Socket Fields

We decided that only four states are needed to manage our Socket - *CLOSED, ESTABLISHED, WRITE_ONLY, READ_ONLY. WRITE_ONLY* and *READ_ONLY* represents the states when the one side of the socket is already closed.

Except state socket holds usual fields needed to handle the connection, namely local and remote addresses, ports and sequence numbers. Since IP addresses are needed both in little-endian (for the Packet construction) and big-endian (for the checksum calculation), we decided to follow the Java convention and keep them in big-endian. Recalculating them to little-endian have not proven to be significant overhead yet, and they can be kept in both forms if it will in the future.

Additionally Socket holds one *Packet* and one *Segment* for sending and receiving these structures. They are described in more detail in “Sending and Receiving Packets”.

remoteEstablished fields informs us whether we are sure that our remote party established connection with us. It is used to recognize resent SYN-ACK segments and reacknowledge them.

closed means that both sides of the connection were closed, a port freed and this *Socket* cannot be used anymore.

Sending and Receiving Packets

We assumed that only five types of TCP segments are valid for our implementation: SYN, SYN-ACK, ACK, FIN and data-carrying segment. Each type of segment has corresponding *deliverXXXSegment()* and *receiveXXXSegment()* method inside *Socket* where *XXX* is replaced by the segment name.

Each *receive* method calls first *receiveSegmentWithTimeout* (except *receiveSynSegment* which blocks until valid SYN is received) and then checks if the the received segment is a valid segment of the given type.

Timeout parameter in *receiveSegmentWithTimeout* means that the method will return after the specified time passes from the last (probably invalid) segment received. That means that as long as some packets are received, this method tries to extract valid TCP segment from them. We found this fair since we probably should not assume packet lost after 1 second, when there is a high network traffic, because it will probably take it more time to arrive to us. On the other hand this create a possibility of other party blocking us with a stream of fake packets. We assumed though that protection against DoS may lie beyond the scope of our simple TCP.

Each *deliver* method sends a segment and waits for a corresponding acknowledgement, using *receiveSegmentWithTimeout* mentioned above.

Segments Representation

To represent a segment *TcpSegment* class was created. To avoid garbage collection overhead every *Socket* holds one *Packet* and one *TcpSegment* instance.

Since segments have to be transformed from/to byte arrays very often, we decided to keep *TcpSegment* as a wrapper around *InfiniteByteBuffer* class. This is just a *ByteBuffer* implementation that automatically extends its size when needed. This allows us to start a connection with small buffers but extend them when the large data transfer is needed.

Port Management

Every instance of *TCP* class holds *BitSet usedPorts*. Initially only first 1024 ports are set as used, because they corresponds to the well-knows ports most probably really used by the system. When a new *Socket* is created, *usedPorts* is checked either for availability of the given port or for the first free one. When the connection is closed by both sites, that is the state of the *Socket* is *CLOSED* again, the port is freed and the corresponding bit in *usedPorts* is cleared.

We are aware that there is still possibility of clash with other services using ports beyond 1023, but we assume that the real TCP service would be informed of all services bound.

Initial Sequence Number Generation

Since we implemented only simplified version of TCP, we decided just to use *Random.nextInt()* to generate initial sequence number. Since we do not support multiplexing the chance of duplicated sequence numbers seems to be negligible.