The Implementation and Performance Analysis of a Total Order Delivery Protocol for Group Communication*

Raimundo José de Araújo Macêdo
Universidade Federal da Bahia - LaSiDi

Santosh K. Shrivastava
Department of Computing Science
University of Newcastle upon Tyne, UK

ABSTRACT

It is widely accepted that Group Communication is a powerful abstraction that can be used whenever a group of distributed processes cooperate for the execution of a given task such as load-sharing or fault-tolerance. We have developed a causality preserving total order protocol for group communication, named Newtop. In developing Newtop, we have considred system settings where message transmission and processing times cannot be accurately estimated (asynchronous modeling). Our protocol offers a simple method for dealing with overlapping process groups and has low message space overhead (the protocol related information contained in a multicast message is small). We have implemented Newtop over a set of networked UNIX workstations and we present in this paper some relevant aspects of this implementation as well as performance data collected from a serie of experiments.

SUMÁRIO

É largamente aceito que "Group Communication" é uma abstração poderosa que pode ser usada toda vez que grupos de processos distribuídos cooperam para a execução de uma dada tarefa tal como "load-sharing" ou tolerância a falha. Nos desenvolvemos um protocolo de ordem total que preserva a origem causal das mensagens, chamado Newtop. No desenvolvimento de Newtop, consideramos sistemas onde tempos de transmissão e processamento de mensagens não podem ser estimados de forma acurada (modelagem assíncrona). Nosso protocolo oferece um método simples para tratar grupos que se sobrepõem e tem "overhead" de mensagens baixo (a quantidade de informações contidas numa mensagem de multicast e relacionadas com o protocolo é pequena). Nos implementamos Newtop numa rede de máquinas UNIX e apresentamos neste artigo aspectos dessa implementação, bem como dados de desempenho coletados a partir de uma série de experimentos.

---

* This work has been partially supported by CNPq, grant number 200811/89
1 Laboratório de Sistemas Distribuídos (LaSiDi). Prédio do CPD/UFBa, Campus de Ondina, Salvador-Ba, CEP: 40.170-110 email: macedo@ufba.br fax: (071) 247 9907
1. Introduction

It is widely accepted that group communication (or multicast) is a powerful abstraction that can be used whenever groups of distributed processes cooperate for the execution of a given task such as load-sharing and fault-tolerance [Bir91, ADKM92, BSS91, MMA90, PNS89, VRV93]. We have developed a causality preserving total order protocol for group communication, named Newtop. Newtop offers a simple method for dealing with overlapping process groups and has low message space overhead (the protocol related information contained in a multicast message is small). Newtop does not make any of the following assumptions, which makes it distinct from previous works: message transmission latency is bounded and known, the underlying communication network supports broadcasting of messages, there is always a coordinator process that sequences messages, or functioning processes can always communicate with each other (i.e. there is no network partitioning). We have carefully discussed all these aspects of Newtop elsewhere [Mac94, EMS95, MES93a, MES93b]. In this paper we present aspects of the implementation of Newtop over a set of networked UNIX machines (Sun sparce stations) and also comment on performance results obtained from a series of experiments.

Section 2 describes the system model assumed. Section 3 presents an overview of the Causal Blocks model (our message ordering mechanism) and section 4 presents how Causal Blocks are constructed for total order. Section 5 presents aspects of the implementation of Newtop. Section 6 shows performance data collected from a serie of experiments. Finally, section 7 concludes the paper.

2. The System Model and Failure Assumptions

We assume a set of sequential processes which are distributed possibly on distinct processors or sites and communicate with each other only by exchanging messages. We assume a transport layer that provides lossless, uncorrupted, and sequenced message delivery between any pair of functioning processes (FIFO assumption). Once a message has been sent by the transport layer, it can take an potentially unbounded amount of time to be received at the destination (i.e. we assume an asynchronous system). For simplicity and space reasons, in this paper we assume a failure-free environment (i.e., processes do not crash or misbehave). In [Mac94] we present mechanisms for coping with dynamic changes in the membership (due to failures such as process crashes and network partitioning) to be used with the total order protocol we will describe. Finally, we assume that processes can belong to multiple (overlapping) groups.

