A Sequencer-based Total Order Protocol for JavaGroups

Manish Sambhu
Department of Computer Science
Cornell University
mms21@cornell.edu

May, 1999

Introduction

JavaGroups is a useful group communication toolkit for Java. One of the advantages of JavaGroups is that applications can select a set of protocols to use based on properties desired by the application. One such property may include maintaining total order of messages broadcast to the group. The TOTAL protocol maintains this total order property via a sequencer-based mechanism. The figure below illustrates a simple example of this mechanism.

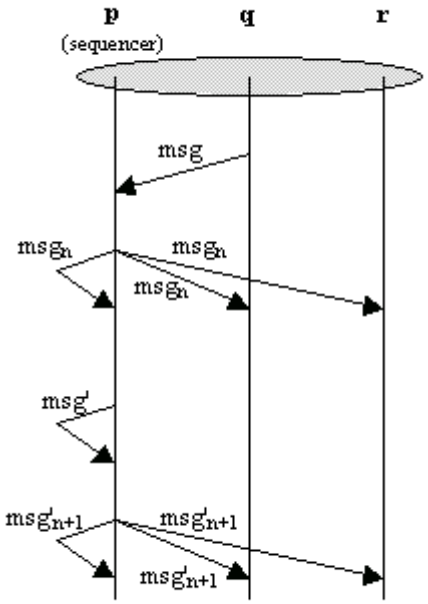

Figure 1: Simple example of the sequencer-based total order mechanism

A total order protocol has numerous applications within a group environment. One class of examples includes applications where a group of servers need to maintain a consistent state where the state changes based on received messages. If two different servers receive messages in different orders, their states may diverge. If messages are guaranteed to be received in the same order, each server has the same sequence of states.

In order for this TOTAL protocol to function properly, it needs a group membership protocol residing below. This group membership layer should pass up VIEW_CHANGE events whenever the current view changes. The group membership layer also needs to guarantee virtual synchrony so that messages sent in one view may only be received in that view. In addition, the TOTAL layer expects a SET_LOCAL_ADDRESS event to come up the stack at some point (usually at the beginning of the connection).

Design

Architecture

The TOTAL protocol is based on the existence of a group so that broadcast messages to the group can be totally ordered. One of the members of the group is declared the sequencer, but all members (or servers) in the group hold some responsibility for maintaining total order. Each message that is broadcast to the group has an associated sequence id. For each new broadcast message received by a server, it expects the sequence id of the message to increment by one. Therefore, each member can deterministically expect messages to arrive in a certain order. If the TOTAL layer receives a message that is out of order, it will not be passed up to the next layer. It can either discard the message or save it until it is the appropriate next message to pass up. With this scheme, each server can independently pass up messages in order according to their sequence id. Hence, if every message broadcast to the group has a unique sequence id, the TOTAL layer in each member will pass up these messages in the same order. For now, we will assume the uniqueness of sequence id's but it will be enforced by the sequencer.

Members

As described above, members are responsible for passing up messages based on sequence id. Each server maintains state for the next expected sequence id. A message with this sequence id would be the next message passed up the protocol stack. If we receive a message with a lower sequence id, then this message can be discarded because it has already been received and passed up the stack. If we receive a message that has a sequence id that is greater than or equal to the next expected sequence, then we have not passed this message up the stack yet. The message is stored in a queue of messages ordered by sequence id. If the queue already contains a message with this sequence id, then the message is discarded since the messages are the same (by our assumption of unique sequence id's). Each time a new message is added to this queue, the server checks if it can safely pass any messages up the protocol stack, that is, does the queue contain a message with the next expected sequence id. If so, then the message is removed from the queue and passed up the stack. The next expected sequence id is also incremented by one. This process repeats until either the queue is empty or the lowest sequence id stored is greater than the next expected sequence id.

In addition to receiving broadcast messages, members may also want to send their own broadcast messages. Typically, each member would broadcast its own message but maintaining total order would be difficult without additional conditions. If a member wishes to broadcast a message, it sends it only the sequencer instead of the entire group. The sequencer broadcasts the message to the group.

Sequencer

Each group using the TOTAL protocol layer will consist of exactly one sequencer. Selection of the sequencer occurs implicitly with the list of members supplied by the group membership layer residing below the TOTAL layer. Since this member list is guaranteed to have the same contents and to be ordered identically between all members, we can safely choose the first listed member as the sequencer and know that all members have chosen a consistent sequencer.

One responsibility of the sequencer is to forward messages from a member to the group on behalf of that member. When the sequencer receives such a broadcast request, it assigns a sequence id to the message and broadcasts it to the group. In assigning sequence id's to messages, the sequencer needs to guarantee that no two messages receive the same sequence id and that no sequence id is skipped when assigning sequence id's. To implement these policies, the sequencer maintains state for the next sequence id to assign, similar to how a member keeps state for the next sequence id to expect. Each time the sequencer receives a broadcast request, it assigns the next sequence id to assign to this message and then increments the next sequence id to assign. Finally, it broadcasts the message (with the assigned sequence id) to the group.

