S_b : A Semantics preserving distributed message ordering protocol for group communication

Chaitanya Krishna Bhavanasi and Sridhar Iyer
KR School of Information Technology,
Indian Institute of Technology - Bombay
(email:chaitanya, sri@it.iitb.ac.in)

***Abstract*— Message ordering is a key component in developing group communication applications. A message ordering protocol must guarantee that when two semantically related messages are sent to a group, one after another, the receivers will deliver them to the application in the same order. Also if two messages are semantically unrelated, delivery of the later message should not be blocked in waiting for the delivery of the earlier message. Traditional solutions like total ordering protocol or causal ordering protocol do not take into account the semantic relationship among messages and hence are inadequate for many distributed group communication applications.**

In this paper we propose a new message ordering, called S_b ordering, and a corresponding protocol, called S_b protocol, which is implemented by every member of the group. The primary objective of the S_b protocol is to order received messages, based on the semantic relationship among them, irrespective of the chronological order in which they are received. As a result, the S_b protocol also minimizes the *delivery delay* at a process (the time from the moment a message is received at a process to the time the message is delivered to the application consuming it), by blocking delivery of a message only if it is yet to receive any semantically preceding message(s). The S_b protocol is a fully distributed protocol and does not rely on any central servers. We present proofs for the correctness and liveness of the protocol and also discuss the time and space complexities for its implementation algorithms.

I. INTRODUCTION

The key property of any group communication [8] is that when messages are sent to the group, all the members receive it. However different members may receive these messages in different order depending on the transmission delays between the sender and receivers. For example if a group member sends a message Y_1 as a response to message X_1, then it is possible that some of the group members may receive Y_1 before X_1 (see Fig 1). Hence a message ordering protocol is required to guarantee that every group member will deliver the message X_1 before delivering Y_1 to the application. Also if X_2 and Y_2 are semantically unrelated messages, then a member receiving Y_2 before X_2, should not block delivery of Y_2 by waiting for the arrival of X_2.

Existing message ordering protocols like total ordering protocol [2] or causal ordering protocol [11] do not allow application−specific semantic relations between messages. The total ordering protocol assumes that two messages (X and Y) are related if they are sent one after another (say, Y after X) according to an assumed global clock. Hence if a process receives message Y before X, it cannot deliver Y to the application until it receives X, even though Y may not be semantically related to X.

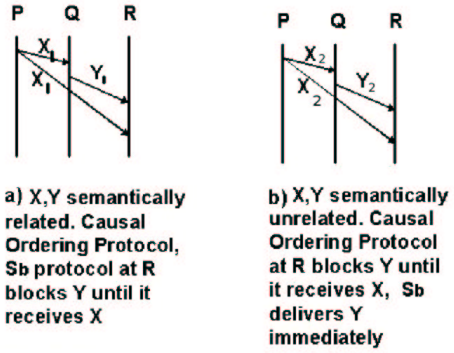

Fig. 1.

Causal ordering, guarantees that if message X, **causally precedes** another message Y then X must be delivered before Y at all their common destinations. The **causal precedence** has been defined in literature based on Lamport's "happened before" relationship, denoted by $\xrightarrow{h_b}$ and defined by transitive closure of following two relationships: (i) X $\xrightarrow{h_b}$ Y, if X and Y are events in the same process and X happens before Y, and (ii) X $\xrightarrow{h_b}$ Y, if event X is sending of the message by one process and event Y is the receipt of same message by another process. If two messages do not have causal precedence relationship then they are said to be concurrent. Nevertheless, if a process receives message Y before X, it cannot deliver Y to the application until it receives X, even though Y may not be semantically related to X. For example, as shown in Fig. 1, process Q sent message Y_2 after receiving message X_2 and hence $X_2 \xrightarrow{h_b} Y_2$. The process R receiving Y_2 before X_2, blocks the delivery of Y_2 until X_2 is received, even though X_2 and Y_2 are not semantically related.

To the best of our knowledge, there is no protocol that considers ordering among messages using semantic relationships among them. In this paper, we propose a new message ordering called S_b ordering (*semantically before*) and a fully distributed message ordering protocol called S_b protocol that preserves application specific semantic ordering among the messages. For example in Fig 1a, as Y_1 is semantically related to X_1,

process Q sends this information to the group along with the message Y_1. If process R is using S_b protocol then on receiving message Y_1 before X_1, it blocks delivery of Y_1 until it receives X_1. On the other hand, as in Fig 1b, as Y_2 is semantically unrelated X_2, process R upon receiving message Y_2 before X_2, delivers Y_2 immediately without waiting for X_2. Hence the message **delivery delay** at the receiver R is reduced.

Since the semantic relationship between two messages is application specific and it is difficult to determine automatically by simply inspecting the messages at the receiver, our protocol derives information about semantic relationships of the messages from the application.

The S_b protocol is fully distributed and does not rely on centralized message ordering servers, thereby avoiding problems such as the server being the point of failure or bottleneck for large groups. Hence it is suitable for peer-to-peer as well as mobile computing systems.

This paper is organized as follows: Section II motivates the need for S_b ordering using an example application. Section III defines S_b ordering and its properties. Section IV describes the S_b protocol and Section V proves its correctness and liveness. Section VI describes the S_b protocol implementation algorithm and discusses its time and space complexities.

