8005 Final project

Design & Documentation

Eric Wu, A00961904
Hong Kit, Wu A00968591

BTech 2020 Winter

Table of contents

[Objective](#)

[Approach](#)

[Pseudo implementation](#)

[State diagram](#)

[Usage](#)

[How to run](#)

[Testing](#)

Objective

- To design and develop a network application that uses advanced TCP/IP programming techniques.
- To design and implement a minimum-functionality "Port Forwarder" using any language of your choice.
- ~~A maximum-functionality port forwarder could include a proxy server, caching, etc.~~

Approach

First, we will create a c program that creates an epoll instance and assigned it with an IP/port pair. Upon receiving an external connection request, the epoll instance will handle that request, accepted it and add it to the monitoring list. The accept operation will be offload to multiple threads. We will implement this feature using OpenMP.

If the epoll instance receives data that is destined to an internal client on a connection established socket, it will remove that socket from the monitoring list, and fork a child process to handle further communication between the external and internal client.

We will create a bash script to run the c program as deamon once for each of the IP.port pairs defined in the config file.

Pseudo implementation

Initialize socket {
    Create a stream socket for listening (**fd_server**);
    Set the listening socket to address reusable;
    Bind an address to the listening socket;
    Listen on listening socket;
}

Monitoring setup {
    Create an event instance (**event**) with EPOLLIN, EPOLLERR, EPOLLHUP, EPOLLET and EPOLLEXCLUSIVE;
    Create an event list (**events**) instance use for storing events return from **epoll_wait;**
    Create an epoll fd instance and subscribe to the events in **event**;
    Set number of threads to use for OpenMP
}

