Reliable Transport Protocol Design Report

CS 3251 B

Keegan Nesbitt

Tyler Litrel

Summary of Changes

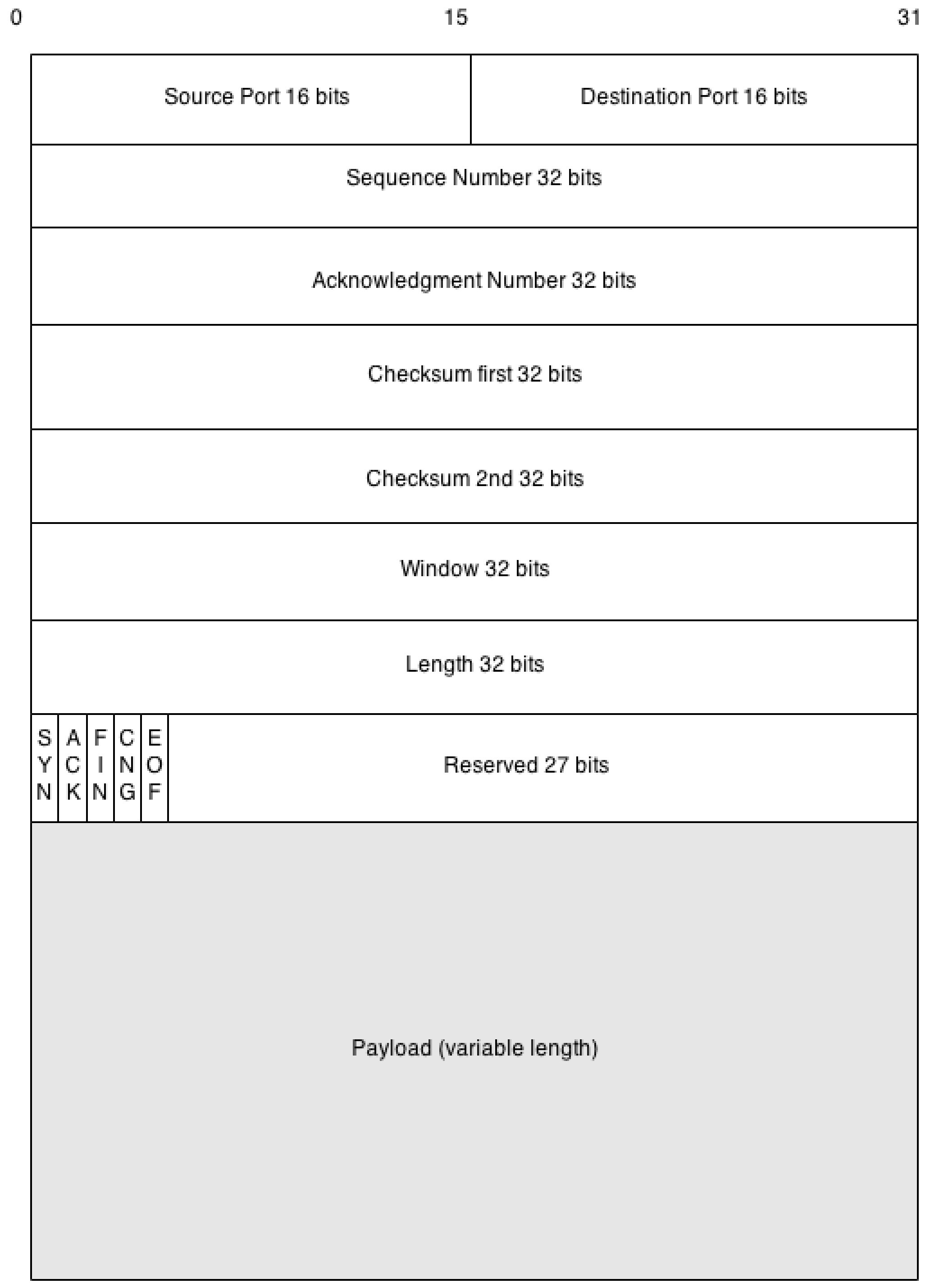
Our original protocol design detailed a pipelined protocol. However, the RTP protocol we implemented utilizes stop-and-wait instead. We also added a bit to the header to indicate the end of a stream of messages. Other changes to the protocol design document include minor changes to the parameters of our functions.

