Implementing Totems' total ordering protocol in JavaGroups reliable group communication toolkit

Vladimir Blagojevic
Dept. of Computer Science
York University
vladimir@cs.yorku.ca

Abstract

Reliable group communication systems emerged in recent years, including Horus [vRBM96], Totem [AMMS+95], JavaGroups [Ban98] and others. They all used similar notion of process group in order to bring reliability, fault tolerance and load balancing to distributed computing. Totem and JavaGroups implement process groups somewhat differently. This paper discusses implementation of Totem's single ring totally ordered message protocol in JavaGroups group communication toolkit.

1 Introduction

There has been a significant surge in demand for a fault tolerant, load balanced and reliable distributed systems in the past decade. It is a difficult task to develop these process group based distributed systems. Critical components must be replicated within the systems so that when one component fails others can continue to work uninhibited.

Maintaining consistency of the replicated data introduces further problems. In addition to replication, process groups are usually expected to load balance the requests among the group members. They are also expected to divide larger problems into smaller ones and to process them concurrently by different members of the systems.

However, these distributed process group based systems do not attempt to solve a particular instance of a certain distributed problem. Rather, they act as a toolkit that is used by application clients to solve any such problem generally solvable in a distributed computation model.

JavaGroups is a first such reliable group communication toolkit implemented entirely in Java. It is modelled after Ensemble and Horus but attempts to exploit the benefits of the Java platform. It is roughly modelled as a collection of protocol stacks above connectionless UDP and unreliable multicast service provided by the operating system.

In section 2 we will overview the general architecture of Horus and JavaGroups reliable group communication systems. In section 3 we will focus on the details of JavaGroups design. Section 4 points the differences and similarities of Totem and JavaGroups while section 5 focuses in detail on implementation of JavaGroups total token ordering layer. We close the discussion with preliminary performance data, conclusion and brief look into future work.

2 Architecture

Reliable group communication systems have to provide multiple message ordering capabilities in order to support a vast range of distributed application requirements. In some scenarios

FIFO ordering between message senders is strong enough to satisfy message ordering requirements, while in others FIFO order is not sufficient and thus CAUSAL and/or TOTAL order is required.

In certain scenarios, the requirement of the application could be to satisfy properties such as virtual synchrony [BSS91] .However, in other scenarios, virtual synchrony could introduce significant overheads unacceptable to other system requirements (such as realtime communication). A data encryption option may be desirable over insecure networks, but at the same time superfluous in a closed network environment behind a firewall (i.e., local LAN).

The above observations illustrate a need for architecture of the system where it is possible to specify runtime constraints for certain runtime environments, and to have systems that assemble themselves according to specified runtime parameters. The central abstraction that emerges from these observations is a protocol abstraction.

Indeed, the foundation JavaGroups is a flexible protocol stack architecture that allows users to put together custom tailored stacks. This provides different ordering properties and reliability guarantees. Some of the examples of the protocols in the stack include the UDP protocol layer at the bottom of the stack, which is an adaptation to OS provided udp and multicast service. NAKACK protocol is used in combination with UDP in order to provide lossless transmission. STABLE protocol layer tracks which messages have been received by which members of the communication group. STABLE layer is usually put on top of NAKACK layer so that messages retained in NAKACK (for possible retransmission) can eventually be removed and garbage collected.

In addition to above mentioned protocols, various ordering protocols can be added on top of reliable transmission protocols. FIFO order is already implemented in NAKACK layer. CAUSAL and TOTAL order may be included in the stack in order to provide causal message delivery as well as total order between the communication group nodes.

Consistent group membership information is provided by the Group Membership Service (GMS). The GMS tracks the membership of the group, and notifies group members when a new member joins or an existing member leaves or crashes. Failure detector (FD) protocol is used to suspect crashed members. In the event of communication link failures, MERGE layer can be employed to consolidate two or more groups into one. If there is a requirement that new members have to obtain the current state of existing members, then the STATE_TRANSFER protocol must be present in this custommade stack. Additionally, CRYPT protocol can be used for enhanced security, TUNNEL protocol layer can be used to connect multiple groups over a long distance internet link, and so on.

3 JavaGroups design overview

As pointed out in the previous section, `Protocol` is the central abstraction in JavaGroups design. JavaGroups' `Protocol` class is a superclass for defining protocols. Every protocol has to extend abstract class `Protocol`. `Protocol` class provides interface that enables easy protocol stacking as well as common services that all protocols use. `Protocol` provides two queues and two threads each servicing one queue. A thread servicing `up_queue` is called `up_handler` while the thread servicing `down_queue` is called `down_handler` accordingly. `up_handler` thread retrieves messages that have arrived in this layer's `up_queue` from the protocol layer below, executing the layers up method and finally passing messages to the layer above. `down_handler` thread retrieves messages that have arrived in this layer's `down_queue` from the protocol layer above, executing down method call and finally passing messages to the protocol layer below. `Protocol` subclasses must provide an implementation of the up and down interface calls.

