Design Document

Kurt Rudolph
Md Tanvir Amin

October 8, 2011

Contents

Chapter 1

Introduction

This document describes our implementation of a Distributed, Reliable, and Causally ordered chat protocol. The program is designed to work between a set of processes, communication is achived via multicasting of the messages, which in turn uses IP-Unicast. Groups are assumed to be closed, and all process joins the chat before the first message is sent.

An `n x n` heartbeating system is incorporated into the protocol to detect process failures. There is no distributed group management of dynamic join and exit of the participants, but the system can continue and withstand as many process failures as possible.

Our protocol is highly efficient both in terms of message transmission and in terms of reliability. The channel is unreliable, but we do not use the `R_multicast` algorithm, which requires $|g|$ transmissions per process for each message, which is clearly inefficient. Instead, our protocol limits the number of message transmissions per process close to 1. As the channel is unreliable, sometimes retransmissions are necessary. Processes understand the requirement of necessary retransmission both using the piggybacked acknowledgments in heartbeats, and the piggybacked timestamps in data messages. When a process learns about one or more missing messages (which is quite fast), it waits a while. If the message do not arrive within that period, the process then randomly selects (uniform distribution) a currently alive process which had received that message. It then sends the negative acknowledgment for that particular message. Such a protocol can easily withstand arbitrary process failures, and arbitrary message dropping as we are not dependent on the original sender.

Chapter 2

Architecture

Our protocol makes uses `B_Multicast` algorithm as baseline. It doesn't use expensive `R_Multicast` algorithm, rather reliability is achieved throgugh negative acknowledgments, heartbeat messages, and message retransmissions. We make use of the piggybacked acknowledgment algrithm as described in textbook. Figure 12.16 in the text describing a causal ordering vector timestamp algorithm and the description of reliable multicasting found in chapter 12. The processes are indexed according to the order in which they enter the group and a accessed via `my_id_index`.

To ensure causal ordering and reliability, the program contains a hold back queue which is implemented as linked list. On the other hand to facilitate retransmissions and various checking of the messages, a bucket for each participant is necessary. The end result is an orithogonal linked list, which is the main data structure the program relies upon.

2.1 Message Types

Only way of communication between processes is message passing. To facilitate for various control operations, we have 3 types of messages. The first byte of all messages received will be designated for defining the message type, allowing for up to 256 message types. The following are the message types defined for the program and their descriptions.

1. **00000000 Ping/Heartbeat** An echo request of the according message by the recipient back to the sender of the message. The format of the heartbeat message is shown below:
   - 1 byte for message type.
   - `mcast_num_members * sizeof(int)` bytes for acknowledgment.
2. **00000001 Data** A message containing data to be displayed to the console. Each of the participants maintain sequence number of the messages which is included with the timestamp. The format of a date message is shown below:

- 1 byte to specify message type.
- `sizeof(int)` bytes to specify message source.
- `mcast_num_members * sizeof(int)` bytes for timestamp.
- Null terminated string as payload (message).

3. **00000010 Retransmission Request / NAK** NAK is a request to retransmit a specific message (according to it's *vector timestamp* as described in section 2.2)) that originated from a specific process. If this retransmission fails or the message is dropped, eventually the requesting participant will time out and request the message from another participant. Format of the retransmission request is shown below:

   - 1 byte for message type
   - `sizeof(int)` bytes to specify the sequence number of the message being requested to restranmit.
   - `sizeof(int)` bytes to specify the original sender of that message.

2.2 Vector Timestamp

Proceeding one byte message type indicator within the header of the each message is the vector timestamp. The vector timestamp consists of an integer array with length equivalent to the number of processes within the group, indicated by global variable `mcast_num_members`.

The vector indices will be incremented upon arrival of new messages from th according sender. For example: suppose process with `my_id_index = 0` sends a multicast message to a group containing four processes including it's self. Initially, `timestamp = [0,0,0,0]` for all processes, upon process with `my_id_index = 0` sending the initial message the timestamp included with the message is `[1,0,0,0]`, where as the internal vector timestamp of each process (including the process with process with `my_id_index = 0`) remains `timestamp = [0,0,0,0]` until the arrival of the initial message. Upon the arrival of message with `[1,0,0,0]` the internal vector timestamps of each process will be updated accordingly.