High-Level Description

Our Reliable Transport Protocol will be connection-oriented and implement the majority of the features discussed in the ARQ protocols. Packets will be sent in a stop and wait fashion utilizing selective repeat. Lost packets will be detected by the absence of an acknowledgement from the recipient. After a specified period of time, the sender will resend the lost packet to the recipient. Corrupt packets will be detected through the use of a checksum. Our protocol's checksum will utilize the first 64 bits of a SHA-2 256-bit hash over the header and payload of the packet, instead of TCP's implementation of simply summing the bits in the packet. Corrupted packets will be handled in the same way as lost packets; an acknowledgement will not be sent if the packet is not valid. Out of order packets will be handled by ignoring the out of order packet. Because the sender did not receive an acknowledgement for the missing packets, they will be re-sent after a specified timeout value has elapsed. If the recipient receives duplicate packets, the duplicate packets will not be saved and simply be discarded by the recipient. Our protocol will be able to handle bi-directional data transfers between the two connected hosts. The packets (including the acknowledgements) sent from Host A to Host B will be no different than those packets sent from Host B to Host A. This is similar to the approach implemented in TCP. In addition, we will be implementing a connection establishment approach that will prevent denial-of-service attacks cause by a flood of SYN messages being sent to the server, which causes resources to be allocated on the server.