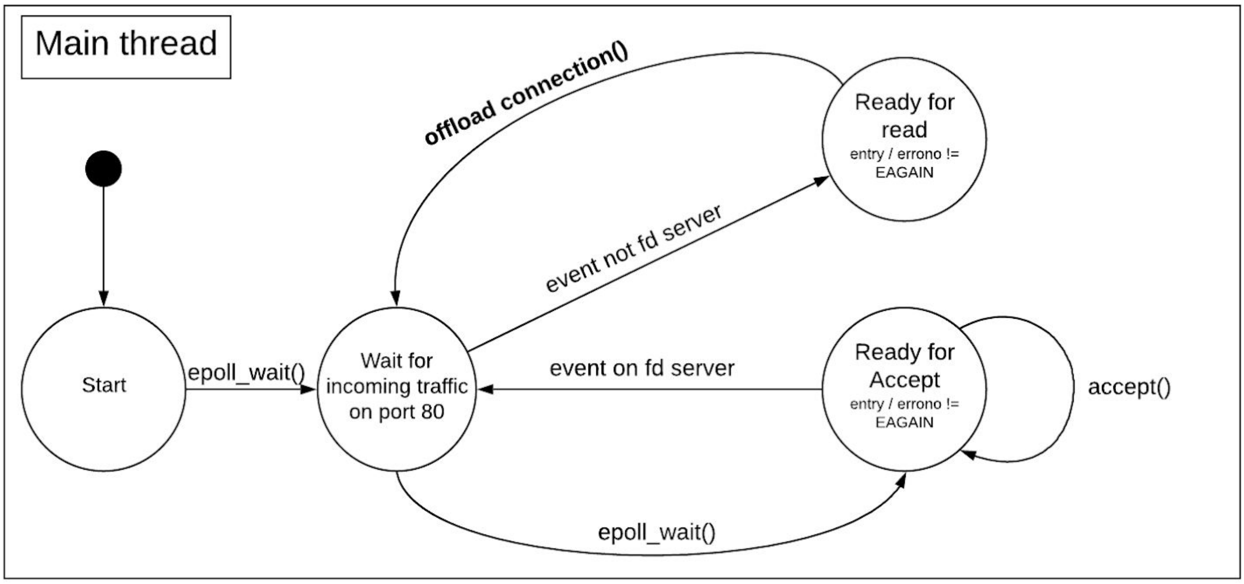
OffloadConnection(port, ip, **fd**) {

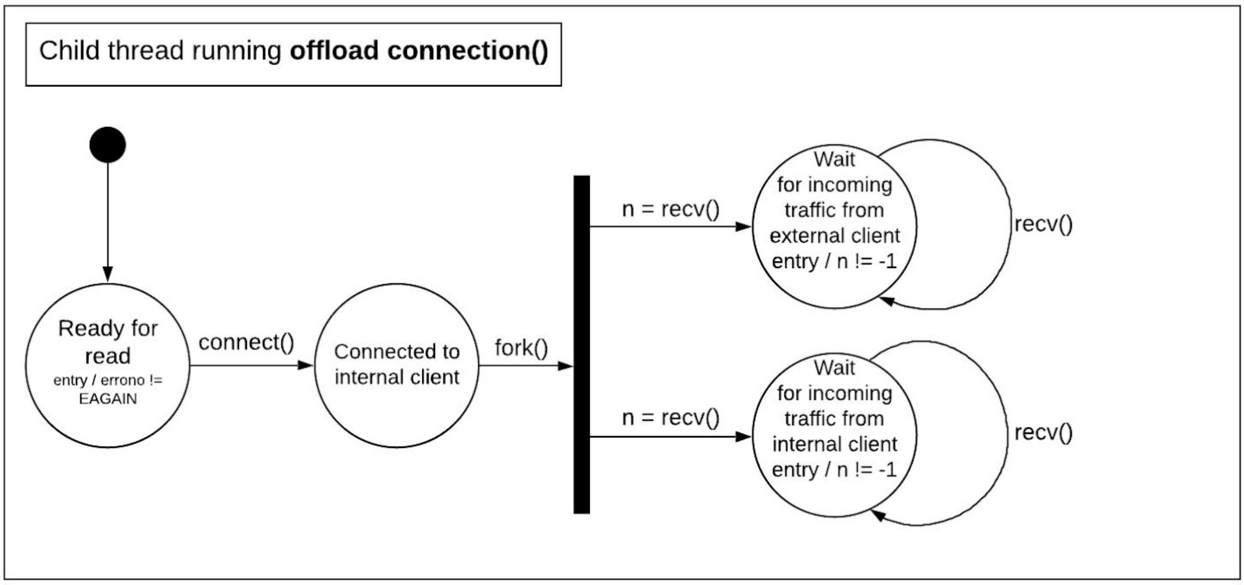
```
    create a new socket server_fd
    establish a new connection to the internal host (ip) on port (port) using server_fd
    childpid = fork()
    If childpid == 0
        handle incoming traffic from fd, and forward it to server_fd
    else
        handle incoming traffic from server_fd, and forward it to fd
    exit(0)
}

Event loop {
    while (true) {
        epoll_wait() call, block and wait for activity, store events to events on unblock;
        Loop through events in events up to the number of descriptor return from
        epoll_wait {
            if the new event is EPOLLHUP {
                Close the client socket;
                continue;
            }
            if new event is EPOLLERROR {
                Print error to stderr and close the socket
                continue;
            }
            if new event's fd == fd_server {
                Create a flag for indicating EAGAIN error(EAGAIN_REACHED), and
                initialize it to false;
                Run in parallel {
                    while (!EAGAIN_REACHED) {
                        fd_new = accept() the new connection;
                        if fd_new == -1, check errno {
                            if errno is either EAGAIN or EWOULDBLOCK {
                                flip the EAGAIN_REACHED flag to true;
                                break;
                            } else {
                                Error occurred, print the error message to stderr
                                break;
                            }
                        }
                        Set the socket fd_new to non blocking;
                        Add the new socket descriptor to te epoll loop;
                    }
                }
                continue;
            }
            Reset the event trigger fd to blocking mode, and remove it from epoll
            childpid = fork()
            if childpid == 0 {
                OffloadConnection(internal server port, internal server ip, new event's fd)
```

```
            }
        }
    }
}

main() {
    Initialize socket;
    Monitoring setup;
    Event loop;
}
```

State diagram

server.c





Usage

Name

server port hostname host-port

Arguments

port	server listening port
hostname	internal client IP
host-port	internal client port

Description

This program bridges traffic between external clients and internal clients. For example, assuming the arguments are set to **port** = 80, **hostname** = 192.168.0.2 and **host-port** = 8080, any incoming traffic on port 80 from external client will be redirect to 192.168.0.2 on port 8080.

