latency time, but suffers from the large number of ACK messages, especially in point-to-point communication networks. However, optimization is possible to reduce ACK messages. The basic idea is to multicast an ACK message only when necessary.

Definition 9 *Most Recently Multicast Timestamp (MRMT)*
Each site k maintains an integer, called the *Most Recently Multicast Timestamp (MRMT)*, which records the timestamp in the most recently multicast message of any type (GOP/ACK) from site k.

Definition 10 *The MRMT-based acknowledging scheme*
After receiving a GOP message m from a remote site $m.sid$ by site k, an ACK message needs to be multicast only when

1. $m.ts > (MRMT + 1)$, if $m.sid < k$, or
2. $m.ts > MRMT$, if $k < m.sid$.

Journal of Computing and Information, Vol. 2, No. 1, 1996, pp. 219-237.

It should be noted that in the above MRMT-based acknowledging scheme, in case that $m.sid < k$, even when $m.ts = (MRMT + 1)$, there is no need for site k to multicast an ACK message to stabilize m since it is sure that m has become stable with respect to site k according to the second condition in the stability checking scheme (Definition 8). Therefore, it is possible for m to become stable without a single ACK message multicast by any site in the system. For example, if, at the time of multicasting m, all sites' logical clocks had the same value and $m.sid$ is the smallest identifier, then m is ensured to be stable even at the moment of its multicasting, so zero ACK message is needed in the system to stabilize m. As another example, if, before m's arrival at site i, every other site i has multicast a message m_i, with $m_i.ts \geq m.ts + 1$ if $i < m.sid$, or $m_i.ts \geq m.ts$ if $i > m.sid$, then after receiving m, no site needs to multicast an additional ACK message to stabilize m. Generally, the number of ACK messages per GOP message is in the range of $[0, (N-1)^2]$ inclusive, where N is the number of sites in the system.

By integrating the MRMT-based acknowledging scheme into the basic protocol, we have the following optimized protocol.

Protocol 2 The SP at site k starts by initializing itself: $LCV_k[i] := 0$, for all $i \in \{0, 1, ..., N-1\}$, $MRMT := 0$. then it goes into an infinitive loop, with each iteration containing the following steps:

1. Get a message m from *IMQ*.
2. If the message is a local one, i.e., $m.sid = k$, then do the following:
   (a) Update the local logical clock according to **UR1** of Definition 2: $LCV_k[k] := LCV_k[k] + 1$.
   (b) Multicast a GOP message $m\_out =< GOP,\ k,\ LCV_k[k],\ m.op >$.
   (c) Update *MRMT*: $MRMT := LCV_k[k]$.
   (d) Insert the GOP message m_out into the proper position in *PMQ*.
3. If the message is from a remote site, i.e., $m.sid \neq k$, then do the following:
   (a) Record the incoming message timestamp in LCV: $LCV_k[m.sid] := m.ts$.
   (b) If the incoming message is of the type GOP, then do the following:
      i. Update the local logical clock according to **UR2** of Definition 2: $LCV_k[k] := max(LCV_k[k], m.ts)$.
      ii. Check to see if an ACK message is needed according to Definition 9. If so
         - Multicast an ACK message $m\_out =< ACK,\ k,\ LCV_k[k],\ \phi >$.

Journal of Computing and Information, Vol. 2, No. 1, 1996, pp. 219-237.

- Update $MRMT$: $MRMT := LCV_k[k]$.

iii. Insert the incoming GOP message m into the proper position in PMQ.

4. Check to see whether there are any pending messages in PMQ which have become stable according to the two conditions in Theorem 2. If so, execute them in the order determined by the total ordering relationship "$\Rightarrow$".

Theorem 4 *Under Protocol 2, the following is ensured:*

1. *Safety: Given two GOP messages m_1 and m_2, if $m_1 \Rightarrow m_2$, m_1 will be executed before m_2 at all sites.*

2. *Liveness: Given a GOP message m, the execution of the operation in m will never be delayed indefinitely.*

Proof: The safety proof is the same as that for Theorem 3. The following proves only liveness. After receiving a remote GOP message m by site k, $0 \leq k \neq m.sid < N$:

- if $m.sid < k$, we have either $m.ts \leq MRMT + 1$ when no ACK message need be multicast, or $m.ts = MRMT$ after an ACK message is multicast otherwise, according to Definition 9. In any case, m will be stable with respect to site k according to the second condition of Definition 8.