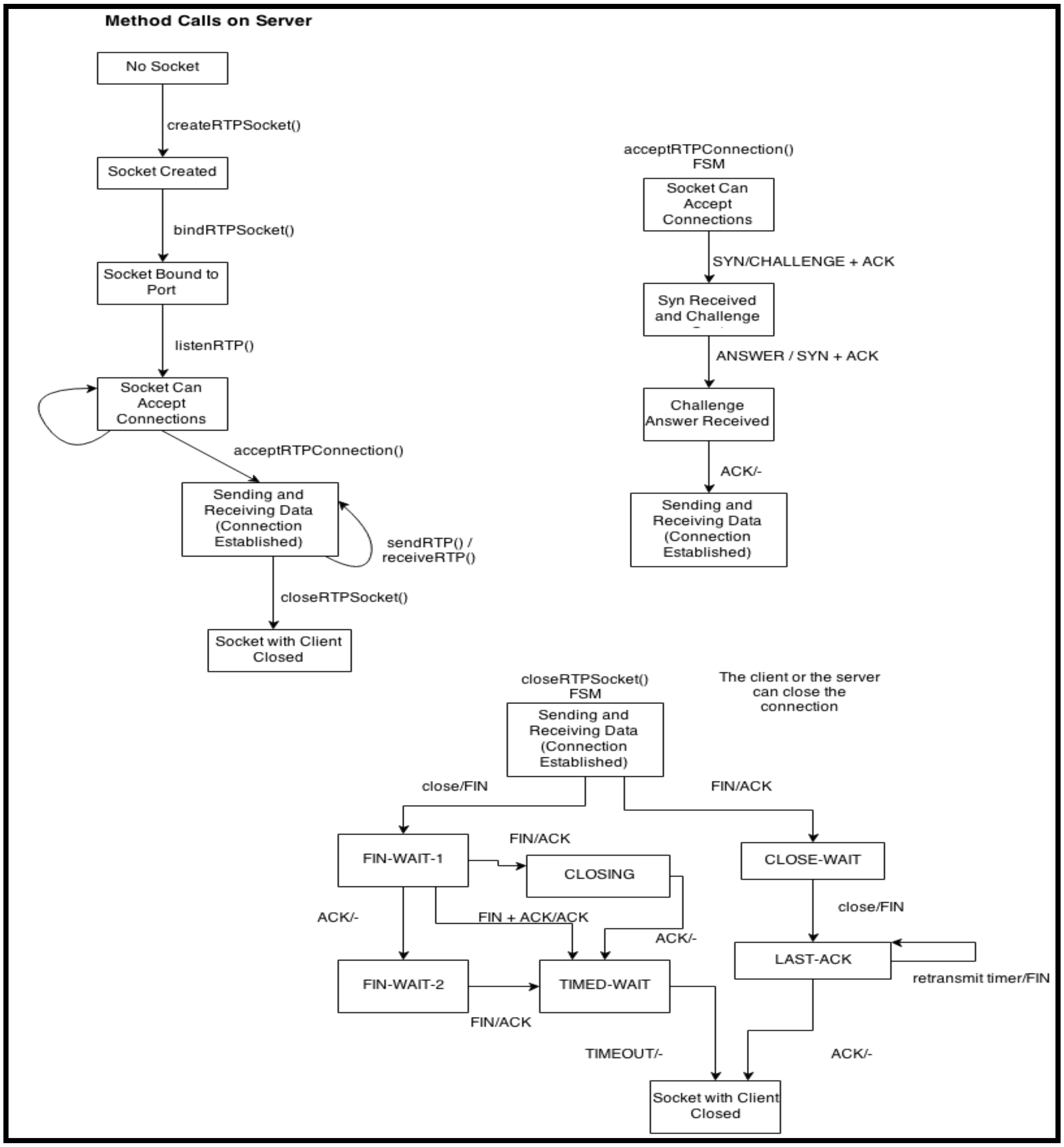
Header Structure

Source Port - This is the 32-bit port number for the sender of the data.
Destination Port - This is the 32-bit port number for the destination of the data.
Sequence Number - This is the 32-bit sequence number of the data being sent from the source to the destination. This number will count the number of bytes being sent.
Acknowledgement Number - This is the 32-bit sequence number of the data that was received from the destination. This will count the number of bytes being sent.
Checksum - This is for verification that package was not corrupted. In our protocol, this field will contain the first 64 bits of a SHA-2 256-bit hash of the packet's data and header (with the exception of the checksum field).
Length - This 32-bit field is the length of the message in bytes.
Window - This 32-bit field will specify the number of bytes the recipient is currently willing to receive from the sender. (This field is still present in our protocol's header, but is not being used).
SYN Flag - If this bit is set, it will indicate that the packet is a SYN packet for connection establishment.
ACK Flag - If this bit is set, it will indicate that the packet's acknowledgement number is valid.
FIN Flag - If this bit is set, then one of the parties wants to terminate the connection.
CNG Flag - If this bit is set, then the client has to respond to the server with the answer to the challenge
EOF Flag - If this bit is set, then it indicates to the current recipient recognizes that it is the final packet in the message. This will cause the receive method to return the data to the application.
Reserved - Different implementations may need additional bits, so we leave these in the fixed header length to accommodate other future implementations.


0
15
31
Source Port 16 bits
Destination Port 16 bits
Sequence Number 32 bits
Acknowledgment Number 32 bits
Checksum first 32 bits
Checksum 2nd 32 bits
Window 32 bits
Length 32 bits
SYN
ACK
FIN
CNG
EOF
Reserved 27 bits
Payload (variable length)