II. Motivation and Related Work

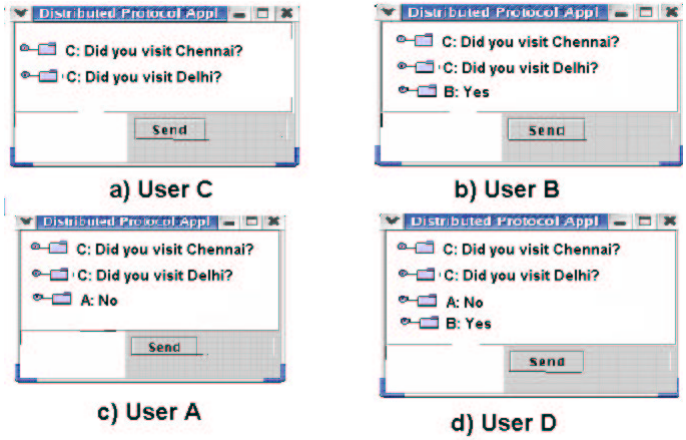

Fig. 2. Chat Application

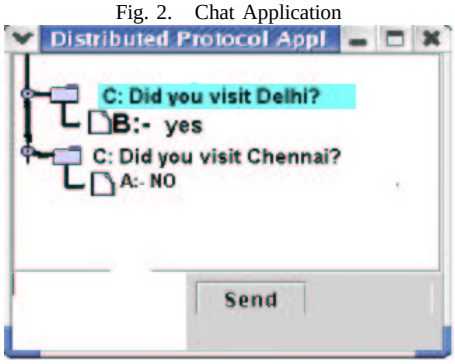

Fig. 3. Threaded Chat Application

Consider a group of processes (A,B,C,D) running on distributed devices and implementing a simple chat application that lets the members of the group interact with each other. The processes communicate with each other by sending messages using a broadcast medium. Suppose the application is implemented using a message ordering protocol based on logical timestamps, such as total ordering [2]. See [15] for a comprehensive survey of total ordering protocols.

As shown in Fig 2, let process C send messages *'Did you visit Chennai?'* and *'Did you visit Delhi?'* with timestamps 1 and 2 respectively. After receiving the above messages, suppose process A replies to the message *'Did you visit Chennai?'* with the response 'No' and process B replies to the message *'Did you visit Delhi?'* with the response 'Yes'. As per total ordering, both A and B would affix the timestamp 3 to their responses. Now, the message ordering protocol at process D on receiving these messages orders them according to their timestamps and displays them on the chat console. However, since there are two messages having the same timestamp, they may get displayed on the console at D in an arbitrary order. This leads to ambiguity because the user at D may not be able to map the responses 'No', 'Yes', to the messages *'Did you visit Delhi?'*, *'Did you visit Chennai?'* appropriately. Hence total ordering protocol is inadequate for such an application. It can be shown that the ambiguity persists even when the messages are ordered using vector clocks, as in causal ordering [11] or even when synchronized global clocks [10] are assumed.

In contrast to the above, consider a threaded chat application [13] that lets users communicate in a **message−response** form as shown in Fig 3. All chat messages are structured in the form of a tree. The key feature of this tree structure is that messages and responses are organized into relationships called **threads**. A user explicitly selects a message before responding to it. As a result, the response is linked directly to the corresponding message, using threads, and other users can perceive the semantic relationship among the messages.

Although the paper [13] does not provide the details of the message ordering protocol used by threaded chat application, such an application can be easily implemented using the S_b protocol. For the above example, upon receipt of messages from process C, process A displays both of them to the user. Now the user at process A would explicitly select the message *'Did you visit Chennai?'* before responding with the message *'No'*. The S_b protocol at process A captures this semantic dependence between the message and its response and sends this information to the group, along with the response. Similarly, the S_b protocol at process B captures the semantic dependence between the message *'Did you visit Delhi?'* and its response *'Yes'* and sends this information to the group. The S_b protocol at D, upon receipt of these responses, orders the messages appropriately and unambiguously, as shown in Fig 3. In the next section, we define the S_b ordering which forms the basis for the S_b protocol.

III. S_b ordering

The primary objective of the S_b protocol is to identify the semantic relationship among received messages and delivering

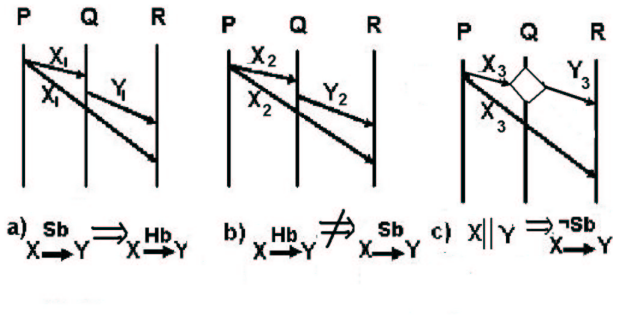

Fig. 4. Relation between S_b and h_b protocols

Fig. 5. Ordering Tree

them to the application in a semantically consistent order. Guaranteeing such ordering involves:

1) Capturing the semantic relationship between a message and its response, from the application at the sender.
2) Representing these semantic relationships in an appropriate form and conveying them to the receivers.
3) Maintaining the relationship at each member of the group with minimum overhead.

In this section, we focus on representing the semantic relationship, which we call S_b **order**. Capturing them and maintaining them are aspects of the S_b protocol and are dealt with in next section.