Should a message fail to reach a given recipient, the according recipient will be detected such a failure by examining the timestamps of future messages (hence utilizing piggyback acknowledgment) and requesting their retransmission accordingly.

2.3 Causal Ordering

Our implementation for causal ordering is based on Birman's algorithm (Figure 12.16 in the text). An hold back queue is used to keep track of the messages. Each process p_i, upon

receiving a message from p_j, waits until (a) it has delivered any earlier message sent by p_j, and (b) it has delivered any message that p_j had delivered at the time it multicast the message. High level description of the multicast algorithm is as following:

Algorithm for group member $p_i (i = 1, 2, \ldots, N)$
On initialization
$V_i^g[j] := 0 (j = 1, 2, \ldots, N)$;
To CO − multicast message m to group g
$V_i^g[i] + 1$;
$B - multicast(g, < V_i^g, m >)$;
On $B - deliver(< V_j^g, m >)$ *from* $p_j (j \neq i)$, *with* $g = group(m)$
place $< V_j^g, m >$ *in hold − back queue*;
wait until $V_j^G[j] = V_i^g[j] + 1$ *and* $V_j^g[k] (k \neq j)$;
$Co - deliver\ m$; //after removing it from the hold-back queue
$V_i^g := V_i^g[j] + 1$;

2.4 Reliable Multicast

The protocol ensures conditions of reliable multicast; integrity, validity, and agreement. The program also combines reliable multicast with piggy backed acknowledgment and negative acknowledgment for efficiency purposes. Here we show how the protocol satisfies these properties.

1. **Integrity** is satisfied because the program keeps a track of the messages already received, queued, and delivered. Each process delivers a message m at most once. Moreover, as sequence number is maintained, and messages from each process go to individual buckets, there is efficient duplication checking upon reception of a message.

2. **Validity** is satisfied because each participant delivers their own messages. In fact, processes do not send their own message to IP layer, rather they directly add it to the queue.

3. **Agreement** is satisfied because if a process delivers a message, eventually all the other processes deliver it. A specific message may not reach a participant due to arbitrary message dropping in the channel or due to process failures. A participant can easily learn about the missing messages checking the timestamps transmitted with each data message, and the acknowledgments piggybacked with each heartbeat message. Thus it can request for a retransmission.

   Our protocol limits the number of message transmissions per process close to 1. When a process learns about one or more missing messages (which is quite fast), it waits a while. If the message do not arrive within that period, the process then randomly

selects (uniform distribution) a currently alive process (apart from itself) which had received that message. It then sends the process a retransmission request for that particular message. Such a protocol can easily withstand arbitrary process failures, and arbitrary message dropping as we are not dependent on the original sender.

As a process is selected randomly from an uniform distribution, the retransmission requests are automatically load balanced. With high probability our protocol reaches the reliability guarantees as provided by expensive `R_Multicast` algorithm, while keeping the number of message transmission low.

2.5 Participant Buckets and Hold-Back Queue

In order to properly display messages in a causal ordering sequence and serve negative acknowledgment requests, a bucket structure (participant buckets) and a hold-back queue is utilized to manage delivery of messages to the console. In figure 2.1 we have illustrated this data structure.

2.5.1 Participant Buckets

Their is a participant bucket for each of the processes. An array contains the pointers to each of the buckets. The pointers point to a doubly linked list containing one node for each message received from the according process. The nodes are ordered from oldest to youngest in compliance with the casual ordering. The receiving thread places the messages in the linked list and a participant thread manages it. As this data structure can be accessed by multiple threads, there is a lock for each of the buckets.

2.5.2 Participant Threads

A participant thread is spawned to manage each bucket. The thread performs the following tasks:

- Identify missing messages and send retransmission requests
- Free memory of message one it has been identified that all other processes in the group have received them.

2.6 Failure Detection

An independent thread is spawned for the purpose of failure detection. A heartbeat is sent to each of the processes every `10*MAXDELAY` time period and after several time periods pass where no heartbeat has been received from a processes, the processes is assumed to have failed. The developed system can proceed correctly even if some of the processes fail.

