UDP++

CSCI 43600 Principles of Computer Networking

April 26, 2010

Adam Darrah

[akdarrah@umail.iu.edu](mailto:akdarrah@umail.iu.edu)

0002332558

Asad Saeed

[asaeed@iupui.edu](mailto:asaeed@iupui.edu)

0002453548

Our UDP++ transport layer protocol was designed with several key ideas kept in mind. First, we wanted to keep the protocol fast, and time sensitive as it was originally designed, and to do this we limited the features we added to only what we felt was necessary. Second, we wanted it to be as flexible as possible and give the application programmer control over the connection. Last, we wanted to be able to support multiple simultaneous connections.

There are three classes involved in our UDP++ transport layer protocol. There is a packet class which models the data sent and received. The “UDPPlus” class encapsulates the core C++ UDP transport layer protocol, and manages active and incoming connections. And the “UDPPlusConnection” class models an active connection and handles all sent or received packets and connection state.

The packet class is a wrapper for data sent over the core UDP++ transport layer. We include several other fields that allow us to manage packets and make sure that they are in order. Here is a diagram of the header used:

0 1 2 3

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Source Port | Destination Port |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Length | Checksum |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

|S|A|S|F|O| | Header | | |E|C|Y|I|P| | Length | Sequence # | |Q|Q|N|N|T| | | |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| | Optional Field For SACK |

| Acknowledgment # | Controlled by OPT BIT and |

| | Header Length |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| data |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

The first two fields in the diagram are included with UDP, so we get source port, destination port, length, and checksum for free. Usually with UDP, after those two fields you would just insert the data you wanted to send and send it on its way. But in order to implement UDP++, we added more necessary fields. First, we have one bit flags to indicate the packet type, such as SEQ, ACK, SYN, FYN, or OPT packets. Based on these flags UDP++ can determine what to do with the packet and what other fields will be used. Because we have an optional field in the header, we need to know the header length and that is what the next eight bits after the flags are for. Sequence number and acknowledgement number are used as expected. If the OPT flag has been set, the 16 bits after acknowledgement number are reserved for optional data. Then of course the data is inserted into the message.

The “UDPPlus” class encapsulates UDP connection methods and manages a list of all “UDPPlusConnection” objects. The “UDPPlus” class has a single thread, which receives all data (packets) sent to the binded port. The listener thread then determines if the packet received is from a new or existing connection. If it is a new connection, it creates a new “UDPPlusConnection” object and adds it to the list of active connections. If it is a current connection, it passes the packet to the connection’s “handlePacket” method and that connection determines what to do with the packet.

The “UDPPlusConnection” class represents one active connection. A single program could have many active connections, and they could be managed individually or through a “UDPPlus” object. We allow the application programmer to specify the buffer size and maximum number of connections when they create their “UDPPlus” object. The “UDPPLusConnection” object manages the connections state and handles the sending and receiving of packets. This class also has a buffer of incoming and outgoing packets. There is a timer thread that monitors the buffer of sent packets and checks for timeouts.

This diagram shows the relationships between the classes in UDP++. You can see that this protocol will have a maximum of n connections + 1 threads. All connections share the same listener thread where packets are received. Then each “UDPPlusConnection” has a timer thread that can detect timeouts.

+-+-+-+-+-+-+-+-+ +-+-+-+-+-+-+-+-+-+

| UDPPlus | -> | listener thread |

+-+-+-+-+-+-+-+-+ +-+-+-+-+-+-+-+-+-+

|

| has many UDPPlusConnections

|

---------------------------------------------------------------------------------------

| |

+-+-+-+-+-+-+-+-+-+-+ +-+-+-+-+-+-+-+- +-+-+-+-+-+-+-+-+-+-+ +-+-+-+-+-+-+-+-

| UDPPlusConnection | -> | timer thread | | UDPPlusConnection | -> | timer thread |

+-+-+-+-+-+-+-+-+-+-+ +-+-+-+-+-+-+-+- +-+-+-+-+-+-+-+-+-+-+ +-+-+-+-+-+-+-+-

As stated above, we designed the protocol to be flexible to the application. In fact, the methods called to use UDP++ are very similar to using TCP. Following are examples of how to use UDP++ at the application level (keep in mind that you will need to include appropriate header files and boost libraries in order for these programs to run correctly).

Here is an example client program:

int main(int argc, char\* argv[]) {

UDPPlus \*conn = new UDPPlus;

UDPPlusConnection \*open;

const int PORT = 30000;

// setup sockaddr\_in struct

struct sockaddr\_in host;

memset((char \*) &host, 0, sizeof(host));

host.sin\_family = AF\_INET;

host.sin\_port = htons(PORT);

// connect to server

open = conn->conn(&host, sizeof(host));

// do something with open (UDPPlusConnection object

// can send or receive data packets)

return 0;

}

This example shows how you could use UDP++ for a client application. You would create a “UDPPLus” and “UDPPLusConnection” object and define the port number you would like to use for this connection. You then setup the “sockaddr” struct just as you would for a normal UDP connection. Then call the “conn” method with the “sockaddr” struct and length of “sockaddr” struct as arguments and this will return the “UDPPlusConnection” object you have just created. You can now send and receive data using this object. You can either close the connection yourself, or delete the “UDPPlus” object, which will close all connections.

Here is an example of a server program:

int main(int argc, char\* argv[]) {

UDPPlus \*conn = new UDPPlus;

UDPPlusConnection \*open;

const int PORT = 30000;

// setup sockadr\_in struct

struct sockaddr\_in local;

memset((char \*) &local, 0, sizeof(local));

local.sin\_family = AF\_INET;

local.sin\_port = htons(PORT);

local.sin\_addr.s\_addr = AI\_PASSIVE;

conn->bind\_p(&local, sizeof(local));

open = conn->accept\_p();

// do something with open (UDPPlusConnection object

// can send or receive data packets)

return 0;

}

It is very similar to the example client program. You make the same objects, but you need to setup the “sockaddr” struct differently to define the IP address (“AI\_PASSIVE” means localhost) and "AF\_INET” defines IP version. Call “bind\_p” method on the “UDPPlus” object to bind the port and start the listener thread to wait for incoming data. Then call “accept\_p” to accept a new connection, which returns the “UDPPlusConnection” object. Then you can either send or receive data using this object.

We are happy with the state our UDP++ protocol is in currently. However, here are several things that could be improved for the future. First, there is no flow control, currently if the buffer is full we drop the packet and send an “ACK”, with “SACK” bit mask, asking for the packet to be resent. Second, our protocol does not support congestion control, which is something that TCP utilizes. Third, we have not implemented pipelined data transfer with a send and receive window. And last, we do not use a checksum to validate the header fields.