A single `Protocol` instance in the protocol stack provides certain quality of service guaranteed by the protocol stack as a whole. Architecture based on protocol stacking allows different protocol layers to be highly isolated and independent from implementation specifics. For example, currently there are two different algorithms providing total message ordering. TOTAL_TOKEN described in this paper implements total ordering using token passing while TOTAL layer provides total message ordering using sequencer [KT96] approach. The protocol stack as a whole is not concerned about the implementation specifics of the totally ordered layer as long as it is satisfying the desired property and functioning according to the specification.

However, it does not semantically make sense to randomly order the protocols in the stack. JavaGroups has a mechanism that prevents random protocol stacking. Each `Protocol` defines minimum set of other protocols that need to be present up and down in the protocol stack relative to that protocol. `ProtocolStack` class wraps a set of stacked and interconnected protocols, loads their configuration from any network location, and controls the lifetime of all the protocols in the stack.

Application clients do not interact with the `ProtocolStack` instance but rather a higher level abstraction `Channel`. `Channel` represents an indirection layer enabling different alternatives of a group communication transport, while retaining the same interface application clients use. Applications are completely isolated from the specifics of the transport layer implementation. JavaGroups can, for example, provide transport layer plugins to other group communication toolkits while application clients remain unaware of that.

An instance of a `Channel` represents a process group member. The first member of a group instantiated becomes the group coordinator.

4 Totem and JavaGroups (differences and similarities)

Rather than implementing various message ordering protocol layers as done in layered protocol stack approach of JavaGroups, Totem organizes process group members into a logical ring and provides totally ordered message delivery. Totem has only three protocol layers: membership layer, total order layer, and recovery layer.

Membership layer at the bottom of the stack ensures consensus on membership configuration among configuration members and termination (every member installs same configuration with an agreed membership within bounded time, unless it fails). The membership protocol also recovers token and generates token and recovers messages that had not been received by some members when failure occurs.

Above the membership layer, Totem has a protocol layer that implements reliable delivery, total ordering and flow control. Recovery layer is the topmost layer in Totem implementing recovery due to member failure, link failure and token loss failure.

Totem's goal was to provide highthroughput, low latency and low overhead totally ordered protocol that can then be used to provide weaker causal and FIFO order.

The adaptation of Totem total order protocol to JavaGroups is done in the following steps. Firstly, a somewhat complex four state finite state machine Totem membership protocol is not adapted at all but the current JavaGroups GMS protocol layer is used to generate consistent membership views. Secondly, since Totem's membership protocol includes elements of failure detection that have already been implemented in JavaGroups, failure detector layer from JavaGroups is used. Finally, the recovery protocol has been changed.

The main similarity between Totem and JavaGroups' TOTAL_TOKEN layer is the total ordering protocol. Therefore, JavaGroups essentially throws away Totem's recovery and membership protocol and uses its own GMS and failure detection, while adapting Totem's middle layer implementing reliable delivery, total ordering, and flow control. In the following section we will look closely at details of JavaGroups total ordering protocol.

5 JavaGroups' total token layer

Total order requires that all the messages in the same group membership are delivered in a total order (i.e., if message *m1* was delivered prior to message *m2* at process group member p then all other process group members deliver *m1* prior to *m2*). Programming of application clients is significantly easier if the order provided is total rather than causal as well as causal rather then FIFO order.

TOTAL_TOKEN layer constructs a logical token passing ring among group member nodes. The token controls access to the broadcast medium and thus only the member who currently owns a token can transmit on that medium. Whenever member owns a token it can send multiple message subject to flow control constraints. If no member has any messages to broadcast, token continues to circulate. Each message broadcasted in the ring has the unique sequence deduced from the token field thus being a unique sequence for the whole ring.

5.1 Architecture

As previously mentioned in section 3, each JavaGroups protocol layer has two threads servicing `up_queue` and `down_queue` respectively. Besides those two threads total token layer has a third, token transmitter, thread that accepts token from the previous member in the logical ring, executes the total ordering protocol (see section 5.2) and forwards token to next member in the ring. Token transmitter thread has a token timer that is an estimate of token arrival within a constant (see section 5.4 below).

JavaGroups messages are abstracted in `Message` class. Each message has sender and recipient address, header stack and byte contents of the message (payload). Total header, bearing a unique sequence number, is added to each message that is sent through total token layer.