Definition of S_b order:

Two messages X and Y are said to be related in S_b order if and only if Y is produced semantically in response to a unique message X. This is represented as $X \xrightarrow{S_b} Y$. **Also if X and Y are not semantically related then it is represented as** $X \xrightarrow{\neg S_b} Y$.

For a group of messages, we conceptually represent the semantic relationship among them in the form of a tree, called the **Ordering Tree** (OT), as shown in Fig 5. The OT has the following structure:

- The vertices of the OT are identities of the messages; each message has a unique system-wide identity.
- The directed edges of the OT represent the semantic message−response relationships among messages. There is an edge between any two vertices in the OT, if and only if and the corresponding messages are related in S_b order.
- The root of the OT is a virtual node, denoted by OTR. OTR is assumed to be semantically before all the messages sent to the group. If a message is not a response to any other message in the OT, it is considered to be a response to OTR.

A. Properties of S_b order

Some salient properties of S_b order are as follows:

1) **Response semantics:**
If $X \xrightarrow{S_b} Y$ then P(Y) = X, i.e., X is said to be parent of Y.
The OT represents this relationship in the form of a directed edge between a parent node X and a child node Y. For example, the relationship among the messages exchanged in the threaded chat application of Fig 3 can be represented as in the Fig 5. The directed edge of the tree from C1 to A1 represent the response *'No'* of process A to the message *Did you visit Chennai?* sent by process C, i.e $C1 \xrightarrow{S_b} A1$ and P(A1)=C1.
2) **Uniqueness:**
If $X \xrightarrow{S_b} Y$ then $P(Y) \neq Z$ ($\forall$ Z, $Z \neq X$), i.e., X is unique. The OT represents this by allowing a node to have multiple number of child nodes but a child node can have exactly one parent node. In other words, a message sent to the group may generate multiple responses from various members of the group but any given response is associated with one and only one message and not with multiple messages.
3) **Transitivity:**
$X \xrightarrow{S_b} Y \bigwedge Y \xrightarrow{S_b} Z \implies X \xrightarrow[*]{S_b} Z$.
The OT represents this as having a path from X to Z, if there is an edge from X to Y and an edge from Y to Z. We use the notation $\xrightarrow[*]{S_b}$ to represent such transitive closure. It can be easily seen that the following also hold:
   - $X \xrightarrow[*]{S_b} Y \bigwedge Y \xrightarrow{S_b} Z \implies X \xrightarrow[*]{S_b} Z$
   - $X \xrightarrow{S_b} Y \bigwedge Y \xrightarrow[*]{S_b} Z \implies X \xrightarrow[*]{S_b} Z$
   - $X \xrightarrow[*]{S_b} Y \bigwedge Y \xrightarrow[*]{S_b} Z \implies X \xrightarrow[*]{S_b} Z$

B. Relationship between S_b and h_b

The following relationships hold between S_b (semantically-before) and h_b (happened-before):

1) $X \xrightarrow{S_b} Y \implies X \xrightarrow{h_b} Y$.
If Y is a response to X, then X 'happened-before' Y. For example, in Fig 4a, process Q sends message Y_1 to in response to message X_1 ($X_1 \xrightarrow{S_b} Y_1$). Since process Q can respond to a message only after receiving it, $X_1 \xrightarrow{h_b} Y_1$ also holds.
2) $X \xrightarrow{h_b} Y \not\Rightarrow X \xrightarrow{S_b} Y$.
If Y 'happened-after' X, then Y need not be a response to X. For example, in Fig 4b, process Q sends a message Y_2 after receiving X_2 ($X_2 \xrightarrow{h_b} Y_2$). However, message Y_2 need not be semantically related to X_2.
3) $X \parallel Y \implies X \xrightarrow{\neg S_b} Y$. If X and Y are 'concurrent', then they are not semantically related. For example, in Fig 4c, process Q sends message Y_3 independent of

when it receives message X_3 from the group. Hence message Y_3 is semantically unrelated to message X_3.

IV. S_b PROTOCOL FOR GROUP COMMUNICATION

In this section we present the S_b protocol that includes:

1) At the sender: Captures the S_b order between a message and its response and includes this information while broadcasting the response.
2) At the receiver: Maintains the S_b order information and determines the action to be taken for each received message. A message is delivered immediately to the application either if its parent in the S_b order has been delivered or if it is not a response to any other message, i.e., it has the root of the OT (OTR)as its parent. Otherwise the delivery of the message is deferred, until the receipt and delivery of its parent.

We now describe the system model, data structures and algorithmic details of the S_b protocol.

A. System Model and Assumptions

1) The system consists of a group of processes hosted on devices that communicate through underlying broadcast medium. The composition of the group does not change.
2) The processes do not share any physical memory and communicate by sending messages to the group over broadcast medium. Examples of such systems are group of process hosted on devices connected in LAN or wireless devices that are in communication range of each other.
3) Broadcast medium is assumed to be reliable and guarantees message delivery to every member of the group. However it may suffer from nondeterministic bounded delay in message delivery. Messages in transit need not follow FIFO order.
4) All members of the group have unique identities.
5) Every member of the group maintains a sequence counter and increments it by one every time it sends a message. The sequence counter and the member identity are used to generate a unique identifier for each message.

