Content

---

1. Format of Packet

Each packet contains two parts: the header and the data, we abstract the packet to a class `Packet`, this class encapsulate the header fields and data, in addition, this class has a field that records when the packet was sent. The class provide a method `pack()` to serialize the packet instance into a bytes array in the packet format below.

In order to convert and parse the header fields of received bytes, we provided a function `parse_packet` which receives a byte array and parse the fields, store them into a instance of `Packet` class.

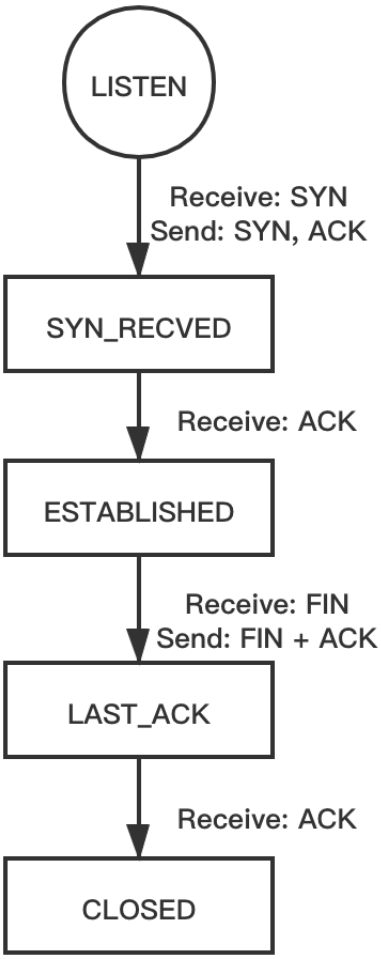
2. Design of Server

2.1 States transition

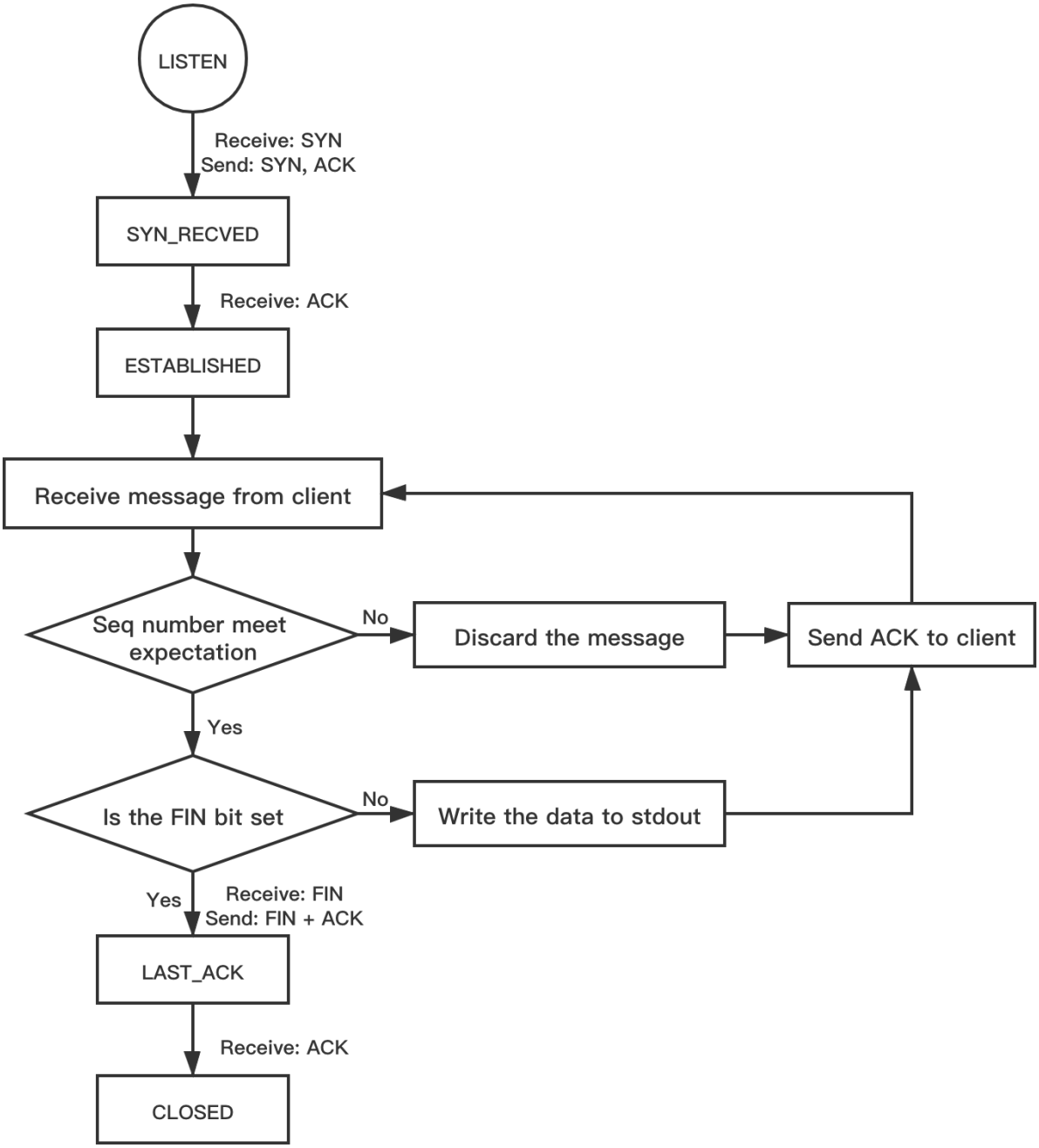
The server has five states: `LISTEN`, `SYN_RECVED`, `ESTABLISHED`, `LAST_ACK`, `CLOSED`.