---

2 Newtop stands for Newcastle Total Order Protocol.
3 This can be realised by introducing sequential numbers to messages, with positive acknowledgement and retransmission of missing messages (TCP/IP provides such a functionality).
A process execution consists of a sequence of events, each event corresponding to the execution of an action by a process. Within a given process, events are naturally ordered by the sequence they happen. However, ordering events from distinct processes is not possible unless they execute communication actions among themselves. An example of a communication action executed by a process, say $p$, can be to send a multicast message, say $m$, to a group that is recorded in $m$ as $m.g$. The corresponding event will be denoted as $send_p(m)$. Similarly, we denote the event of a process $q$, belonging to the group $m.g$, receiving $m$ as $receive_q(m)$. Then, we can define a 'happened before' relation, denoted as $\rightarrow$, on $send$ and $receive$ events in a given set of system events. Thus, when $a$, $b$, and $c$ are three distinct events in a subset of system events, each referring to either $send$ or $receive$ events,

(i) if $a$ comes before $b$ in the same process, then $a \rightarrow b$;
(ii) if $a$ is a $send_p(m)$ and $b$ is $receive_q(m)$, then $a \rightarrow b$; and
(iii) if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

A message $m$ will be said to have potentially caused $m'$, if $send(m) \rightarrow send(m')$, and distinct messages $m$ and $m'$ will be said to be concurrent if neither $send(m) \rightarrow send(m')$ nor $send(m') \rightarrow send(m)$ is true. Hence, the relation $\rightarrow'$ establishes a partial order of events in a distributed system [Lamport'78]. For notational simplicity, when $m$ and $m'$ are two distinct multicasts, $m \rightarrow m'$ will denote that $send(m) \rightarrow send(m')$.

When some specific delivery order is required, received messages may have to be retained for later delivery until certain ordering conditions are satisfied (otherwise, the delivery order stated may be violated). Thus, we need to define $delivery_p(m)$ as the event of delivering message $m$ to process $p$.

Newtop delivers all messages in the same order and that delivery respects causality. Formally, delivery of messages by Newtop satisfies the following property.

**total order delivery**\(^4\): for any $p_i$ and $p_j$ $\in m.g \cap m'.g$:

- if $delivery_i(m')$, $delivery_j(m)$, $delivery_j(m')$, and $delivery_j(m')$ occur, then
  - (i) $delivery_i(m) \rightarrow delivery_j(m') \Leftrightarrow delivery_j(m) \rightarrow delivery_j(m')$ and
  - (ii) $m \rightarrow m' \Rightarrow delivery_j(m) \rightarrow delivery_j(m')$.

(i) guarantees that messages are delivered in the same global order and (ii) guarantees that delivery will respect causality.

\(^4\)Recall that we are assuming in this paper a failure-free environment. In the presence of failures, total order delivery can not be guaranteed deterministically in an asynchronous system [FLP85], unless mechanisms such as failures detectors [Mac94, EMS95] are considered.
3. An Overview of the Causal Blocks Model

In what follows we briefly describe our basic timestamping mechanism upon which the total order protocol is developed. We have named this mechanism the Causal Blocks model which is a framework for developing group communication protocols with different ordering and reliability requirements [MES93a, Mac94].

Consider the existence of a group, $g = \{p_1, p_2, \ldots, p_n\}$. Each process $p_i$ maintains a logical clock called the Block Counter and denoted as $BC_i$. $BC_i$ is an integer variable and its value increases monotonically. Transmitted messages are timestamped with Block Counters, and, unlike Lamport’s Logical Clock that is advanced on send/receive events, we have decided to advance Block Counters on send/delivery events, since a transmitted message can only be causally dependent on previously delivered ones.

We can state the three following properties possessed by block-numbers of multicast messages. For notational simplicity, we denote $send_{pi}$ as simply $send_i$.

- $pr1: \text{send}_i(m) \rightarrow \text{send}_i(m') \Rightarrow m.b < m'.b$.
- $pr2: \text{for any } m, P_j \text{. } m.g: \text{deliver}_j(m) \rightarrow \text{send}_j(m') \Rightarrow m.b < m'.b$; and,
- $pr3: \text{for all } m', m'': m'.b = m''.b \Rightarrow m' \text{ and } m'' \text{ are concurrent.}$

The properties $pr1$ and $pr2$ imply that for any distinct $m$, $m'$: $\text{send}(m) \rightarrow \text{send}(m') \Rightarrow m.b < m'.b$. The property $pr3$ states that distinct messages multicast with the same block-number are necessarily concurrent and these messages must have been multicast by distinct processes, as two send events cannot occur in a given process with the same value of BC.