- if $m.sid > k$, we have either $m.ts \leq MRMT$ when no ACK message needs be multicast, or $m.ts = MRMT$ after an ACK message is multicast otherwise, according to Definition 9. In any case, m will be stable with respect to site k according to the first condition of Definition 8.

In summary, m is ensured to become stable with respect to site k, for all $k \in \{0, 1, ..., N-1\}$. □

It is worth pointing out that Protocol 2 is self-adaptive to the traffic load and pattern of GOP messages in the system:

- As the GOP messages traffic load increases and becomes evenly distributed among all sites, the average latency time and the number of ACK messages per GOP message decreases and may approach zero at the best.

- As the GOP messages traffic load decreases and becomes uneven, the average latency time and the number of ACK messages per GOP message increases: the latency time may approach the maximum d, and the number of ACK messages per GOP message may approach the maximum $(N-1)$ in multicasting networks or $(N-1)^2$ in point-to-point networks.

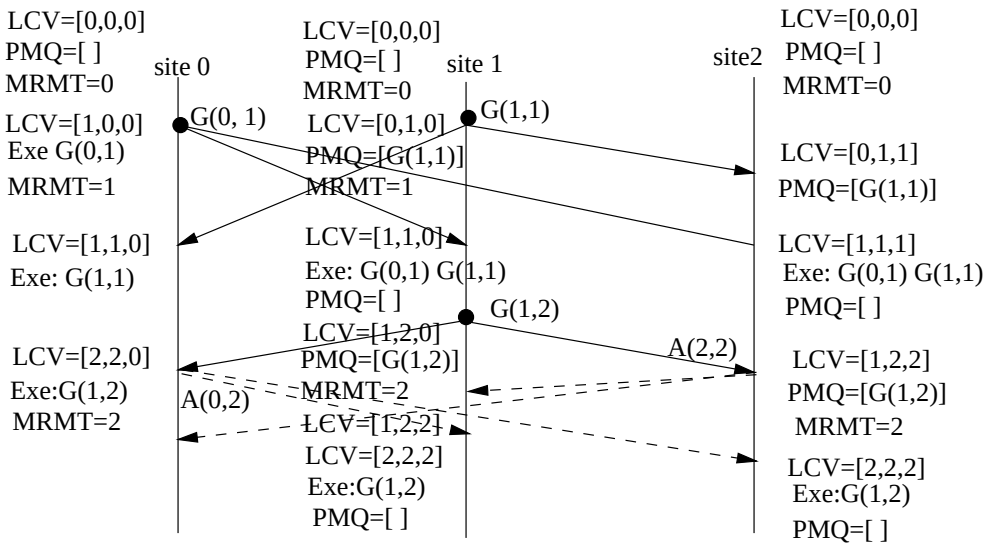

Figure 3: An example for the optimized protocol

Let us re-examine the scenario in the example in Section 4.4, with each site augmented with the MRMT-based acknowledging scheme. As shown in Figure 3, when sites 0 and 1 are multicasting GOP messages $G(0,1)$ and $G(1,1)$ at the same (logical) time (a case where the GOP message traffic load is heavy), no single ACK message is needed for stabilizing them according to the MRMT-based acknowledging scheme and the latency time for $G(0,1)$ is zero. Later on, when only site 1 multicasts a GOP message $G(1,2)$ (a case where the GOP message traffic load is light), both site 0 and site 2 have to multicast ACK messages $A(0,2)$ and $A(2,2)$, respectively, resulting in 2×2 point-to-point messages in the worst case and the latency time is d. Furthermore, it can be seen that after the stabilization of each GOP message at all sites, the logical clocks values at all sites are the same, and at the end of the scenario, the values in LCV, $MRMT$, and PMQ are the same for all sites.

6 Conclusions

In this paper, a single-phase protocol for the total and causal ordering of group operations in distributed systems has been proposed and discussed. The main features of the proposed protocol are summarized below:

- It is based on the use of a system of specially defined logical clocks to determine both total and causal ordering among operations in distributed systems.

- Each cooperative process makes decisions on the total order and stability of group operations based on its local logical clock and the knowledge of other processes' logical clocks values obtained from incoming messages, without the use of a centralized coordinator, so the protocol is fully distributed and does not have a single point of failure or performance bottleneck.

- The total ordering of group operations is determined at the moment when operations are generated, without the need for a second phase of determining the total ordering as in other two-phase protocols. Therefore, the latency time can be kept in the range of $[0, d]$: zero at the best, or d propagation delay at the worst, instead of at least $2d$ propagation delay in two-phase protocols.