Token contains the following fields:

- `tokenSeq`: a sequence number of token incremented when each member receives token.
- `seq`: the largest sequence number of any message that has been broadcasted on the ring.
- `aru`: all received up to represents a sequence number of a message *m* and denotes that all members have received all message less then or equal to sequence number of message *m*. aru field represents acknowledgement mechanism.
- `retransmissionList`: contains sequence numbers of messages requested for retransmission by some group member.
- `aruId`: address of the member that last set the value of aru less than seq.
- `type`: regular or recovery token
- `broadcastCount`: indicates number of messages transmitted in last round.
- `backlog`: total number of all message queued for transmission by all members
- `windowSize`: size of the transmission window

Each process group member represented by `ProtocolStack` abstraction contains instance of TOTAL_TOKEN layer. In turn each instance of TOTAL_TOKEN contains the following variables:

- `myAru`: the sequence number of a message denoting that this member has received all messages up to a message with `myAru` sequence number.
- `newMessageQueue`: messages queued for transmission at this member node.
- `receivedMessagesQueue`: the queue of message received in process group waiting to be delivered to application

5.2 The operational protocol

Upon a receipt of a token, group member updates token's retransmission list. Member traverses its `receivedMessageQueue` and adds any sequences greater than `myAru` and less then highest sequence that are missing from receivedMessageQueue.
After updating token retransmission list, the member queries control flow component for an allowed number of messages to transmit. Member first rebroadcasts any messages that are requested in token's retransmission list and available in its `receivedMessageQueue`. Member node removes sequence of the message it rebroadcasted from the token's retransmission list and decrements number of messages allowed to transmit by the number of messages rebroadcasted.
If number of allowed transmissions is greater than zero and `newMessageQueue` is not empty, member broadcasts allowed number of message from `newMessageQueue`. The broadcasted messages are copied into `receivedMessageQueue` as well as sent to the broadcast medium. For each new message transmitted the member increments token's seq (highest sequence) field. Whenever the `seq` and `aru` fields are equal, the member increments `aru`, `seq` and `myAru` in the same step and sets `aruId` field to null value.
After broadcasting, member compares value of `myAru` field with token's `aru` field. There are three cases in which member sets the value of its `myAru` field to token's `aru` field and sets `aruId` to its address:

- `myAru` is less then token's `aru`
- token's `aruId` is equal to the member address
- token's `aruId` is null value.

The first case and last case are intuitive while the second one handles the situation when member has updated `aru` field of the token in the last token rotation and no other member changed the `aru` during the token rotation.
Before forwarding token to the next member, the current member owning the token removes all messages from the receivedMessageQueue determined to be received by all members. That value is the minimum of token `aru` and last round token `aru`.

5.3 The flow control algorithm

Sliding window flowcontrol with positive acknowledgements algorithms have performed excellent in point-to-point communication. However, in broadcast environment use of positive acknowledgements has to be avoided due to acknowledgement explosion.
The basic characteristic of totally ordered reliable broadcast protocol is that the rate of message broadcast can not exceed the rate at which slowest member can receive messages.
JavaGroups total token layer reuses a very simple and effective flow control algorithm introduced in Totem.
The flow control mechanism depends on the three token fields: `broadcastCount`,`windowSize` and `backlog`. `broadcastCount` and `backlog` represent number of messages broadcasted and total sum of all local node backlogs in last token rotation respectively.
Each total token layer has an instance of `RingNodeFlowControl` class. The variables of `RingNodeFlowControl` are:

- `previousBacklog`: the number of messages in newMessageQueue at the time when the member forwarded token in previous round
- `currentBacklog`: the number of messages in newMessageQueue at the current token rotation

In one token rotation, each member is allowed to broadcast n messages. The value n is taken to be the minimum of the three following values:

- `maxMessages`: This value is a constant or some other simple bound
- `windowSize - broadcastCount`: The allowed number of transmitted messages for a certain member must not exceed window size minus the number of messages broadcasted in previous round.
- `windowSize*currentBacklog / (backlog+ currentBacklog - previousBacklog)`: The allowed number of transmitted messages for a certain member must not exceed its fair share of the window, based on a ratio of its backlog and the sum of all other members backlogs.