B. Notations, Message Format and Data Structures

1) Notations::

- $< pid_i >$: denotes identity of the member i of the group.
- $< seqno_i >$: denotes sequence counter value at member i.
- $< mid_i >$: denotes message identity of message i and $< mid_i >$ is $< pid_i, seqno_i >$

2) Message Format:: The message is in the format: $< mid_c, mid_p, data >$ where mid_c is the message identity, mid_p is the identity of its parent ($mid_p \xrightarrow{S_b} mid_c$) and $data$ is the application information. If a message is not a response to any other message then the identity of its parent (mid_p) is set to OTR (Ordering Tree Root).

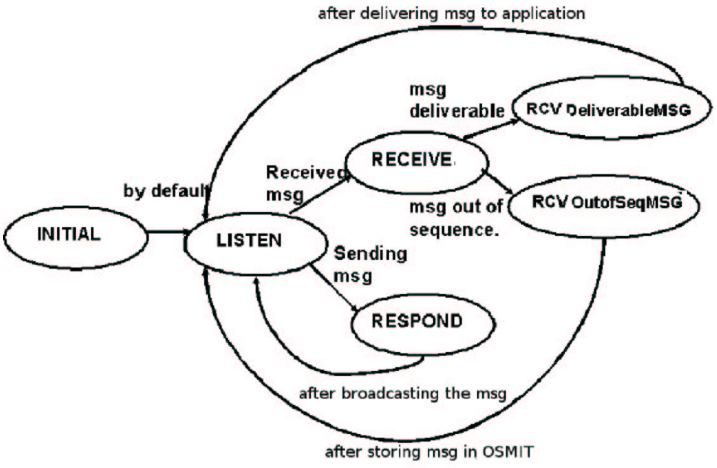

Fig. 6. State Diagram of the protocol

3) Data Structures:: Every process maintains the following two data structures:

1) **Ordering Tree (OT):** As discussed earlier, the OT represents the S_b order among the messages of a group. Each process constructs its OT dynamically by recording the identities of those messages that have been received in S_b order.
2) **Out of Sequence Messages Store (OSMS):** OSMS saves messages that have arrived out of S_b order. For every such message $< mid_c, mid_p, data >$, mid_c, mid_p, $data$ are are saved in the OSMS.

C. Protocol Actions:

The state diagram of the S_b protocol is as shown in Fig 6. In the INITIAL state the data structures OSMS,OT are initialized to $NULL$, OTR (default root of OT)respectively and the process then waits in the LISTEN state. When the application wants to send a message to the group, the process goes to the RESPOND state, where it augments the message with the S_b order information, broadcasts the message and returns to the LISTEN state.

When a message is received from the group, the process goes to the RECEIVE state, where it checks the S_b order information of the message with the OT (Ordering Tree). If it has delivered the parent of the current message, it goes to the RCVDeliverableMSG state, else it goes to the RCVOutSequenceMSG state. In the RCVOutSequenceMSG state, the process simply saves the message in the OSMS and returns to the LISTEN state. In the RCVDeliverableMSG state, the process delivers the message to the application as well as any of its S_b order children that may be saved in the OSMS and returns to the LISTEN state.

A more detailed description of the protocol actions in each state, for a group of n processes, is as follows:

1) **INITIAL STATE:**
   (a)

$$\forall_{i=1}^{n} pid_i \{seqno_i = 0, OSMS_i = NULL, \forall_{j=1}^{n} OT_i = OTR\}$$

(b) go to LISTEN STATE.

2) **LISTEN STATE:**
Listen until a message is received or application wants to respond to a message.

if message is received **then**
go to RECEIVE state
else if application sends a message to the group **then**
go to RESPOND state.
end if

3) **RECEIVE STATE:**
Process i on receiving a message $M =< mid_c, mid_p, data >$,
if $mid_p \neq OTR$ and $mid_p \notin OT_i$ **then**
go to RCVOutSequenceMSG STATE.
else
go to RCVDeliverableMSG STATE
end if

4) **RCVOutSequenceMSG STATE:**
a) insert $< mid_c, mid_p, data >$ in $OSMS_i$
b) go to LISTEN state.

5) **RCVDeliverableMSG STATE:**

a) Call the UpdateOT operation described below with received message M as its parameter.
b) UpdateOT(M)

i) Append $M.mid_c$ into OT_i as a child node of $M.mid_p$.
ii) Deliver the $M.data$ to the application.
iii) /* Let Msg represents a message in $OSMS_i$ and $Msg.mid_p$, $Msg.mid_c$ represent the mid_p, mid_c values of the message Msg respectively. */
for each message $Msg \in OSMS_i$ having $Msg.mid_p == M.mid_c$ **do**
UpdateOT(Msg)
Remove message Msg from $OSMS_i$
end for

c) go to LISTEN state.

6) **RESPOND STATE:** When application at process i responds to a message with identity mid_p, then,
a) $seqno_i = seqno_i + 1$
b) $mid_i =< pid_i, seqno_i >$
c) $broadcasts :< mid_i, mid_p, data >$
d) go to LISTEN state.

If message is not related to prior messages then mid_p is OTR.

D. *Protocol illustration*

Consider a group formed by two processes with idenitities A and B respectively. We follow the Ordering Tree representation explained in Section II, to show the semantic relationships among the messages exchanged between A and B. Recall that the root of the tree is a default node OTR and we assume OTR is semantically before every message sent to the group. The vertex of the tree represents message identity in the format $< pid, seqno >$.

