CS425 MP2 report

Mingwei Hu (mhu9), Martin Liu (liu242)

April 2, 2014

Implementation of multicast primitives

First, we use reliable_multicast as the basic primitive for our protocol. It is implemented using reliable_unicast which includes unicast_send(ip, port, mesg) and unicast_recv(port). Both of them use UDP sockets to send packets. The job of unicast_send is just to send a piece of message to a given address and also simulate the packet drop. The unicast_send(ip, port, mesg) is called by a call back function called sender_cb which is associated with a child thread spawned by the multicast thread. The reliable unicast functionality is achieved by sending back an "ACK" after the receiver receives the message. A time-out is assigned to each user-if the user doesn't receive the "ACK" within this limit the it sends the message again. The time-out and "ACK" functionality is achieved by using the function "pthread_cond_timedwait(cond, lock timeToWait)" in the sender_cb function so that the thread is either waken up by a "ACK" signal or the time-out signal.

Unicast_receive, on the other hand, keeps receiving incoming messages and spawns a child thread for each message. A call back function - mesg_processor_cb(void* arg) is called in each of these child threads to process its message. This function then called process_message function to parse the message and determines the type of the message (whether it is a ack or a normal message). It then accordingly calls other functions based on the message type. If it is an ack it will send a pthread_cond_signal to the thread being blocked. If it is a normal message, it will prepare a ack message and unicast_send it back to the sender.

Implementation of losses and delay

Both losses and delay are implemented using a random number generator rand(). Losses are determined in the unicast_send() function. A generated random number "dr" is compared with the meanDropRate. And if dr < meanDropRate, then packet is just not sent before we exit the function. The delay is implemented in the mesg_processor_cb(void* arg) call back function (which is called by threads spawned by the unicast_receive). Similarly, the delay is just set to $rand()\%(2 * user- > meanDelay)$.

Implementation of causal and total ordering

Causal ordering is implemented using the timestamps approach. Specifically, each user struct maintains a vector of int (vector$< int >$ timestamps) that records the timestamps of all the users. Every time the R_multicast() function is called by user i, $timestamps_i[i]$ is incremented by one. After that, the timestamps is formatted as a string and prepended to the message content and then is multicasted to the group. When $user_i$ receives a message from $user_j$, it first pushes this message to a special data structure called causal_holdBackQueue (which is essentially a unordered_set). It then iterates through the queue to see if any message is deliverable by checking with the condition : $timestamps_j[j] == timestamps_i[j] + 1$ && $timestamps_j[k] == timestamps_i[k] (k \neq j)$. if the condition is true, it delivers the message and erases it from the queue. Then it updates its timestamps again by setting element j to be $timestamps_i[j] = timestamps_i[j] + 1$. It then goes back to the beginning of the queue and does the checking procedure again until no message is deliverable.

Total ordering is implemented using the ISIS approach. A priority queue, which is implemented using a map¡key, mesg, comp¿, is maintained for each user struct. The queue is sorted by the priority of each messages in ascending order. And a comp function pointer is provided such that the queue automatically sorts itself. Sender first multicasts the message to the group. On receiving the message for the first time, receivers should attach a proposed priority to the message (their biggest priority seen so far + 1) and mark it as undeliverable and then push it to the queue. After that all the receivers respond to the sender with their proposed priorities, and the sender will get the maximum and multicasts the maximum along with the original message. On receiving the message for the second time, receivers will iterate the queue, find the original message, mark it as deliverable, change its priority to the maximum (if the maximum is bigger the the local maximum seen by the user) and reorder the queue. Finally, receiver checks the head of the queue to see if it is deliverable. If it is, then receiver delivers it (which just prints it to the screen).