- In contrast to the arbitrary total ordering imposed by two-phase protocols, the total ordering imposed by the proposed protocol is consistent with the causal ordering among group operations, which is needed by many applications in distributed systems.

- With an optimized acknowledging scheme integrated, the number of acknowledgment (overhead) messages involved in each group operation can be kept in the range of $[0, N-1]$ in multicasting networks, or $[0, (N-1)^2]$ in point-to-point networks.

- The proposed protocol has an interesting property of being adaptive to the traffic load and pattern of GOP messages in the system: As the GOP messages traffic load increases and becomes evenly distributed among all sites, the average latency time and the number of ACK messages per GOP message decreases and may approach zero at the best. However, as the group operation traffic load decreases and becomes uneven, the average latency time and the number of ACK messages per GOP message increases: the latency time may approach the maximum d, and the number of ACK messages per GOP message may approach the maximum $(N-1)$ in multicasting networks or $(N-1)^2$ in point-to-point networks.

The proposed protocol has been tested in a UNIX-based simulation system, Simulation results have shown that the protocol is correct. More theoretical and experimental analysis is still needed to gain a better understanding of the statistic properties of the proposed protocol, such as the mean latency time per group operation, the mean number of acknowledgment messages per group operation, etc. Moreover, we are designing and experimenting another optimized version of the proposed protocol, which use a point-to-point acknowledging scheme, instead of the multicasting acknowledging scheme, to reduce the average number of acknowledgment messages in point-to-point networks. More results will be reported in future papers.

Journal of Computing and Information, Vol. 2, No. 1, 1996, pp. 219-237.

Acknowledgements

The work reported in this paper is partially supported by an ARC (Australian Research Council) Small grant.

References

[1] K. Birman, A. Schiper, and P. Stephenson: "Lightweight causal and atomic group multicast," *ACM Trans. on Comp. Sys.,* Vol. 9, No.3, pp. 272-314, Aug. 1991.

[2] K. Birman:"The process group approach to reliable distributed computing," *Communication of the ACM* Vol. 36, No.12, pp. 37-53, Dec. 1993.

[3] G. Coulouris, J. Dollimore, and T. Kindberg:*Distributed Systems: Concepts and Design,* 2nd ed. Addision-Wesley, ISBN 0-201-62433-8, 1994.

[4] M. Dasser:"TOMP: a total ordering multicast protocol," *ACM Operating System Review*, Vol 26, No. 1, pp.32-40, 1992.

[5] C. A. Ellis and S. J. Gibbs:"Concurrency control in groupware systems," In *Proc. of ACM SIGMOD Conference on Management of Data*, pages 399–407, 1989.

[6] H. Garcia-Molina, and A. Spauster:"Ordered and reliable multicast communication," *ACM Trans. on Comp. Sys.* Vol. 9, No.3, pp. 242-271, Aug. 1991.

[7] S. Greenberg, D. Marwood:"Real time groupware as a distributed system: concurrency control and its effect on the interface," In *Proc. of ACM Conference on Computer Supported Cooperative Work,* pp. 207–217, Nov. 1994.

[8] L. Lamport:"Time, clocks, and the ordering of events in a distributed system," *CACM 21(7)*, pp.558-565, 1978.

[9] M.F. Kaashoek, A. Tannenbaum:"Fault tolerance using group communication," *ACM Operating System Review*, Vol 25, No. 2, pp.71-74, April 1992.

[10] P. Keleher:*Lazy release consistency for distributed shared memory,* Ph.D Thesis, Rice University, Jan. 1995.

[11] S. Mullender:*Distributed Systems,* 2nd ed. Addision-Wesley, ISBN 0-201-62427-3, 1993.

[12] M. Raynal and M. Singhal:"Logical time: capturing causality in distributed systems," *IEEE Computer*, pp. 49-56, Feb. 1996.

[13] C. Sun, Y. Zhang, and Y. Yang:"Distributed synchronization of group operations in cooperative editing environments," In *Proceedings of The Second International Conference on Concurrent Engineering,* Washington DC., pp.279 - 290, 1995.

[14] C. Sun, Y.Yang, Y. Zhang, and D. Chen: "A consistency model and supporting schemes for real-time cooperative editing systems," In *Proc. of the 19th Australian Computer Science Conference,* pp. 582-591, Melbourne, Jan 1996.

[15] A.S. Tanenbaum:*Distributed Operating Systems,* Prentice-Hall International, 1995.