A given process, using $pr3$, constructs Causal Blocks to represent concurrent messages it sent/received with the same block-number. Causal Blocks maintained by a process are arranged in the increasing order of the message block-number they represent, giving rise to a matrix which is called the Block Matrix and denoted as BM. Thus, BM[B] will represent the Causal Block for message block-number B.

4. Construction of Causal Blocks for Total Order Delivery

To start with, consider the existence of a single group $g_X = \{p_1, p_2, \ldots, p_n\}$. It is assumed that every member process $p_i$, $1 \leq i \leq n$, knows the whole membership, and that, when $g_X$ is created, the $BC_i$ of every $p_i$ is initialized to zero. The Block-Matrix of $p_i$ for $g_X$ will be denoted as $BM_{X,i}$. Thus, a message $m$ sent or received by $p_i$, will be represented in the row $BM_{X,i}[m.b]$.

Before a process $p_i$ multicasts a message $m$, it advances $BC_i$ by $\alpha$, where $\alpha$ a non-zero positive integer. The contents of the incremented $BC_i$ is assigned to $m$ as its block-number in the field $m.b$. As $BC_i$ is advanced by $\alpha$, for every multicast, consecutively sent messages will have increasing block-numbers.
CA1 (Counter advances during send\(_i(m)\)):: Before \(p_i\) multicasts \(m\), it increments BC\(_i\) by \(\alpha\), and assigns the incremented value to m.b.

A Causal Block BM\(_{X,i}[\beta]\), \(\beta \geq 1\), will be said to be complete, when \(p_i\) can no longer send or receive a message, \(m\), m.b = \(\beta\) and m.g = G\(_X\). When a process identifies complete Causal Blocks, it can identify the set of all concurrent messages with a given block-number (or the absence of it); also, the set of all messages that may be causally related to a given message can be identified; for instance, if a message m is represented in BM\(_{X,i}[\beta]\) that is complete, then \(\{m_0 \mid m_0 \to m\} \subseteq \{\text{messages represented by every BM}\_{X,i}[\beta_0]\}, \beta_0 < \beta\}.\) Hence, Newtop obeys the following safe conditions for total order message delivery:

safe1: after a Causal Block is complete, a fixed pre-determined order for delivery is assigned to messages represented there; and,

safe2: a message \(m\), m.b = \(\beta\), is delivered only after the delivery of all messages with block-numbers less than \(\beta\) and the messages with block-number \(\beta\) that were ordered before \(m\).

The above conditions state the safety property of Newtop for a single non-overlapping group. Its correctness follow straight from the properties pr\(_1\), pr\(_2\), and pr\(_3\). Every received message will eventually be delivered by a process (liveness property), if it can be guaranteed that every Causal Block maintained by the process will eventually complete. Now we introduce a simple mechanism called the time-silence mechanism, to enable a process to remain lively by sending null messages, during those periods it is not generating computational messages to complete created Causal Blocks.

**Time-silence Mechanism**

The time-silence mechanism of a process \(p_i\), timesilence\(_i\), works as follows: whenever \(p_i\) creates a new Causal Block as a result of receiving a multicast message with block-number \(\beta\), a timeout for some predetermined period (called local-time-silence) is set for that Causal Block, BM\(_{X,i}[\beta]\), if \(p_i\) has not already multicasted a message with block-number larger than or equal to \(\beta\). This timeout period indicates the duration within which \(p_i\) is expected to multicast a message with block-number \(\beta\) or larger - thus contributing to the completion of BM\(_{X,i}[\beta]\) at all member processes of g\(_X\) (including itself). Note that \(p_i\) multicasting a message with block-number \(\beta\) will contribute to the completion of blocks BM\(_{X,j}[\beta_0]\), for \(\beta_0 \leq \beta\) and for all \(j, 1 \leq j \leq n\). So, if \(p_i\) multicasts a message with block-number \(\beta'\), \(\beta' > \beta\), before the expiration of the timeout set for BM\(_{X,i}[\beta]\), then the timeout set for any and every BM\(_{X,j}[\beta_0]\), for \(\beta_0 \leq \beta\), are cancelled. If, on the other hand, the timeout for BM\(_{X,i}[\beta]\) expires, then timesilence\(_i\) will force \(p_i\) to multicast a special null message. This null message is multicast with the largest block-number that \(p_i\) has "seen" so far, i.e. with a block-number \(\beta'' = \max\{m'.b \mid \text{receive}\{m'\}\} \geq \beta\), so that, this multicast will contribute to the completion of all Causal Blocks BM\(_{X,j}[\beta_0]\), for \(\beta_0 \leq \beta''\) and for all \(j, 1 \leq j \leq n\).
null message will also cancel the timeouts set for any BMX,i[β0], for β0 ≤ β". With the
introduction of time-silence, Block-Counters of processes advance not only by sending
application related messages (CA1), but also when null messages are sent. This possibility is
stated as CA2.