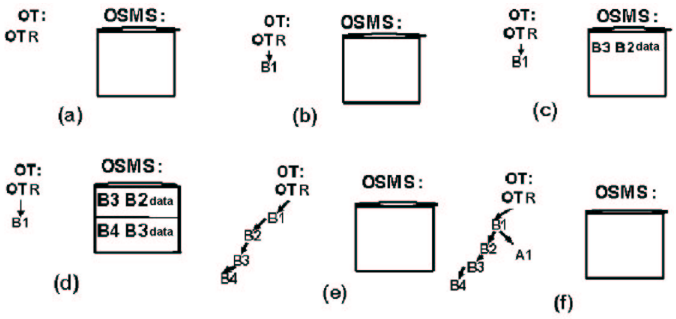

Fig. 7. S_b protocol illustration

Protocol Actions at process A are illustrated below:

1) In the initial state, the OT_A, OT_B at process A,B respectively contains root node OTR. Also the $OSMS_A$, $OSMS_B$ at process A,B respectively are empty. The sequence counters $seqno_A$, $seqno_B$ are set to zero. The process remains in LISTEN state until a message is received from the group or application responds to a previous message. The ordering tree is as shown in Fig 7a.

2) Process A on receiving a message $< B1, OTR, data >$ enters `RECEIVE` state. Since $m_p = OTR$ the process A goes to `RCVDelivarableMSG` state, saves B1 in OT_A as child node to OTR, and delivers the $data$ to the application. As the $OSMS_A$ is empty, it cannot find any messages m_r such that $(m_c \xrightarrow[*]{S_b} m_r)$ and goes back to LISTEN state. The data structures at process A are as shown in Fig 7b.

3) Process A on receiving a message $< B3, B2, data >$ enters RECEIVE state. Since $B2$ (mid_p field of the message), is not present in OT_A the protocol goes to `RCVOutSequenceMSG`. Following the S_b protocol, the $< B3 >$, $< B2 >$, and $data$ are saved in $OSMS_A$ as shown in Fig 7c.

4) Similarly on receiving $< B4, B3, data >$, the process A goes to RECEIVE state and then to `RCVOutSequenceMSG` because B3 is not in OL_{AB}. $B4$, $B3$, $data$ are saved in OSM_A. The Fig 7d shows $OSMS_A$, OT_A.

5) On receiving $< B2, B1, data >$, the process goes to RECEIVE state and then to `RCVDeliverableMSG` state because $B1$ is present in OL_A. In this state, the protocol delivers $data$ corresponding to received message and saves its identity $B2$ is saved in OL_A as child node of $B1$. Messages with identities B3, B4

are retrieved from $OSMS_A$ because B2$\xrightarrow{S_b}$B3 directly and B2$\xrightarrow[*]{S_b}$B4 transitively. The $< data >$ corresponding to messages with identities $B3, B4$ are delivered to the application in S_b order and identities B3, B4 are saved in OT_A as shown in Fig 7e.

6) If the application at process A wants to respond to a message having identity $B2$, then it goes to RESPOND STATE. It increments the sequence counter value to 1, receives $data$ from the application and broadcasts the message in the format $< A1, B2, data >$. Since the message is a broadcast, process A on receiving its own message goes to `RCVDelivarableMSG` delivers $data$ and updates OT_A as shown in(Fig 7f).

V. S_b PROTOCOL CORRECTNESS AND LIVENESS

We prove the correctness and liveness of the S_b protocol by using the semantic relationships among the messages as represented by the Ordering Tree (OT). Recall that by the transitive closure property of S_b order, the root of the OT is semantically before all messages sent to the group. $OTR \xrightarrow[*]{S_b} \{\forall messages \in OT\}$.

A. Correctness

Theorem 1: The S_b protocol preserves S_b ordering.
Explanation: Let $(OTR, A_1, A_2, A_3, A_4 \ldots A_n)$ be the identities of the messages in the OT along the path from root to any node A_n of the OT. As these messages are in S_b order, we need to prove that the S_b protocol delivers them to the application in the same order.
Proof:
The proof is by induction on the number of messages n.
Base case:
For n = 1, the first message having identity A_1 sent to the group is not semantically before any other messages except OTR. Hence every process receiving it will deliver information corresponding to A_1 to the application. A_1 is represented as the child of OTR in OT.

If a process does not receive A_1 but receives a message A_2 such that $A_1 \xrightarrow[*]{S_b} A2$ then the process saves A_2 in the OSMS until it receives message A_1. The process after receiving A_1 will deliver information corresponding to A_1 to the application before delivering information of A_2. Hence the ordering is preserved.

Induction Hypothesis:
Assume that the S_b protocol preserves the ordering for n messages i.e for the messages $(OTR, A_1, A_2, A_3, A_4 \ldots A_n)$ in the ordering tree (OT).

Induction Step:
Suppose a member of the group sends a new message in the format $< A_{new}, mid_p, data >$ to the group.

- **If mid_p is node A_n**: Since A_n is present in the OT, the $data$ corresponding to identity A_{new} is delivered to the application and A_{new} becomes child node of A_n in OT. Hence the protocol preserves the ordering for $n+1$ messages $(OTR, A_1, A_2, A_3, \ldots A_n, A_{new})$.

