HOME EXAM 2

INF3190 - SPRING 2018

15130

Answer these question:

– What problem occurs in case two files are sent simultaneously from two different sources to the same destination using port 30? Why? How can this problem be solved?

If the destination doesn't have a way to uniquely identify the incoming files, then the destination would think that those packages it receive are from the same file as it assumes that if the datagram.port == recv_file.port then it has to be the same file. And then it would try to mash the packages it receives from to different sources together, and create a mesh of the two files. A way to solve this is to have a 2-tuple of (mip_src, port), this way you can separate the two files from eachother. Which is also what I implemented in miptp. (at least tried to).

– How is the problem solved on the Internet, e.g. with respect to a web server which listens on the same port (80) irrespective of where connections are coming from?

For each connection the web server receives, the web server creates a separate connection socket (kinda like the listening socket I created in miptp), and each socket is identified by a unique four-tuple <src_ip, src_port, dst_ip, dst_src>. And when the server receives a datagram, it examines these four values to determinate to which connection socket it should pass the payload of the incoming datagram.

– How can the time out value be computed automatically, so that it need not be specified as a command line argument?

By periodically measuring the time it took too receive an ACK response from x when we sent y bytes, and then storing that value in a cache, and based on this, the next time we send to x we can then take a "estimate" on how long it is going to take to receive the next ACK, and if we store the last 5 times in this cache, then we could get an average time expected. And since we are updating the cache every time we get an ACK, we would also adapt to changes in the network that might cause increased Round trip time.

Design:

The overall task of home-exam2 was to implement the transport layer, capable of transporting files up to 65535 bytes.

Files.c:

> Files.c is a rather simple program, it takes a sockpath and a port argument, and then tries to connect to a miptp running on the given sockpath, and provides the miptp with its port. And then waits for a file to write to the file system. It receives a file with the length prefixed before the file. It then writes the file, increments the file counter, and waits for the next file from the miptp. The filename starts with the port and ends with the file written counter in this fashion: “40_recieved_file_0” … : “40_recieved_file_1”.

Filec.c:

> Filec.c is also a rather simple program, it takes a sockpath, a filename, a mip_adress and a port. Then it reads a file which it sends to a miptp it connected to via the given sockpath. The file is prefixed by 5 bytes. Which consists of : the mip_addr, the port and the length of the file. After it has successfully sent the file to the miptp it shuts down.
>
> Filec.c discards file of greater than 65535 size, due to the constrictions given in home-exam2

Miptp.c:

> Miptp.c however is a more complex beast than the other two programs.
> The general rundown of the program is that it accepts multiple servers/clients, and then tries to send the files from filec, to a files running on another miptp by fragmenting the file and then sending miptp-packets consisting of maximum 1496 bytes to the mip-daemon which then either does a arp- or route look up, and then sends it to the correct mip-daemon which then sends it to its local miptp. And then the receiving miptp defragments the packet into a file, which it then sends to a files listening on the given port, or discards the file if no files is listening on the given port.
>
> The way the files gives its port to the miptp is by sending the port prefixed by the mip_addr of 255. Since 255 is an invalid mip_addr (since it is reserved for the broadcasting of routes from HE1), then we can use it as a special case for easy delivering of ports from files.
>
> I chose to store the files In a linked list, as I have thought that would be better than having a large array of send_file / recv_file pointers, and I found linked list easier to wrap my head around when having a very variable number of files I had to store.

I also had a hard time putting a time_t inside the file_tp2 struct, so I had to make the linked list a new struct instead of having pointers to the next file_tp2 inside file_tp2

The miptp uses the go-back-n (with windows size of 10) strategy to make sure that all the packages gets sent in the correct order, and when it receives an ack, it increment the window 1 up. (i.e 0-9, received ack for 0, new is 1-10…), and since it sends the whole window each time it sends a package, I let one timeout pass before I send the next window of packages, due to avoiding of congesting the network to much. Since it would be wasteful of to send 1-10 right after we have comfirmed 0. When we can wait just a little bit, to see if we get the next ack, and so on. It might slow down the sending process, but it reduces the congestion of the network so I deemed it worth it to implement go-back-n in this way.

How to Run the program:

You might have to attempt starting the more than one time due to unbinding some of the sockpaths. It shouldn't happen, but it might.

The easiest way to compile is to use the supplied makefile.

make daemon

make miptp

make filec

make files

make ruter

and to run the program you can use the following commands:

./daemon [-h] [-d] <socket_application> <routing_socket> <forward_socket> [mip_adresses]

./miptp [-d] <MIPD_PATH> <MIPTP_PATH> <TIME_OUT>

./filec <FILE NAME> <MIPTP_PATH> <MIP ADDR> <PORT>

./files < MIPTP_PATH > <PORT>

./ruter [-h] <routing_socket> <forward_socket>

NB: mip_addresses must be in the range of 1-254.

NB: the help command does not execute the program, but only prints the help text. NBB: only mipdaemon and ruter has help print.

Program files:

home-exam2:

- filec.c
- miptp.c
- files.c
- makefile

other files:

- ruter.c
- daemon.c
- colours.c

the other files, are not a part of the home-exam2, but are the environment that I build the miptp application around.

Assumptions, peculiarities and other closing comments

I assumed that <mip_addr, port> formed a unuiqe indentifier for the file we are sending. In other words, one miptp daemon only supports sending one file at the time from the same client to the same host. (I.E if you send to <10 , 20>, then you can't queue up another packet to <10 , 20> on the same host before the original packet has finished sending. And it is the same the other way, you can't receive multiple packets from the same <mip_addr, port> until you have finished receiving the first packet.

NB: the program leaks memory like it is no one concern, I had some trouble freeing things sometimes causing segfaults, and valgrind did not work for me in the mininet environment.
I'm only bringing this up, because I ran out of memory one time, but that was after running the vm for around 8 hours.