CA2 (Counter Advances due to sending of a null message by timesilence) before pti multicast a
null message m, it sets m.b = max{m'.b | receive(m') has occurred}, and BCi = m.b.

Note that for a multicast, block-numbers are computed using different algorithms for null and
non-null messages. Despite this difference, p1 (sendi(m) → sendi(m') ⇒ m.b < m'.b) is still
valid. That is, successive multicasts from pti will have increasing block-numbers (see Mac94 for a
proof).

Overlapping Groups

Consider two groups g1 = {p1, p2, p3, p4} and g2 = {p3, p4, p5, p6}. The processes p3 and
p4 are members of both g1 and g2. Suppose that p1 multicasts m1 in g1 and that m1 is delivered
to p3 which subsequently multicasts m2 in g2. P4, being a member of g1 and g2, will receive m1
and m2 (not necessarily in that order) and must be able to deduce that deliver3(m1) →
send3(m2), i.e. m1 → m2. Newton treats group overlapping in the following way: processes
which are members of more than one group should maintain a single Block Counter which
should be advanced subject to CA1, no matter which one of the groups a sent message belongs to.
So, if the multigroup member process p3 maintains a single BC for both g1 and g2, then the
block-number given to m2 will be m2.b = m1.b, as send1(m1) → deliver3(m1) → send3(m2).

More precisely, suppose that a process pti is a member of more than one group. Let Gti be the
set of groups pti belongs to: Gti = {gx | pti ∈ Gx}. So, |Gti| > 1. We assume that pti maintains, as
before, a single BCi which will be updated as mentioned in CA1 - irrespective of the group m
was multicast. It will, however, maintain a distinct BMX,i for each group gx in Gti - representing
messages sent or received with m.g = gx. Although pti advances its BCi by only one every time it
 multicasts a non-null message, it can appear to advance its BCi by a randomly chosen α, α > 0,
between successive multicasts in a given group, if it has performed multicasts to other groups in
between those two multicasts. Despite this, the following Causal Block property will still be valid
in a given BMX,i of pti: if two distinct non-null messages m and m' are represented respectively in
BMX,i[β] and BMX,i[β'], and m → m', then the row BMX,i[β] will precede the row BMX,i[β'].

The time-silence mechanism of pti will operate independently for each distinct gx: the timeout
set for BMX,i[β] will not be cancelled when pti multicasts a μ with μ.b ≥ β and gx ≠ μ.g; similarly,
if the timeout set for BMX,i[β] expires, a null message will be multicast only in gx. Since
the messages multicast by a given process in a given group have increasing block-numbers
and are received in FIFO order, a multi-group member process pti can identify the complete
blocks in each of its \( BM_{X,i} \). The message delivery condition, \( safe_1 \), must be modified to take account of the fact that a process belongs to more than one group. The new condition for a process, \( p_i \), is:

\( safe_1 \): after \( BM_{X,i}[\beta] \) is complete for every \( g_x \in G_i \), a fixed pre-determined order for delivery is assigned to the messages with block-number \( \beta \).

5. The Implementation of Newtop

We have implemented and tested Newtop over a set of networked UNIX workstations (sun sparcs). The architecture of the implementation consists basically of concurrent processes (receiver, transmitter, deliver, time-silence, and membership) communicating through message queue UNIX IPCs\(^5\). The transport multicast layer has been implemented through multiple point-to-point TCP/IP sockets\(^6\) and daemon processes to send/receive multicasts in a given group. Newtop communicates with the transport layer through two message queues, the sending and the receiving message queues.