How to run

To quickly test the program, cd into the "executable" directory, and run through **Step 1** and **3** only. If this does not work, cd into the "source" directory, and run through all **Step 1 ~ 3**.

Step 1: add IP/port pair in the setup.config file
Example
7005 -> 192.168.0.1:8005
7006 -> 192.168.0.2:8006
In this example, the port forwarder will listen on both port 7005 and 7006. Any traffic destined to port forwarder on port 7005 will be redirected to the internal client 192.168.0.1 on port 8005. Any traffic destined to port forwarder on port 7006 will be redirected to the internal client 192.168.0.2 on port 8006.

Step 2: compile the c program using the Makefile in the same directory
$ make

Step 3: run the bash script "**port-forward.sh**". This will run the c program up to the number of entries defined in the config file. The program will run in the background.
$ ./port-forward

Finally, to stop the background process, run. This will stop the already running port-forwarder.
$./stop-script

Testing

Case 1: In this case, we try to test the maximum active socket the server can handle in one session. We will keep adding external clients indefinitely until there's a significant degradation in performance on the server-side. Each spawn client will send an echo request to the server and wait for a reply, this process will continue for enough time until the performance on the server-side drops.

(* We don't have access to workstations at the lab, nor do we have a desktop at home. Because this test can not be conducted on a laptop, we will use the test result from the previous epoll assignment. The epoll module we use for this final project is exactly the same as the one we used for the assignment.)

Testing params

- 20000 echo request (10000 each on two client machines)
- 10000 nano sec delay in between each echo request (not including the response time from the previous request)
- 0.002-sec client spawn delay

Monitoring tool: Wireshark

Results:

Wireshark conversations on server

Figure 1 Wireshark Conversation

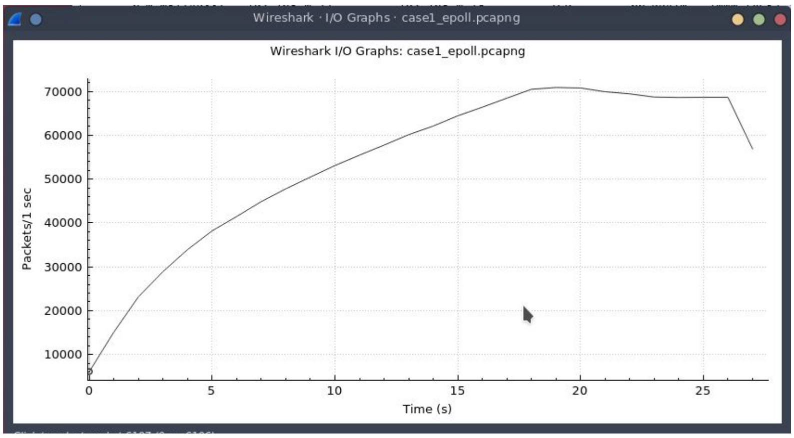


Figure 2 Wireshark I/O graph on server

In figure 1, we recorded that there were roughly 20000 active socket connections in one session by using epoll. And we found that if we kept the test longer as the number of active connections goes higher, the epoll server showed that the incoming packets dropped when the active socket count hits 20000 counts according to the figure 2. We believed that the reason behind this was because the server was not able to reply back to the client as quickly as before, i.e. the server response time increased. The client will only send one request when the previous request is answered.

Case 2: In this case, we try to verify if the server can handle multiple external traffics destined to multiple internal clients. In essence, if there are two internal web servers, A and B, clients from outside should be able to request a web page content from either webserver A or B through the port forward server.

Machine IPs:

- 192.168.0.44: Client
- 192.168.0.41: Forwarder
- 192.168.0.52: Internal HTTP Server
- 192.168.0.53: Internal HTTP Server

Testing params:

- 192.168.0.44 connects to 192.168.0.52 (Port 8008) through 192.168.0.41(Port 8005)
- 192.168.0.44 connects to 192.168.0.53 (Port 8009) through 192.168.0.41(Port 8006)

Monitoring tool: Wireshark

Results

Address A	Port A	Address B	Port B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Rel Start	Duration	Bits/s A → B	Bits/s B → A
192.168.0.41	35812	192.168.0.52	8009	7	1,068	4	682	3	386	7.891966	0.0081	671 k	379 k
192.168.0.44	44092	192.168.0.41	8006	7	1,068	4	682	3	386	7.884150	0.0178	305 k	173 k