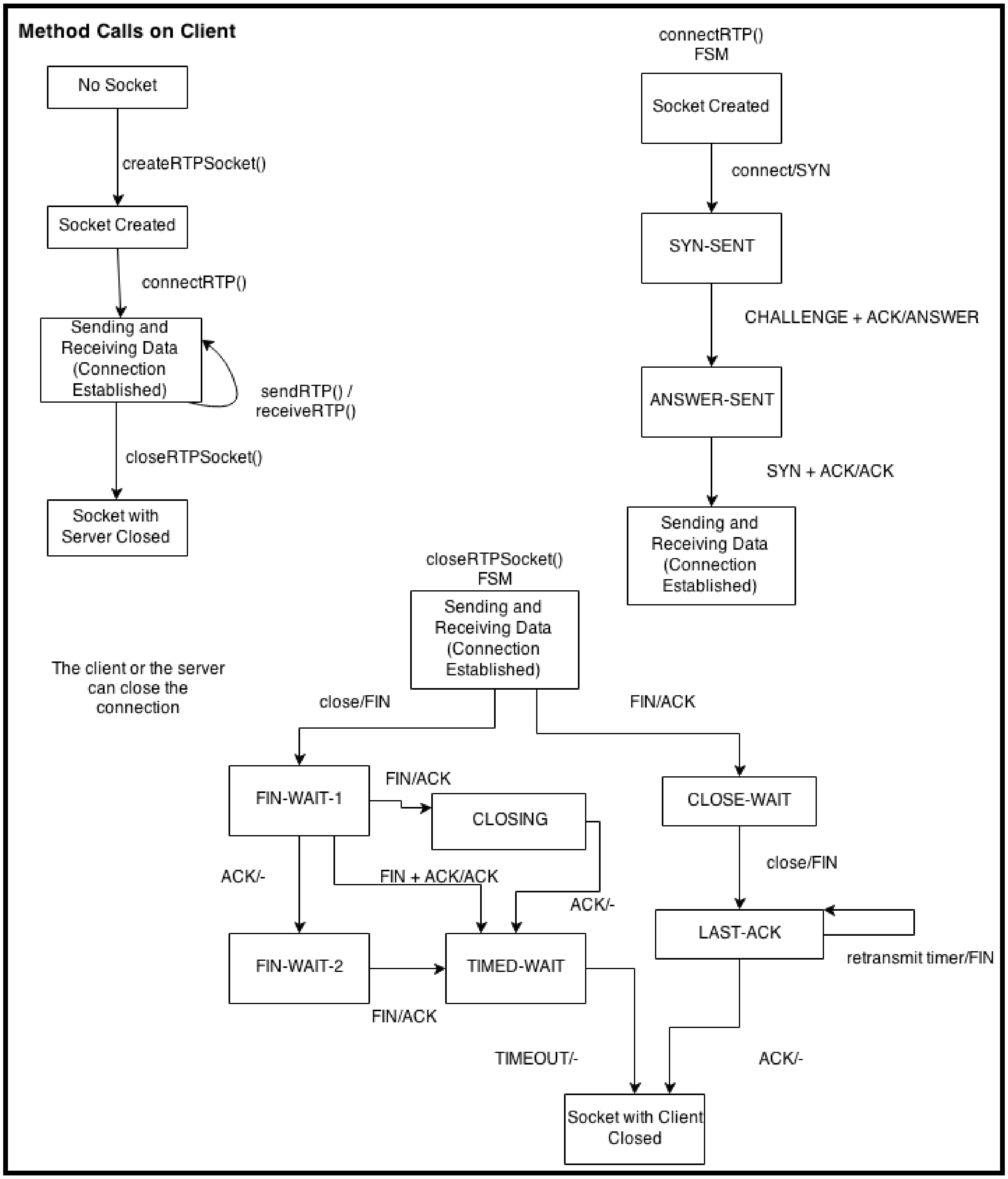
Finite State-Machine Diagrams

Finite State-Machines for Server



Method Calls on Server
No Socket
createRTPSocket()
Socket Created
bindRTPSocket()
Socket Bound to Port
listenRTP()
Socket Can Accept Connections
acceptRTPConnection()
Sending and Receiving Data (Connection Established)
sendRTP() / receiveRTP()
closeRTPSocket()
Socket with Client Closed
acceptRTPConnection() FSM
Socket Can Accept Connections
SYN/CHALLENGE + ACK
Syn Received and Challenge
ANSWER / SYN + ACK
Challenge Answer Received
ACK/-
Sending and Receiving Data (Connection Established)
closeRTPSocket() FSM
The client or the server can close the connection
Sending and Receiving Data (Connection Established)
close/FIN
FIN/ACK
FIN-WAIT-1
FIN/ACK
CLOSING
CLOSE-WAIT
ACK/-
FIN + ACK/ACK
ACK/-
close/FIN
LAST-ACK
retransmit timer/FIN
FIN-WAIT-2
TIMED-WAIT
FIN/ACK
TIMEOUT/-
ACK/-
Socket with Client Closed

Finite State-Machines for Client



Protocol’s Programming Interface

RTPSocket createRTPSocket( )
Creates an RTP socket. The function call will return an object of RTPSocket, which can be used by the program to send and receive messages.

void bindRTPSocket(IPaddress, portNumber)
Associates a port on the host machine with the socket invoking this function. This function will be invoked by the host that is acting as the server.