The BM Matrix has been efficiently implemented in a two level data structure (figure 1). The first level of BM is a hashing table addressed by block-numbers. Each entry of the hashing table contains a Causal Block number and the pointers to the messages associated with that Causal Block. These messages are kept in a dynamically allocated message pool (the second level). The size of the hashing table, say \( N \), is known when the group is created. When a non-null message \( m \) is to be entered in BM (or retrieved from), the hashing function \( h \), where \( h(m.b) = m.b \mod N \), will produce the address in the hashing table where the pointers to the messages of block \( m.b \) can be found. Collision of addresses (block \( b \) and block \( b + N \), for instance) never happens due to the flow control provided [MES95].

---

\(^5\) Message queues are IPC mechanisms of UNIX System that appeared for the first time in the System V. These queues are created and manipulated by system calls and reside in the system kernel [Ste90]

\(^6\) TCP/IP channels (UNIX stream sockets) are reliable and deliver messages in FIFO order [Ste90].
A process \( p_i \) has - in addition to \( BC_i \) and a \( BM_{X,i} \) for each \( g_X \) in \( G_i \) - a vector, called the Completed Blocks Vector and denoted as \( CBV_i \), which is of size \( |G_i| \) and is maintained in the following manner: for each \( g_X \) in \( G_i \), \( CBV_i[x] \) indicates the largest number of complete blocks in \( BM_{X,i} \). Process \( p_i \) also maintains the value of \( \min\{CBV_i[x] \mid g_X \in G_i \} \) in a variable \( B_{\text{min}} \). \( CBV_i \) is initialized to \([0,0,...,0]\), \( B_{\text{min}} \) to 0, and every time a message is added to any \( BM_{X,i} \) both are updated if possible. Whenever the value of \( B_{\text{min}} \) changes, all sent and received, but not delivered messages with block-number less than or equal to (the new value of) \( B_{\text{min}} \) are delivered according to the message delivery conditions safe1' and safe2.

For the sake of fault-tolerance, it is necessary to ensure that a given process can always retrieve a missing message from a functioning process. Thus, a message \( m \) stored in \( BM \) cannot be discarded until it is known that all group members have received \( m \) (i.e., \( m \) is stable). To compute stable blocks (blocks which contain stable messages), a process maintains a vector called the Stability Vector (VS) with one entry per process in the group. \( SV[i,j] \) represents what process \( p_i \) knows about the last complete block number at \( p_j \). When a process detects a block stable (i.e with block number \( b \leq \) minimum value in SV), it frees the space occupied by block \( b \) in \( BM \).

6. Performance Results

Due to the unpredictability of transmission delays in asynchronous systems, message delivery delay of a total order multicast protocol for such systems is measured in terms of the number of extra messages transmitted so as to deliver a multicast. In the case of Newtop, these messages are null messages generated by the time-silence mechanism in order to complete "old" incomplete blocks. Another relevant performance measure is the message space overhead, and in the case of our protocol, this overhead is constant and very small (basically, the message block-number).

In Newtop, messages are delivered on the basis of block completion. This means that messages to complete a given block have to be sent by all group members before that block can be delivered. So, the activity (rate of message transmission) of the group members will determine how fast blocks get complete, and consequently, messages delivered. The Time-Silence mechanism guarantees that messages are always delivered despite the inactivity of some group members but on the expense of null messages transmitted.

To analyze the behavior of our protocol we have carried out experiments for the two extreme case scenarios in terms of processes activity. That is, the worst case scenario, when only one process sends messages to the group whereas the others remain silent (1-active experiment), and the best case scenario, when all processes are actively sending messages to the group (all-active experiment). By analyzing the behavior of the protocol on these conditions, one can get a good insight of its performance in the presence of distinct scenarios.
Experiments have been run for different group configurations, varying parameters such as the inter-message transmission time period and the time-silence timeout. The delivery delay overhead for our protocol, is the time elapsed between the receipt of a message m by the receiver process and the delivery of that message by the deliver process. When m is received from the transport multicast layer by the receiver process, it is timestamped with the current wall clock value and placed in BM. Then, when it is delivered (put the in application message queue), the current wall clock value is taken again and its value minus m's timestamp is taken as the delivery delay overhead of m. For a given experiment, the average delay overhead of messages delivered by a group member is the sum of all message delay overheads divided by the number of messages delivered. Finally, the average delay overhead showed in the figures is the sum of the average delay overheads of all group members divided by the number of members.

