Radhir Kothuri
kothuri2

MP 1 README

Simulating Network Delay:

In order to simulate network delay, I had a separate sendMessage() function that would each thread to multicast to a different process would run. Once I parse the command line input and the config file for min_delay and max_delay, I used python's random.randrange(min_delay, max_delay) function to pick a random number between the range, and then the thread slept for that amount of time. As soon as that particular thread's message "woke up", it would send the message to the other process's server. This way I print out that my process sent the message immediately, and then the thread.sleep() is used to simulate the potential physical network delay between servers. The receiving server then receives the message, and applies the appropriate ordering.

Implementation of Causal Ordering:

Before I could implement causal ordering, I had to setup the underlying architecture of sending messages between multiple processes with the simulation of random network delay. In addition, each time a process sent a message to another process, it would increment it's position in it's vector timestamp. Therefore, the client of the sending process would piggyback this vector timestamp on top of the message that it sends to the server of the receiving process. I actually use pickle (an object serialization library in python) in order to send other metadata along with a message.

Once the receiving server receives the sending process's timestamp, it then compares it's own timestamp with the sent timestamp. Based on this element-wise comparison of the two vectors, there are some variables that I account for: countExact, countLessThan, and countGreater. countExact counts the number of elements in the vectors that differ by exactly one such that the server_element + 1 = client_element. countLessThan accounts for how many elements that are bigger in the server_vector as compared to the client_vector. Finally, countGreater accounts for how many elements that are smaller in the server_vector as compared to the client_vector. Based on these 3 variables, I have 2 general conditions: one condition to check for the happened-before relation, and one condition to check for concurrent events. Therefore, if countExact == 1 and countLessThan == 0, then that must mean there was only one element in the server_vector that was less than the client_vector (and the difference was 1). Therefore, I can acknowledge the receipt of the client's message and don't have to buffer it. The other case: (countGreater >= 1 and countLessThan >= 1) accounts for concurrent events. In this case, I can just take the max of the two timestamps and acknowledge the receipt of the message. In both cases, I have a separate thread that runs that checks for a potential message that can be moved off the buffer. The buffer is a PriorityQueue based on the first index of the vector_timestamp. The conditions to be removed from the buffer are exactly the same as the conditions to accept the message in the first place. Therefore, this is how I have implemented causal ordering.

Implementation of Total Ordering:

In order to implement Total Ordering, I had 2 numbers that each process kept track of: the global sequence number that the sequencer was on, and the number of messages that the each process has seen so far. Therefore if the global_sequence_number != process_seen_so_far_messages + 1, then the message was buffered into a Queue.

The Sequencer was arbitrarily chosen as the process with the lowest ranking id. Therefore whenever another process multicasts, it first unicasts its message to the sequencer. The sequencer then increments the global_sequence_number and multicasts this number along with the message_information. Therefore whenever the other processes start to receive this message, they first check if the sequence_number they received = num_messages_seen + 1. This makes sure that the order that the messages leave from in the sequencer is preserved whenever the other processes retrieve it. This process is very similar to the FIFO strategy implemented in the FIFO Order Multicasting.

Execution Instructions:

Files:

1. basics.py -> runs unicast_send (e.g. send 2 hello), multicast_send and multicast_receive with network delay built in but without any specific ordering
2. fifo.py -> runs FIFO Order Multicasting
3. causal.py -> runs Causally Ordered Multicasting
4. total.py -> runs Totally Ordered Multicasting

In order to run any of the four files, use this format:

python [file_name] [config_file.txt] [process_id]