Figure 1 TCP conversation (192.168.0.52 HTTP Server)

tcp && ((ip.src==192.168.0.44 && ip.dst==192.168.0.41) || (ip.src==192.168.0.41 && ip.dst==192.168.0.52) || (ip.src==192.168.0.52 && ip.dst==192.168.0.41) || (ip.src==192.168.0.41 &

No.	Time	Source	Destination	Protocol	Length	Info
108	2020-04-03 21:41:12.782918746	192.168.0.44	192.168.0.41	TCP	74	44092 → 8006 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 …
109	2020-04-03 21:41:12.782958448	192.168.0.41	192.168.0.44	TCP	74	8006 → 44092 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 S…
110	2020-04-03 21:41:12.786171682	192.168.0.44	192.168.0.41	TCP	66	44092 → 8006 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=71882686 …
111	2020-04-03 21:41:12.786199229	192.168.0.44	192.168.0.41	HTTP	476	GET / HTTP/1.1
112	2020-04-03 21:41:12.786210027	192.168.0.41	192.168.0.44	TCP	66	8006 → 44092 [ACK] Seq=1 Ack=411 Win=64768 Len=0 TSval=2555744…
113	2020-04-03 21:41:12.790733879	192.168.0.41	192.168.0.52	TCP	74	35812 → 8009 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 …
114	2020-04-03 21:41:12.792655615	192.168.0.52	192.168.0.41	TCP	74	8009 → 35812 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 S…
115	2020-04-03 21:41:12.792677226	192.168.0.41	192.168.0.52	TCP	66	35812 → 8009 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=386577912…
116	2020-04-03 21:41:12.792836327	192.168.0.41	192.168.0.52	TCP	476	35812 → 8009 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=410 TSval=38…
117	2020-04-03 21:41:12.795891120	192.168.0.52	192.168.0.41	TCP	66	8009 → 35812 [ACK] Seq=1 Ack=411 Win=64768 Len=0 TSval=8937409…
118	2020-04-03 21:41:12.798846022	192.168.0.52	192.168.0.41	TCP	246	8009 → 35812 [PSH, ACK] Seq=1 Ack=411 Win=64768 Len=180 TSval=…
119	2020-04-03 21:41:12.798863002	192.168.0.41	192.168.0.52	TCP	66	35812 → 8009 [ACK] Seq=411 Ack=181 Win=64128 Len=0 TSval=38657…
120	2020-04-03 21:41:12.798882053	192.168.0.41	192.168.0.44	HTTP	246	HTTP/1.1 304 Not Modified
121	2020-04-03 21:41:12.800768256	192.168.0.44	192.168.0.41	TCP	66	44092 → 8006 [ACK] Seq=411 Ack=181 Win=64128 Len=0 TSval=71882…

Figure 2 Wireshark pcap (192.168.0.52 HTTP Server)

According to Figure 1, we can see that the client is trying to send requests from high port to port 8009 of the HTTP Server(192.168.0.52) via port 8006 of the Port Forwarder. Then when we take a look at the pcap for detailed info (Figure 2), we find that the client sends an HTTP GET request after doing a three-way handshake to the

Port Forwarder(no.111). After that, Forwarder will then forward the request to the HTTP server when it finishes the three-way handshake to port 8009 of the HTTP Server(no.116). HTTP server then sends the result back to the Port Forwarder after receiving the GET request(no.118). Finally, the Port Forwarder sends back the result to the client (no.120).

Address A	Port A	Address B	Port B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Rel Start	Duration	Bits/s A → B	Bits/s B → A
192.168.0.41	53224	192.168.0.53	8008	11	1,335	6	815	5	520	2.060299	5.6237	1,159	739
192.168.0.44	52346	192.168.0.41	8005	9	1,202	5	749	4	453	2.055577	5.6647	1,057	639

Figure 3 TCP conversation (192.168.0.53 HTTP Server)

=192.168.0.41 && ip.dst==192.168.0.53) || (ip.src==192.168.0.53 && ip.dst==192.168.0.41) || (ip.src==192.168.0.41 && ip.dst==192.168.0.44))&& (tcp.port == 8008|| tcp.port == 8005)