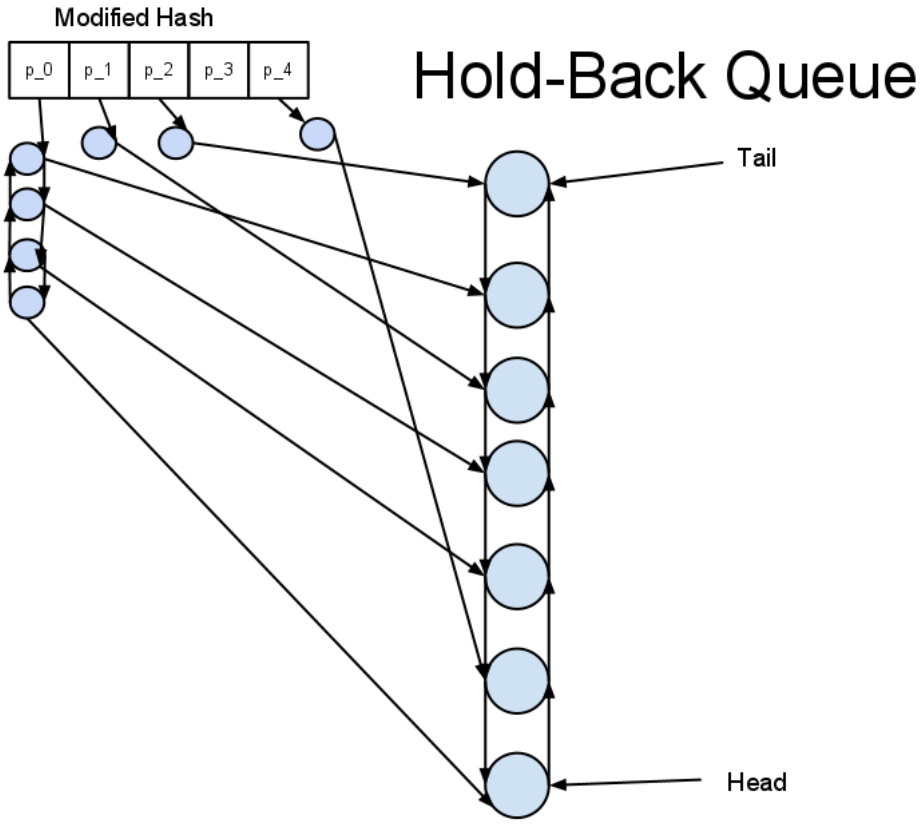

Figure 2.1: Data structure for the hold back queue and participant buckets

Chapter 3

Development and Testing

3.1 Implementation

We have implemented our solution in C language as required. As there are multiple threads, we have added locks to ensure the consistency of the data structures. Our program achieves the desired task without crashing or locking resources. In total there are 3 + `mcast_num_members` threads. There is one thread for each participant, 1 sender thread (main thread already provided), 1 receiver thread (already provided), and a heartbeating thread.

The main data structure of the program are the participant buckets and the hold back queue linked orthogonally to each other. We have used the linux kernel linked list as a library (`list.h`) which implements a doubly circular linked list efficiently.

Inside the prgoram, each of the incomign data messages are stored with the following structure:

- Source index
- Timestamp vector
- Sum of the timestamp
- Payload (actual message to display)
- Pointers to the next and previous messages in the hold back queue
- Pointers to the next and previous messages in its corresponding bucket

Upon receiving a message the message handler is called. The message handler is utilized for sending the various message types to the appropriate receivers.

3.2 Testing

No formal test suit was implemented for this assignment. The lack of which greatly slowed development. So, after adding each features, we performed regression tests, and functionality tests manually. This verification procedure was rather tedious and time consuming. In future MPs a test suit will accompany the program.

3.3 Known Issues

The program does contain problems related to memory leaks but that is due to the lack of a callback function in the `chat.c` code. The `main()` function is in `chat.c` and we are not allowed to change it. The fundamental framework in `chat.c` and `unicast.c` accounts for a callback function `mcast_join` whenever a new memeber is added, so we could reallocate the data strucutres dynamically and properly. However, it lacks a `cleanup()` callback function, which was supposed to be called when the user types `/quit` in the chat window.

Chapter 4

Conclusion and Future works

The general concepts behind the MP were not particularly difficult. However, implementing the program in C proved to be rather difficult and more time consuming than anticipated.

The program oers a solid base for work with GPU computing. The vector timestamp comparisons are ideal operations to be computed on a GPU. Although the program only runs on a single machine, a networked implimentation oers potential for ncorporating feature of of NVIDIA GPU direct as well.