RTPSocket acceptRTPConnection( )
This function, invoked by the server, blocks until a client attempts to connect to the server. Once client attempts to form a connection, this function carries out the necessary handshake procedure necessary to form a connection. The function then returns true if a socket object that can be used to communicate with the now connected client has been created.

String receiveRTP(int numBytes)
This function blocks until data is received via the socket that function is invoked on. A buffer will be created to collect at most the number of bytes specified in the parameter. Once all the data is received, the function returns it to the callee. This function will be responsible for returning the data in the correct order to the application. In addition, this function will guarantee the data received from the socket is not corrupt through the use of the checksum field within the RTP header.

void listenRTP( )
This function will cause the invoking socket to allow connections from clients to be formed.

void connectRTP(IPaddress, portNumber)
This function causes the invoking socket to attempt to form an RTP connection with the server at the specified IP address and port number. It will carry out the the necessary handshake steps.

void sendRTP(bytearray message)
This function will send the given message through the socket to the destination host with which the local host has formed a connection. In order for this function to work properly, the client or server will have to first invoke connect it is the client or accept if it is a server. The sendRTP function guarantees that the message will be sent to the destination and will arrive in order. It will also be responsible for forming the RTP header in which the message will be encapsulated.

void closeRTPSocket( )
This function will end the connection formed between the socket invoking the function and the remote host. It will cause the necessary connection teardown process to occur in order to end the RTP connection between the two hosts.

Algorithmic Descriptions

1. **Checking for Corrupt Packets**

   For the checksum, we will be implementing a SHA2 hash. We are using the 256-bit version. SHA2 produces a secure hash, meaning that we can use any 64 of the 256 bits it produces for our checksum. We will use the first 64 bits. The message and the header will be hashed, with the checksum being treated as sixty-four 0s for the purpose of hashing.

   If the calculated checksum matches the checksum in the header, the receiver will send an acknowledgement for the packet to the sender. If not, no acknowledgement will be sent. This will cause the sender's acknowledgement timer to expire, resulting in the corrupt packet being sent again.

2. **Checking for Lost Packets**

   We will use a timeout period of 2 seconds. If an acknowledgement is not received in that period, then we will resend the packet. The receiver will ignore any packets with sequence numbers ahead of what it was expecting. This will enable the timeout to work.

3. **Checking for Out-of-Order Packets**

   If we receive a packet with a sequence number ahead of what the client is expecting, we will ignore it. The sender will not receive an acknowledgement, and will resend, The receiver will then receive the re-sent packet and add it to the received data.

4. **Duplicate Packets**

   Duplicate packets will be ignored by the receiver and will not be saved. An acknowledgement will not be sent back to the sender.

5. **Bi-Directional Transfer**

   Since the packet structure will be identical for the two directions of data transfer, data will be able to be included in the packet's payload field. Therefore, like TCP, data can be included in a packet going from Host B to Host A alongside an acknowledgement for data sent by Host A to Host B.

6. **Connection Establishment**

   In our protocol, we implement a strategy in order to prevent a denial-of-service attack known as the SYN Flood Attack. In order to do so, the host that initiates the connection has to send two messages to the server, one is the initial SYN request and the second is an answer to a challenge from the server. In our implementation, the server will send the client host a randomly generated number within its challenge packet (indicated by the CNG (challenge) bit being set in the header). The client will have to successfully send this same randomly generated number back to the server as the response to the challenge.

Once the server receives the answer to its challenge from the client, any necessary resources for the connection will be allocated.