Reliable Communication Channel over UDP

Praseem Banzal Suvidha Kancharla Gowtham Kaki

Purdue University
{pbanzal,skancher,gkaki}@cs.purdue.edu

Abstract. We describe the architecture and working details of reliable and FIFO network communication channel build over unreliable UDP protocol. This work is done towards fulfilling the requirements of cs505 Distributed Systems course work.

1 Architecture

A reliable communications channel is duplex by nature. At the high-level, the channel is composed of two components - sender and receiver. The sender component accepts a message, which is a string of arbitrary length, converts the message to byte stream followed by fragmentation into UDP datagrams before sending. The sender requires acknowledgement (ACK) from receiever acknowledging the receipt of every datagram. In order to match acknowldgements with datagrams, every datagram is assigned a unique identifier. To facilitate FIFO order at the receiver, unique identifiers are assigned sequentially, hence they are sequence numbers.

Sender maintains an *UNACK_SET* of datagrams that are yet to be acknowledged by the receiver. Considering that network bandwidth and memory at receiver are both finite resources, the size of *UNACK_SET* is bound. If the bound is reached, sender goes to sleep. It periodically wakes up to see if any datagrams in *UNACK_SET* are acknowledged. If there are any such datagrams, it removes them from the set, resends the rest, and fills the gap by sending fresh datagrams, if there are any. Sender repeats this till there are no more datagrams to send.

The task of the receiver component of the channel is to receive and reconstruct the message sent by sender. Since message is fragmented, and fragments have sequence numbers, receiver maintains a *sliding window* of *holes* denoting consecutive sequence numbers of fragments that it is expecting. If a new datagram with sequence number in the expected range (i.e., current window) arrives, it is acknowledged and corresponding *hole* is filled. If the sequence number is lowest of the expected range, the bounds of the range are incremented by one (i.e., window slides to right). If the arrived fragment is the last of the fragments of a message, a flag is set to denote the same. Once such fragment arrives, given that all holes corresponding to previous fragments are filled, the message is reconstructed and handed over to the listener.

2 Implementation

`RChannel` class implements the interface `ReliableChannel`. The constructor of the class accepts local port, destination port and destination IP, and opens a UDP socket on the local port. Sender and receiver components manifest as separate threads of execution implemented by `SenderThread` and `ReceiverThread` classes respectively. Since channel has to continue accepting messages from the user it continues executing the main thread. To dispatch user messages to sender thread, the channel thread uses a `sendBuffer` linked list that is shared between both the threads. Java construct `synchronize` was used to control the exclusive access to the buffer.

The size of payload in each message fragment is restricted to a fixed number, which is 600 unicode characters, or 1200 bytes. Apart from the payload, the message also contains its sequence number, and flags to denote if it is an acknowledgement, or if it is contains the final fragment of a message. The message is implemented by `RMessage` class, which also implements `Serializable` class, so as to enable conversion to byte stream.

The `ReceiverThread` maintains expected sequence number through `RChannel` class, which denotes the starting sequence number (left-most hole) of the *sliding window*. When a datagram with expected sequence number arrives, the expected number is incremented denoting the sliding of the window. `bufferLength` variable denotes the size of the sliding window. Any datagram that falls in the sliding window is acknowledged by sending an `RMessage` with same sequence number as the datagram, but with ACK flag set.

Since ACK itself is a message, the `ReceiverThread` on sender side is also responsible to convey acknowledgements to sender thread. It does so by marking corresponding messages in `sendBuffer` with ACK flag. Therefore, `sendBuffer` is also shared with `ReceiverThread`.

Once the message is reconstructed by `ReceiverThread`, it calls the callback to convey the message to the listener.