Figures 2, 3, and 4, show data collected for the 1-active experiment when inter-message transmission time period was set to 200 msecs. The graphs in figures 2, 3, and 4 show maximum unstable blocks, average number of null messages transmitted per inactive process (the receivers), and average delay overhead, respectively, for time-silence timeouts from 50 to 1050 msecs. We have run the experiments for different group configurations, from 2 to 6 group members. In each of the graphs there is a curve representing the data for a given group configuration (the group size appears on the legend). Group processes were spread over three workstations. Notice that despite the number of group members, increasing the time-silence timeout makes the number of unstable blocks to grow, increases the delivery delay, and decreases the number of null messages transmitted.

![Max. unstable blocks](image)

Figure 2. Maximum number of unstable blocks.
The next three figures (5, 6, and 7) show data collected from the all-active experiment when the inter-message transmission time period was set to 500 msecs for time-silences values from 450 down to 200 msecs. We have run the experiment for different group sizes, from 2 to 6 group members. Figures 5, 6, and 7 show the maximum number of unstable blocks, the average delay overhead, and the number of null messages transmitted by the group, respectively, during the experiment. Notice that the variation on time-silence timeouts did not cause a great impact on the number of unstable blocks and average delay overhead (although the average delay overhead slightly decreased for smaller time-silence timeouts). Similarly, the number of null messages transmitted increased for shorter time-silence timeouts. Observe that although all application processes for this experiment transmit messages in the same rate, they work asynchronously (all processes run in a multi-task and multi-user environment). Therefore, messages to complete a given block will not arrive (or be transmitted) at the same physical time, causing null messages be transmitted by the time-silence mechanism.
We have also collected data from the 1-active experiment when time-silence was fixed to 100 msecs, and due to space limitations, the corresponding graphs have been omitted. In that
experiment, we have varied the inter-message transmission time period from 400 msecs down to 6 msecs. We observed that for smaller inter-message transmission periods, the number of unstable blocks increases and the number of null messages decreases. For the same variation, the average delay overhead increases. This increase of the average delay overhead is due to the fact that a larger number of unstable blocks imposes extra delay overhead on messages waiting for block completion. Thus, adding to the overall average message delivery delay overhead. Finally, we noticed that despite the increase on average delay, the numbers of messages delivered per second (throughput) increased for smaller inter-message delivery time. The explanation for this is that when the inter-message message transmission period in small (so, transmission rate is high), several blocks are delivered at once. Thus, as long as an application can afford buffers, the existence of incomplete blocks will not affect the overall system performance.

7. Conclusions

We have presented a symmetric\(^7\) (i.e., there is no sequencer or coordinator process) total order protocol for groups that may be overlapping. Besides being simple to handle, even in the presence of overlapping groups, the approach presented has the main advantage of the constant and low message space overhead [Mac94, EMS95, MES93a, MES93b].

In this paper, we have described the implementation of the Newtop protocol. The implementation described has been tested over a set of networked UNIX sparc stations. We have run experiments to evaluate the performance of Newtop under varied group configurations, transmission rates, and time-silence timeouts, when processes crashes were not considered. The value set for the time-silence timeout is determinant on the number of null messages transmitted. When only one group member is transmitting messages, choosing shorter time-silence timeouts will on one hand cause a larger number of null messages be transmitted, but on the other hand, it will reduce the message delay overhead and the number of unstable blocks produced. When all group members are actively transmitting messages at the same rate, varying the time-silence timeout does not cause a great impact on the number of unstable blocks and the average delay overhead improved only slightly for shorter time-silence timeouts. So, choosing the appropriate time-silence value is a key point for tuning the behaviour of Newtop to specific application and system requirements such as local buffers (maximum number of unstable blocks), delay overhead, and consumption of transmission bandwidth (extra null messages transmitted).

References


\(^7\) In [EMS95] we have presented a version of Newtop that can work either symmetrically or asymmetrically.