Exercise 3 Design

Jiayao Wu, Suyi Liu

November 2016

1 Idea

Use the spread to build a protocol that reliably sends messages between multiple processors in the same spread network in agreed order. We have mcast.c and net_include.h.
In mcast.c, it takes num_of_messages, process_index and num_of_processes as inputs. num_of_messages indicates how many messages this processor needs to send. process_index is a unique number assigned to this processor. num_of_processes indicates how many processors there should be in this membership. Each processor in the end has an output file: process_index.out that has all the received messages in it.

2 Data Structures

Packet Structure:
int type(type -1 means regular message; type 0 means finished sending for this round; type 1 means finished sending all num_of_messages; type 3 means this processor will send messages; type 4 means this processor does not send messages)
int machineindex
int packetindex
int randomnumber
1200 bytes additional message

3 Algorithms

Step 1: Join the spread group. Then wait for all other members to join by looking at the membership message. Once all members have joined, then proceed to step 2
Step 2: If num_of_messages is 0, make packet's type to be 4, otherwise, make packet's type to be 3. Then multicast only this packet.
Step 3: Wait and receive all the packets sent from step 2 to determine how many processors actually send messages. Let's denote this number as n (in code, we

used all_received for this number). We need this number for the termination requirement. Then go to step 4.
Step 4: There are two cases:
If num_of_messages is 0, then keep receiving messages until have received all messages from n processors. Each time when receiving a packet, write its information to the file. Specifically, the outer while loop (all_received != 0) makes sure the processor receives all num_of_messages from other processors, because whenever it receives a packet whose type is 1, decrement all_received and stoprecv. The inner stoprecv is always set to be equal to all_received because if one process p has finished sending all its packets, our process that the program is running on has no need to wait for p's message. This ensures our stoprecv to successfully decrease to 0 in each round. The inner while loop (stoprecv != 0) makes sure the processor receives all messages for this round, because whenever it receives a packet whose type is 0, decrement stoprecv.
If num_of_messages is not 0, then while there are still more packets to send or there are more packets to read:

1. if there are still more packets to send: first send a burst of messages and mark packet's type as -1, 0 or 1 accordingly. If there are no more packets to send, directly got to 2.

2. receive all messages for this specific round (same logic as stoprecv != 0 described above) and decrement all_received once it receives a packet whose type is 1.

Step 5: When both conditions are satisfied: the process has sent all the packets and the process has received all the packets, close the file, print the duration of time and exit.
The burst size we chose is 600 (reason is in analysis section). This algorithm ensures flow control because the processor sends a burst for this round, then receives all packets for this round. So there won't be too many packets stacked somewhere unread and each processor won't send too many packets in a round. As described in the algorithm, termination is taken care of by using while loops.

4 Performance Analysis

We tested on 8 machines with 6 of them sending 100000 packets each while 2 of them sending no packets with different burst sizes of 50, 80, 100, 110, 120, 150, 200, 300, 400, 450, 500, 550, 650, 700 and 800 packets per round.
And we found the performance are shown to be best between burst size of 300-700. Among them, burst size of 600 has the best result of 15.806104 seconds averaged.
Occasionally, the spread fails to connect for some reason such as we entered the wrong command, or the network was not stable. Especially, around 7:30pm on Nov 7th, burst size of 600 will lead to spread failure. We believe this is because too many people are running the program
We didn't test burst sizes of larger than 800 since SP connection was lost when we were testing on 800 packets per round. This may be because spread has too

much packets in the buffer.

The reason why the performance was not as good when burst size is rather small is maybe because the main while loop of sending a burst and receiving bursts is executed too much times, so that the flag is checked too frequently, thus reducing the speed.

The reason why the performance was good with burst sizes such as 400, 500, or 600 is because they reduced number of rounds in total while not exceeding maximum packets that spread can buffer.

Below is the graph of performance result with different parameters:(The time varies slightly, but in general they are close to 17 seconds)