6 The recovery protocol

Protocol described in the previous section concerned the normal execution of the total ordering protocol without any token loss, member, or link failures. In this section we focus on triggers that cause transition to recovery state, the recovery protocol itself and transition back to operational state.
There are two triggers for transition to recovery state: failure suspect message and token timeout. Both failure detection suspect message and token timeout have to occur (during a view duration) in order to move a process group into recovery state.
Token timeout represents smoothed token ring round trip time function based on previous ring round trip times. Upon transmitting token to the next member in ring, member start token reception timeout. Failure detection layer emits suspect message when it considers that a certain member has crashed. In order to understand why both of these triggers have to occur to initiate recovery consider the case of failure suspect message and token reception timeout ocurring individually.
If only the timer is triggered then there must have been a significant delay spike in token transmission due to any asynchronous model reasons. Process group is not transitioned to recovery state since failure detection layer has announced no crashes and channel links inbetween the ring members are reliable. Therefore, the token will eventually arrive at group member. On the other hand if there has been a member crash announcement but no timeout has been triggered then the member must have crashed while not in possession of a token and hence token will continue to circulate.
If token is not received within token ring round trip time and suspect message has been received, members of the process group are moved to recovery state. The transition to recovery at each member level is bounded by token ring round trip time. Upon transition to recovery state, coordinator sends multiple reliable unicasts message requesting each ring member to report its `myAru` value. When all replies are received, a response list of al `myAru` values is sorted and transformed into a set while dropping the lowest `myAru` value. For example, received response list [4,4,5,6,7,7,8] is transformed into [5,6,7,8] thus not including the lowest value 4.
The objective of the recovery protocol is to have each member receive all messages up to maximum sequence value m from the response list, thus satisfying virtual synchrony properties. Note that if all members report the same `myAru` values, then virtual synchrony is satisfied (since all surviving members have seen the same set of messages) and we can immediately inject operational token that will enable instalment of the new view.
Otherwise, a constructed set S of all `myAru` values represent sequence IDs of messages that have to be received by all members prior to installing new view thus satisfying virtual synchrony properties.

A transitional view, visible only to TOTAL_TOKEN layer is then installed on the ring by a coordinator. Multiple reliable unicast messages are used again. A transitional view is deduced from current view by excluding suspected members. Coordinator creates a recovery token by appending the set S of sequence ids to token retransmission request list. Recovery token is then inserted into transitional ring view. Upon reception of recovery token, ring members are not allowed to transmit any additional new messages but only to retransmit messages from the specified token retransmission request list.
When all members detect that they have received all messages up to sequence value m, the next member that first receives a recovery token converts it to operational token and normal operational state is restored in all nodes. If token is lost during recovery stage, recovery protocol is restarted.

7 Performance

The performance measurements were made using four Sun ultra 10 (300 Mhz CPU, 256 MB) connected by 100 Mb Ethernet.

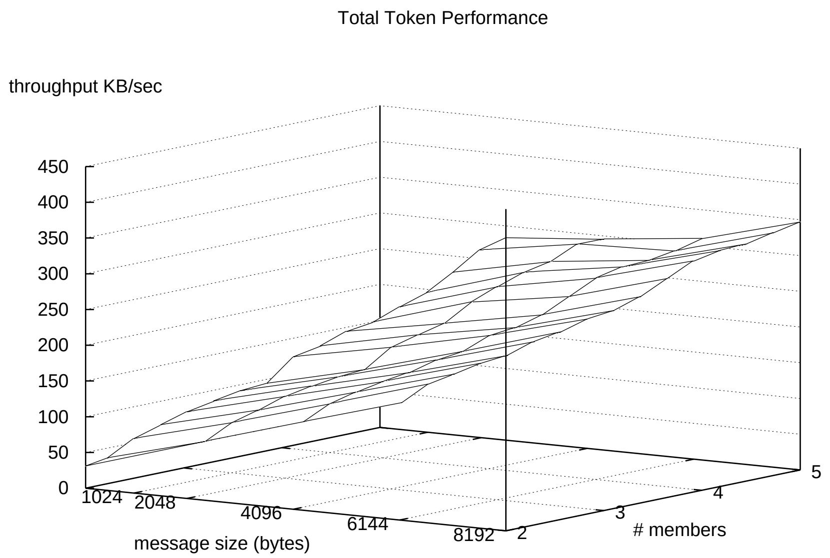

Figure 1: JavaGroups TOTAL_ TOKEN performance

Performance measurements were conducted in the environment with minimum extraneous load on processors and Ethernet. Figure 1 shows performance measurements recorded at each member node. Throughput measurements for a member node were averaged over 10 second period. Each member was broadcasting when performance measurements were taken.

8 Conclusion

JavaGroups total token layer implements total ordering protocol first introduced in Totem reliable group communication toolkit. Distributed applications using JavaGroups now have an option of choosing either sequencer or token passing based total ordering protocol. JavaGroups total token layer implements Totem's total ordering protocol while throwing away

Totem's group membership and recovery layers. Existing GMS and failure detection JavaGroups layers are used in place of Totem membership layer. A completely new and simple recovery mechanism has been introduced.

References

[AMMS+95] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, 1995.

[Ban98] Bela Ban. Design and Implementation of a Reliable Group Communication Toolkit for Java. http://www.cs.cornell.edu/home/bba/Coots.ps.gz, 1998.

[BSS91] K. P. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, 1991.

[KT96] M. Frans Kaashoek and Andrew S. Tanenbaum. An evaluation of the amoeba group communication system. In *International Conference on Distributed Computing Systems*, pages 436–448, 1996.

[vRBM96] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4):76–83, 1996.