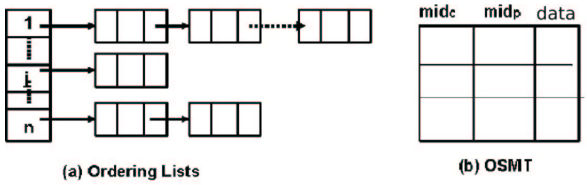

Fig. 8. Data Structures

- **If mid_p is any node A_k in OT**. In this case also, the $data$ corresponding to identity A_{new} is delivered to the application and A_{new} becomes child node of A_k in OT. Hence the protocol preserves the ordering for $n+1$ messages.
- **If mid_p is not in OT**: In this case, A_{new} is saved in the OSMS until the process receives mid_p. Upon delivery of the message corresponding to mid_p, mid_p would be inserted into the OT. Now A_{new} is removed from the OSMS and also delivered to application. A_{new} then becomes child node of mid_p in OT. Hence the protocol preserves the ordering for $n+1$ messages.

B. Liveness

Theorem 2: The S_b protocol is liveness preserving.
Explanation: Every message sent to the group will be eventually delivered to the applications at every process. We need to prove that no process will block a message indefinitely.
Proof:
Consider a message M for which **n** responses have been generated in the group. Consider the receipt of one of these responses, R, at process **i**.

- **If process i has delivered message M to the application**: In this case process **i** also delivers response R immediately, irrespective of other **n-1** responses.
- **If process i has not received message M**: In this case process **i** saves response R in the OSMS and waits for receipt of message M. Since the underlying broadcast medium is assumed to provide reliable message delivery, message M would be eventually delivered to process *i*. When process *i* receives message M, and subsequently delivers it, it traverses the OSMS and also delivers response R.

VI. PROTOCOL IMPLEMENTATION

In this section, we discuss the protocol algorithm details and the corresponding time and space complexities.
The data structures required for S_b protocol at a process i are implemented as follows:

1) **Ordering Tree (OT_i):** We implement this as an array of linked lists, called Ordering Lists (OL_i), as shown in Fig 8a. The size of the array is equal to the number of processes present in the group. Each array element $OL_i[j]$ saves the starting address of the linked list corresponding to process j and the linked list saves the sequence numbers of the messages received from

process j. The data structure support the following operations:

a) **InsertInOL($seqno_j$)**: inserts $seqno_j$ in linked list starting at array element j.

b) **IsPresentInOL($seqno_j$)**: searches for $seqno_j$ in linked list starting at array element j. If the $seqno_j$ is present then returns true else returns false.

2) **Out of Sequence Message Store($OSMS_i$):** We implement this as a 2-dimensional array of 3 columns each and some finite number of rows called Out of Sequence Message Table ($OSMT_i$) as shown in Fig 8b. Message identities of the messages that have arrived out of sequence are saved here. The process i on receiving a out of sequence message $< mid_c, mid_p, data >$ saves the identity of the parent message mid_c in the first column of the row, identity of the message mid_p in the second column and the $data$ in the third column.
The data structure supports the following operations:

a) **InsertInOSMT($mid_c, mid_p, data$):**
The operation uses the first empty row available from the top of the $OSMT_i$ table and inserts mid_c, mid_p, $data$ in the first, second, third columns of the row respectively.

b) **getRow(mid_p):**
The operation searches in linear manner from the beginning of the $OSMT_i$ table and returns the index of row containing mid_p value in its second column. If there are multiple rows containing mid_p in their second column then it returns the first row that it encounters while searching from the beginning of the list.

c) **putOSMsgsInOL(mid_c):**
The operation identifies all the rows of $OSMT_i$ containing messages for which mid_c is either directly or transitively semantically before them. The operation transfers the identities of these messages to OL_i and $data$ corresponding to these messages to application in S_b order. The rows containing these messages are marked empty for reusing them.

A. *Protocol at process* i

1) **In the initial state,**

for $j = 1$ to n (where n is the number of members present in the group.) **do**
$OL_i[j]$=NULL
end for
$seqno_i$=0
The rows of $OSMS_i$ are marked empty.

2) **Process i on receiving a message:**
On receiving a message $< mid_c, mid_p, data >$ from group, where $mid_c =< pid_c, seqno_c >$ and $mid_p =< pid_p, seqno_p >$

if $mid_p \neq \emptyset$ and **IsPresentInOL**($seqno_p$) is false **then**
InsertInOSMT($mid_c, mid_p, data$)
else
InsertInOL($seqno_c$)
Deliver $data$ to the application
putOSMsgsInOL(mid_c)
end if

3) **Process i for responding to a message having identity mid_p**

a) $seqno_i$ = $seqno_i$ + 1;

b) mid_i = $< pid_i, seqno_i >$;

c) Broadcasts: $< mid_i, mid_p, data >$

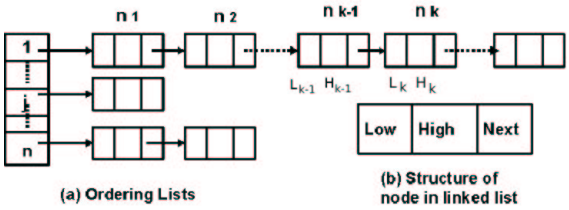

Fig. 9. Ordering List Data structure

B. *Ordering List Operations*

The structure of Ordering List is as shown in Fig 9. An array element $OL_i[k]$ saves the starting address of the linked list corresponding process k. Each node of the linked list contains there 3 fields. The first two fields called Low (L) and High (H) contains sequence numbers of the messages and the third field points to next node. The values L and H indicate that the process i has received all the messages having sequence numbers between L and H inclusive.