The server begins with `LISTEN` state, and once it receives a `SYN` packet, it reply a `SYN+ACK` packet and transform to `SYN_RECVED` state, wait for the ack from client. When the server receives the confirmation from client, the connection is established, and then the data can be transmitted.

When the client finishes transmitting the data, the client will send a `FIN` message requesting to close the connection, and if the server receives this FIN message, it will send back a `FIN+ACK` message and transform to `LAST_ACK` state. When the server receives the last ACK from the client, the server will close the connection and transfrom to the `CLOSED` state.



2.2 Processing Logic



The processing logic of server is quit simple.

Firstly, since the server does not need to transmit any data to the client in addition to SYN messages and FIN messages(both only take one logical byte) and ACK (ACK does not consume sequence number), we do not need to consider the problem of sequence number overflow and cycle.

In addition, when the connection is established, the server cyclically receives the message, and then checks whether the sequence number of the message meets the expectation. If it does not meet the expectation, it directly discards it and sends the `ACK` message. If it meets the expectation, then checks whether the message is FIN, if it is, transform to `LAST_ACK` state, waiting for the client's `ACK` message, if it is not a FIN message, the data carried in the message is written to stdout and also send a `ACK` message to client.

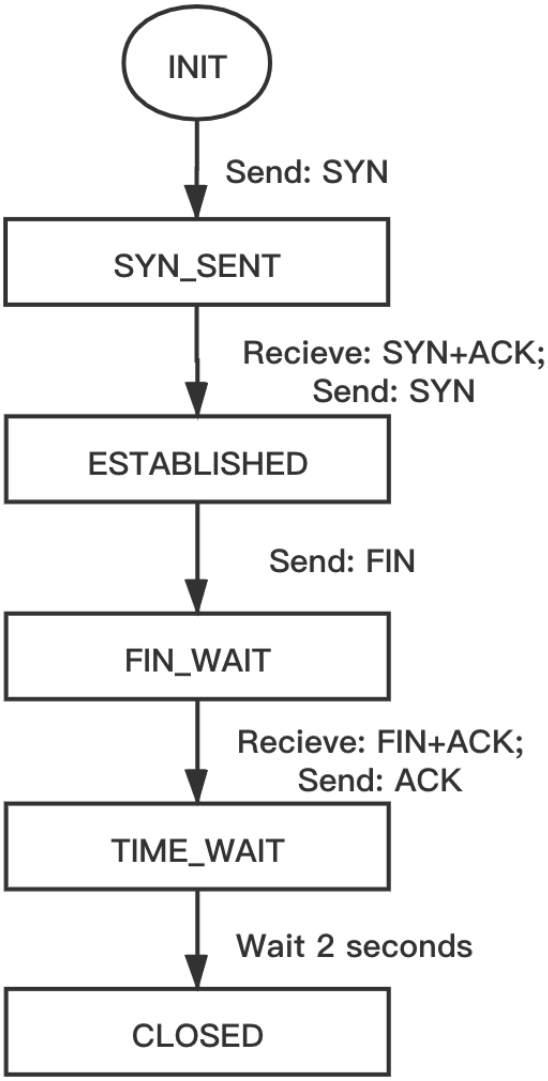
3. Design of Client

3.1 state transition

The client has six states: INIT, SYN_SENT, ESTABLISHED, FIN_WAIT, TIME_WAIT, CLOSED.

When the client is started, it will send a SYN message to the server to request connection, and switch to the SYN_SENT state, wait for the server's response. After receiving the SYN+ACK message from the server, it transform to the ESTABLISHED state, and the connection is established. Then the client can send data to the server.

When the client finishes sending the data, it will send a FIN message to the server to request to close the connection, and transform to the FIN_WAIT state, waiting for the server to respond. After receiving the FIN+ACK message from the server, it sends an ACK message to the server, and switch to the TIME_WAIT state, and exits after waiting for two seconds.



3.2 processing logic

Data structures:

- send_queue: store the packets that has been sent but not yet confirmed
- send_window: the size of data in send_queue

We use a loop to process the data transmission. In each loop, we first wait to receive the confirmation from the server. If the response from the server is not received after the timeout, the retransmission is triggered, and the congestion window size and the slow start threshold are updated.Otherwise, the confirmed message will be removed from send_queue, and the congestion window size will be updated.

Then the client read data from stdin, send it to the server, and append the sent message to send_queue, until the size of unconfirmed data reaches the congestion window size, and enter the next loop.

It is worth mentioning that the size of the sending window must be smaller than the maximum message sequence number, otherwise data with the same sequence number will appear in the sending queue, causing confusion.

When the size of data in send queue is greater than the maximum sequence number, sequence number wraparound will occur. Due to the present of sequence number wraparound, when we receive an ACK message, we cannot simply compare the acknowledge number of the ACK message with the sequence number of the sent message to determine which messages are confirmed (cumulative confirmation).

Our solution is to first check whether a wraparound has occurred. Here we use a clever approach, that is, if the message sequence number at the head of the send queue is greater than the message sequence number at the end of the queue, then wraparound occurs.

If no wraparound occurs, the message with the sequence number less than the confirmation number can be removed from send queue.Otherwise, we divide the message sequence number into the following intervals: [0, s_{tail}], [s_{tail}, s_{head}], [s_{head}, MAX_SEQ], (s_{head}, s_{tail} is the sequence number of packet in the head and end of the send queue)

- if the ack number is in [s_{tail}, s_{head}], all the packet in send queue is confirmed
- if the ack number is in [0, s_{tail}], all the packet in the send queue with packet number less than MAX_SEQ is confirmed
- if the ack number is in [s_{head}, MAX_SEQ], all the packet in the send queue with packet number less than ack number is confirmed

We then follow the rules above, remove all the confirmed packet from send queue.