No.	Time	Source	Destination	Protocol	Length	Info
73	2020-04-03 21:41:06.954345555	192.168.0.44	192.168.0.41	TCP	74	52346 → 8005 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 …
74	2020-04-03 21:41:06.954387940	192.168.0.41	192.168.0.44	TCP	74	8005 → 52346 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 S…
75	2020-04-03 21:41:06.956532699	192.168.0.44	192.168.0.41	TCP	66	52346 → 8005 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=71876857 …
76	2020-04-03 21:41:06.957400315	192.168.0.44	192.168.0.41	HTTP	477	GET / HTTP/1.1
77	2020-04-03 21:41:06.957416048	192.168.0.41	192.168.0.44	TCP	66	8005 → 52346 [ACK] Seq=1 Ack=412 Win=64768 Len=0 TSval=2555738…
78	2020-04-03 21:41:06.959066873	192.168.0.41	192.168.0.53	TCP	74	53224 → 8008 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 …
79	2020-04-03 21:41:06.962347809	192.168.0.53	192.168.0.41	TCP	74	8008 → 53224 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 S…
80	2020-04-03 21:41:06.962369314	192.168.0.41	192.168.0.53	TCP	66	53224 → 8008 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=854651562…
81	2020-04-03 21:41:06.962555528	192.168.0.41	192.168.0.53	HTTP	477	GET / HTTP/1.1
82	2020-04-03 21:41:06.965933386	192.168.0.53	192.168.0.41	TCP	66	8008 → 53224 [ACK] Seq=1 Ack=412 Win=64768 Len=0 TSval=2807600…
83	2020-04-03 21:41:06.966961496	192.168.0.53	192.168.0.41	HTTP	248	HTTP/1.1 304 Not Modified
84	2020-04-03 21:41:06.966978900	192.168.0.41	192.168.0.53	TCP	66	53224 → 8008 [ACK] Seq=412 Ack=182 Win=64128 Len=0 TSval=85465…
85	2020-04-03 21:41:06.966998939	192.168.0.41	192.168.0.44	HTTP	247	HTTP/1.1 304 Not Modified
86	2020-04-03 21:41:06.970230264	192.168.0.44	192.168.0.41	TCP	66	52346 → 8005 [ACK] Seq=412 Ack=182 Win=64128 Len=0 TSval=71876…
93	2020-04-03 21:41:12.066011213	192.168.0.53	192.168.0.41	TCP	66	8008 → 53224 [FIN, ACK] Seq=182 Ack=412 Win=64768 Len=0 TSval=…
94	2020-04-03 21:41:12.106076780	192.168.0.41	192.168.0.53	TCP	66	53224 → 8008 [ACK] Seq=412 Ack=183 Win=64128 Len=0 TSval=85465…
101	2020-04-03 21:41:12.578651605	192.168.0.44	192.168.0.41	TCP	66	52346 → 8005 [FIN, ACK] Seq=412 Ack=182 Win=64128 Len=0 TSval=…
102	2020-04-03 21:41:12.579036334	192.168.0.41	192.168.0.53	TCP	66	53224 → 8008 [FIN, ACK] Seq=412 Ack=183 Win=64128 Len=0 TSval=…
105	2020-04-03 21:41:12.582780234	192.168.0.53	192.168.0.41	TCP	66	8008 → 53224 [ACK] Seq=183 Ack=413 Win=64768 Len=0 TSval=28076…
107	2020-04-03 21:41:12.619076880	192.168.0.41	192.168.0.44	TCP	66	8005 → 52346 [ACK] Seq=182 Ack=413 Win=64768 Len=0 TSval=25557…

Figure 4 TCP Wireshark app (192.168.0.52 HTTP Server)

After that, the client starts to access the second HTTP server (192.168.0.53) via the Port Forwarder. We can see that in Figure 3, the client sends packets from high port to port 8005 of the Port Forwarder. And the Port Forwarder will forward the packets to the port 8008 of the HTTP server(192.168.0.53). In Figure 4, we can see that after the client finishes a three-way handshake to the Port Forwarder, the client starts to send an HTTP GET request to port 8005 of the Port Forwarder (no.76). Then, the Port Forwarder will forward the GET request to the HTTP server(no.81). The HTTP server will send the result back to the Port Forwarder(no.83), which the Forwarder will then route the packets back to the client (no.85).