View Changes

Since a view change alters the list of members in the group, a potentially new sequencer gets chosen after each view change. With this new sequencer, a problem arises in assigning unique sequence id's to messages because the new sequencer may not know next sequence id that the old sequencer would have assigned. To address this potential problem, we claim that the new sequencer can begin assigning sequences at any value. Hence, the new sequencer does not need to know the next sequence id that would have been assigned. Properties maintained by the group membership layer allow us to reset the sequence assignment with a view change. Since the group membership layer guarantees that messages sent in a certain view may only be received in that view (virtual synchrony), we know that a message sent in one view will not be received by any member while in any future view. As a result, we can consider each view independently in terms of messages being sent and received so the starting sequence id of one view is not dependent on the sequence id's of any other view. Since the TOTAL layer queues messages, the members clear this queue when they observe a VIEW_CHANGE event in order to maintain independence between views. Another issue that arises with a view change is that the members may not know what sequence id to expect for the first

message in the view. To address this issue, whenever there is a view change, the new sequencer broadcasts a TOTAL_NEW_VIEW message that contains the sequence id of the first broadcast message that will be assigned for the new view.

Retransmission

Since there are no reliability guarantees on message delivery, some messages may get lost or corrupted in transit. This problem is only a major issue for messages being broadcast by the sequencer. If a member never receives a message with a certain sequence id, it will not pass any subsequent messages up the stack and all of these messages will remain queued (until the view changes and they are cleared). The TOTAL protocol implements retransmission for more reliable communication of broadcast messages. Members can request a resend from the sequencer for a message with a certain sequence id. With this structure, each member has the burden of requesting the appropriate requests so that there are no sequence id's that are skipped. Each member contains a retransmission thread that determines if any resend requests need to be sent and sends these requests to the sequencer. To decide if a message needs to be transmitted, the thread periodically checks the queue of messages. It also knows the next sequence id to expect. If the queue of messages is not empty and its lowest sequence id is greater than the next expected sequence id, then a resend request is sent to the sequencer asking for the message with the next expected sequence id. This thread can also make multiple requests in a given iteration if there is a larger gap between the first queued message and next expected sequence id. The thread also holds state for what it has requested most recently and when the resend request was sent. The state for the retransmission thread gets cleared whenever the view changes.

Future Improvements

There are several areas for potential improvements in the current implementation of the TOTAL protocol. The protocol does not currently handle MERGE events passed up the protocol stack. If two groups merge, the result should be similar to a new view being installed. Another enhancement would involve the addition of a flush protocol, which would likely be a completely independent protocol residing below the TOTAL layer in the protocol stack. Without a flush protocol, the TOTAL layer only guarantees total order for a prefix of the messages in one view. In other words, for a given view, messages received by multiple servers will be received in the same order. There is vulnerability when the view changes in that some servers may not receive messages at the end of a view while other servers do receive these messages. When the view changes, the servers that did not receive the later messages will never receive them because of the virtual synchrony guarantee. A flush protocol would delay the view change until all members of the group are synchronized.

Enhancements could also be made to the retransmission mechanism. Since it uses a negative acknowledgement scheme to manage resend requests, the retransmission thread

does not know if there is a suffix of messages that needs to be retransmitted. If the sequencer handled retransmission (perhaps using positive acknowledgements) or each server knew the sequence id of the last broadcast message, then the retransmission thread could actively request a lost suffix of messages. The retransmission thread could also use a windowed resend request structure with intelligent backoff instead of constant request limits and timeouts.

References

Bela Ban. *Design and Implementation of a Reliable Group Communication Toolkit for Java.*
http://www.cs.cornell.edu/home/bba/Coots.ps.gz

Bela Ban. *JavaGroups User's Guide.*
http://www.cs.cornell.edu/home/bba/user.ps.gz

Alexey Vaysburd. *Building Reliable Interoperable Distributed Objects with the Maestro Tools.*
http://www.cs.cornell.edu/home/alexey/thesis.ps

Appendix

TotalHeader types

Header type	Sequence id argument	Description
TOTAL_UNICAST	N/A	A unicast message not to be processed by the TOTAL layer
TOTAL_BCAST	the sequence id assigned to this broadcast message	A normal broadcast message to the group
TOTAL_REQUEST	N/A	A request (sent to the sequencer) for a message to be broadcast to the group
TOTAL_NEW_VIEW	the sequence id that will be used for the first broadcast message sent in this view	Message sent when the view changes, resetting the state of the TOTAL layer (sent by sequencer of the new view)
TOTAL_NEW_VIEW_ACK	N/A	Acknowledgement of the TOTAL_NEW_VIEW message. This header type is not currently used
TOTAL_CUM_SEQ_ACK	the cumulative sequence id being acknowledged	A cumulative acknowledgement of all messages in the current view up to the specified sequence id (inclusive)
TOTAL_SEQ_ACK	the single sequence id that is being acknowledged	The acknowledgement of a single message with the specified sequence id. This header type is not currently used
TOTAL_RESEND	the sequence id of the message to be sent again	A request (to the sequencer) to resend the message with the specified sequence id