1) ***IsPresentInOL($seqno_j$):*** The operation searches for $seqno_j$ in linked list starting at $OL_i[j]$. If the $seqno_j$ is present then returns true else returns false.

Algorithm 1 IsPresentInOL($seqno_j$)

```
1: S_j = OL_i[j] /* Starting address of linked list */
2: Scan the list and find the node n_k such that L_k is greatest
   number less than or equal to seqno_j
3: if H_k ≤ seqno_j then
4:    return true
5: else
6:    return false
7: end if
```

Time complexity: Scanning the list takes linear time, i.e., O(m) where m is the number of nodes present in the linked list.

2) ***InsertInOL($seqno_j$):*** The operation inserts $seqno_j$ in linked list starting at $OL_i[j]$. If process i receives messages with continuous sequence numbers starting from L and ending at H from a process j then these messages are represented in OL_i by storing only L and H values in a node of linked list starting at $OL_i[j]$. The linked list stores these values in the non decreasing order.
So, to insert $seqno_j$, the linked list starting at $OL_i[j]$ is

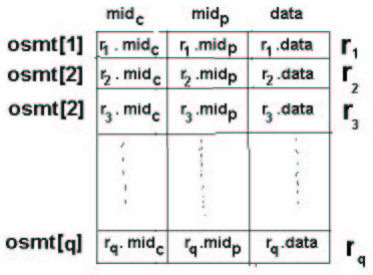

Fig. 10. Ordering List Data structure

scanned to find the node n_k such that L_k is the least value greater than $seqno_j$ as shown in Fig 9a. Let n_{k-1} be the node preceding it. If $seqno_j$ is one more than H_{k-1} and one less than L_k then H_{k-1} is replaced by $seqno_j$ and the node n_k is deleted. Otherwise if $seqno_j$ is one more than H_{k-1} then H_{k-1} is replaced by $seqno_j$. Or else if $seqno_j$ is one less than L_k then L_k is replaced by $seqno_j$. If none of the above conditions satisfy then a new node n_{new} is created with L_{new} and H_{new} fields set to $seqno_j$ and n_{new} is inserted between n_{k-1} and n_k.

Algorithm 2 InsertInOL($seqno_j$)

1: 1. Scan the list and find the node n_k such that L_k is the least number greater than $seqno_j$.
2: **if** $H_{k-1} = seqno_j - 1$ and $L_k = seqno_j + 1$ **then**
3: 1.$H_{k-1} = H_k$
4: 2.Delete node k
5: **else if** $L_k = seqno_j + 1$ **then**
6: $L_k = seqno_j$
7: **else if** $H_{k-1} = seqno_j$-1 **then**
8: $H_k = seqno_j$
9: **else**
10: 1.Create a new node new
2.Assign $L_{new} = H_{new} = seqno_j$
3.Insert it between nodes n_{k-1} and n_k.
11: **end if**

Time complexity:
Scanning the linked list takes linear time, i.e., O(m) where m is the number of nodes present in linked list.

C. *OSMT operations*

The structure of $OSMT_i$ is shown in detail in Fig 10. We assume the table contains a maximum of **q** rows [1] for analyzing time complexity of the OSMT operations. Also for sake of clarity, we refer to a row of table $OSMT_i[k]$ as r_k and columns corresponding to row r_k i.e., $OSMT_i[k][0]$, $OSMT_i[k][1]$, $OSMT_i[k][2]$ as $r_k.mid_c$, $r_k.mid_p$, $r_k.data$ respectively.

[1]We assume q number rows of OSMT are sufficient for a process to save every message that arrives out of sequence.

1) ***InsertInOSMT(****$mid_c, mid_p, data$****):*** As explained earlier the operation performs linear search starting from the beginning of the table and searches until an empty row is found to insert the values mid_c,mid_p,$data$.
Time complexity:
Hence the InsertInOSMT operation takes O(p) time.

2) ***getRow(****mid_y****):*** As explained earlier the operation returns row r_x from OSMT such that $r_x.mid_p = mid_y$. If such row does not exist than it returns null. We assume the operation performs linear search from starting of the table to find the required row.

Time complexity:
Hence the above operation takes linear time, i.e., O(q) time.

3) ***Operation: putOSMsgInOL(****mid_x****):*** The OSMT contains collection of prospective edges of ordering tree OT that arrived out of S_b sequence in the form of rows of OSMT with each row containing parent and child message identities in columns mid_p, mid_c respectively. The operation starting from mid_x performs depth first search (DFS) on the contents of OSMT. It identifies the rows of OSMT that forms the edges of prospective sub tree of OT having mid_x as its root. During depth first search, every time a row is visited, the operation removes the row from OSMT, appends mid_c to OT and delivers $data$ to application. Hence the messages that arrive out of S_b order are delivered to the application in S_b order.

Algorithm 3 putOSMsgInOL(mid_y)

1: **for** each row (r_x=getRow(mid_y))≠NULL **do**
2: Deliver r_x.data to application
3: InsertInOL($r_x.mid_c$)
4: putOSMsgInOL($r_x.mid_c$)
5: remove row r_x from OSMT
6: **end for**

Time complexity:

Searching the $OSMT_i$ and putting them into the OL_i takes quadratic time, i.e., $O((m+p)^2)$ (where m and p are as before).

This may be computed in detail as follows: Depth First Search algorithm for a tree with e edges takes O(e) time. Hence the DFS algorithm for $OSMT_i$ with a maximum of q rows takes O(q) time. To identify each edge of the tree and transferring it to OL_i, getRow operation at line 1 and InsertInOL operation in line 3 takes O(q),O(m) time respectively. Hence time taken by these operations together for each edge is O(m+q). Hence the operation putOSMsgInOL takes O(q(m+q)) time or $O((m+q)^2)$ time.

D. *Complexity of S_b protocol*

1) ***Time Complexity****:* The major operations of the protocol occur at a process while sending and receiving of messages.

- Sending a message takes constant time as it involves only capturing mid_p from the application, incrementing the sequence number, and broadcasting the message.

- On receiving a message, the **IsPresentInOL()** operation takes 0(m) time to check whether the message is deliverable to the application or not. If the message is not deliverable, then operation **InsertInOSMT()** takes linear time proportional to number of rows in $OSMT_i$ i.e., O(q). If the message is deliverable, then the operations performed are **InsertInOL(**$seqno_c$**)**, **putOSMsgInOL(**mid_c**)** and time complexity of each of these operations is 0(m),$0((m+q)^2)$ respectively. (as discussed earlier). Hence the total time complexity of the S_b protocol is **0(**$(m+q)^2$**)**.

*2) **Space Complexity**:* The linked lists in the OL store only the first and last values of contiguous sequence numbers of messages received from a process.

- The **Space complexity for** OL_i**:**The linked lists in the OL store only the first and last values of contiguous sequence numbers of messages received from a process. The **best case** occurs when all the received messages have contiguous sequence numbers. Hence in the best case, each linked list contains only one node and size of array of linked lists is O(n) for a group with n processes. The **worst case** is when a process receives messages with alternate sequence numbers from every process. In this case, the number of nodes in each linked list is equal to the number of messages received from that process and the number of nodes present in the OL_i is equal to the number of messages received by the process from all members of the group. Hence in the worst case, the size of the OL is bounded only by the device's memory limits.
- The **Space Complexity for** $OSMT_i$**:**, the **best case** is when all messages are received in S_b order and the number of entries in the $OSMT_i$ is zero and the **worst case** is when a process does not receive the messages corresponding to the nodes closer the root of the OT_i and the number of entries in the $OSMT_i$ is bounded only by the number of rows allocated.

VII. Conclusion

Message ordering protocols are key components for group communication systems. The most widely used message ordering protocols like total ordering, causal ordering protocols are not suitable for all group communication applications because they do not let the application explicitly specify the order among the messages.

In this paper we have defined a new ordering called S_b that orders the messages according to the semantic relationship among them as specified by the application and we described a protocol called S_b protocol that ensures all the receivers will receive the messages sent to the group in S_b order. We have explained the protocol actions with a state diagram. We have proved the correctness and liveness of the protocol and discussed the implementation issues and time, space complexity of protocol. Our future work lies in extending the protocol to dynamic groups where members may join or leave at any time.

References

[1] C.Fidge. Timestamps in message passing systems that preserve the partial ordering. *Proceedings of 11th Australian Computer Science, page56-66*, pages 56–66, 1988.

[2] D.R.Cheriton and D.Skeen. Understanding the limitations of causally and totally ordered communication. *In the Preceedings of 14th ACM Symposium on Operating System Principles*, pages 44–57, 1993.

[3] C.Fetzer F.Cristian. The timed asynchronous distributed system model. *IEEE Trans. Parall. Distrib. Syst. 10,6*, 1999.

[4] S.Mishra F.Cristian. The pinwheel asynchronous atomic broadcast protocols. *In Proceedings of 2nd International Symposium on Autonomous Decentralized Systems. IEEE Computer Society Press.*, 1995.

[5] S.Mishra. F.Cristian, R.Debeijer. A performance companrision of asynchronous atomic broadcast protocols. *Distrib. Syst. Eng. J.1,4,177-201*, 1994.

[6] A.Schiper F.Pedone. Handling message semantics with generic broadcast protocols. *Distrib. Comput. 15,2,97-107*, 2002.

[7] A.Schiper. F.Pedone. Optimistic atomic broadcast: A pragmatic viewpoint. *Theor. Comput. Sci.291,79-101*, 2003.

[8] R.Vitenberg. G.Chockler, I.Keidar. Group communication specifications:a comprehensive study. *ACM Computing Surv. 9,2(Feb.), 427-469*, 2001.

[9] F. Viegas J. Donath, K. Karahalios. Visualizing conversation. *Proceedings of HICSS-32*, 1999.

[10] L.Lamport. Using time instead of time-outs in fault-tolerant systems. *ACM Trans. Program. Lang. Syst. 6,256-280*, 1984.

[11] L.Lamport. Time,clocks, and the ordering of events in a distributed system. *Communication of ACM*, July 1978.

[12] M.Dasser. Tomp: A total ordering multicast protocol. *ACM Operat.Syst.Rev.26,1*, 1992.

[13] B.Burkhalter M.Smith, JJ. Cadiz. Conversion trees and threaded chats. *ACM Magazine*, 2000.

[14] P.Urban X.Defago, A.Schiper. Comparitive performance analysis of ordering strategies in atomic broadcast algorithms. *IEICE Trans. Inf. Syst. E86-D,12*, 2003.

[15] P.Urban. X.Defago, A.Schiper. Total order broadcast and multicast algorithms: Taxonomy and survery. *ACM Computing Surveys,Vol. 36,No. 4 pp.372-421*, December 2004.