Computer Net work Project: Simulation of cloud-computing

Auther: Haochen Xie
USC ID: 4759523759
Last edit time: 03/31, 2019

This is the menual for the project over view and codes. If you can't see this in the right format, there is also a README.pdf avilable.

Purpuse

This project is about the simulation of a simple cloud-computering model. The storage and computing occurs at different servers controled by a main server. Client send infomation to the main server and save it in storage server. When computering, the infomation is retrived by the main server and send to the computering server. This projects also makes you familier with connection upon TCP/UDP socket and the compile of c/c++ programs.

How to use and examples

Boot up

At the begining, there should be 5 cpps and 1 MAKEFILE in the folder.

- Open a terminal, change direction to the foulder, input make. 5 software will be gererated : awsoutput, client, monitoroutput, serverAoutput, and serverBoutput. Warnings can be ignored.
- Open 3 terminals, input `make aws`, `make serverA`, `make serverB` respectively.
  These will run aws, serverA, and serverB.
- Open a terminal, input `make_monitor`.
  Remember to run monitor after the aws. The monitor will try to connect to aws upon setup.
- Open a terminal, input `./client <write> <BW> <Link length> <Velocity> <Noise power>` or `./client <compute> <Link id> <File length> <Signal power>`.
  - The arguments should be sperated by space. No `<` or `>` needed.
  - BW: Mhz, Link length: km, Velocity: km/s, Noise power: db10, File length: bit, Signal power: db10.
  - The message should be no longer than 100 charcaters, and for each should be less than 20 charcaters.
  - Wrong input of operation label **WILL NOT BE DETECTED**. Nothing will happen if the operation name you inputed is incorrect.
- Each terminal should echo their messages, respectively.

Examples

- A correct input for write and compute:
  Client:

```
ee450@ee450:~/Desktop/ee450_xiehaoch_session1$ ./client write 25 3 200000
-90
The client is up and running.
The client sent write operation to AWS
The write operation has been completed successfully

ee450@ee450:~/Desktop/ee450_xiehaoch_session1$ ./client compute 1 10000 -30
The client is up and running.
The client sent ID=1, size=10000, power=-30 to AWS
The delay for link <1> is <0.04> ms
```

  AWS:

```
ee450@ee450:~/Desktop/Done$ make aws
make: Circular aws <- aws dependency dropped.
./aws
The AWS is up and running.
The AWS received operation <write> from the client using TCP over port
<24759>
The AWS sent operation <write> and arguments to the monitor using TCP over
port <25759>
The AWS sent operation <write> to Backend-Server A using UDP over port
<23759>
The AWS received response from Backend-Server A for writing using UDP over
port <23759>
The AWS sent result to client for operation <write> using TCP over port
<24759>
The AWS sent write response to the monitor using TCP over port <25759>

The AWS received operation <compute> from the client using TCP over port
<24759>
The AWS sent operation <compute> and arguments to the monitor using TCP over
port <25759>
The AWS sent operation <compute> to Backend-Server A using UDP over port
<23759>
The AWS received response from Bacnend-Server A for writing using UDP over
port <23759>
The AWS sent link ID = <1>, size = <10000>, power = <-30>, and link
infomation to Backend_Server B using UDP over port <23759>
The AWS received outputs from Backend_Server B using UDP over port <23759>
The AWS sent result to client for operation <compute> using port <24759>
The AWS sent compute results to the monitor using TCP over port <25759>
```

  Server A:

```
ee450@ee450:~/Desktop/Done$ make serverA
make: Circular serverA <- serverA dependency dropped.
./serverA
The server A is up and running using UDP on port 21759.
Server A received input for writing
The Server A wrote link <1> to database

Server A received input <1> for computering
The Server A finished sending the search result to AWS
```

Server B:

```
ee450@ee450:~/Desktop/Done$ make serverB
make: Circular serverB <- serverB dependency dropped.
./serverB
The server B is up and running using UDP on port 22759.
The Server B received link information: link <1>, file size <10000>, and
signal power <-30>
The server B finished the calculation for link <1>
The server B finished sending the output to AWS
```

Monitor:

```
ee450@ee450:~/Desktop/ee450_xiehaoch_session1$ make monitor
make: Circular monitor <- monitor dependency dropped.
./monitoroutput
The monitor is up and running.
The monitor received BW = <25>, L = <3>, V = <200000> and P = <-90> from the
AWS
The write operation has been completed successfully
The monitor received link ID = <1>, size = <10000>, power = <-30> from the
AWS
The result for link <1>:
Tt = <0.03> ms
Tp = <0.01> ms
Delay = <0.04> ms
```

- If you try to compute without write first, or the link-id has not been saved, `Link not found` would be shown on cliet and monitor.

```
The client is up and running.
The client sent ID=3, size=10000, power=-30 to AWS
Link ID not found
```

Same messages would be shown if you try to use a link whoes id has not been saved.

- If you used wrong input operation, nothing will happen

```
ee450@ee450:~/Desktop/Done$ ./client computee 1 10000 -30
The client is up and running.

ee450@ee450:~/Desktop/Done$
```

Details on codes

Overview

Programs communicating through the socket. For server side, the socket should be setup with protocol. Then the socket got bind with its own ip address and port number. The socket then listen to the port if it's a stream socket, and accept and creat child socket for communication. The datagram socket do not have to do this.
The socket packet for C++ is process oriented. So I packet it to class Socket for convenient. Class socket is simple, actually. The only private member is an int, the socket decipher. It has server member functions to achieve the bind, listen, send, etc. by `Bind_()`, `Listen_()`, `Send_()`, etc..

```
    class Socket
    {
        int socket_;
        public:
        Socket(char* type);
        Socket(int s);
        int Set_(int n);
        int Bind_(struct sockaddr_in addr);
        int Listen_(int port);
        int Accept_(struct sockaddr_in *addr_dist, socklen_t sin_size);
        int Connect_(struct sockaddr_in addr_dist);
        int Recv_(char *msg);
        int Send_(char *msg);
        int Sendto_(char *msg, struct sockaddr_in addr_their, int tolen);
        int Recvfrom_(char *msg, struct sockaddr_in *addr_their, socklen_t
addr_len);
        int Close_(void);
        int Getsockname_(void);
    };
```

To initiate a socket, use `Class Socket sock("TYPE")`, TYPE for TCP or UDP. Other function can be donw by `sock.Bind_(addr)`, `Sock.Listen_(port)`.
By doing this can greatly save time for communication.

Client

Client receive input from terminater. Then the message is sent to AWS server for futher storage or computing. The format for input is

```
write <BW> <Link length> <Velocity> <Noise power>
compute <Link_id> <File length> <Signal power>
```

each argument seprated by space without "<>".
Arguments should be in such units:

- BW: Mhz
- Link length: km
- Velocity: km/s
- Noise power: db10
- Link id: integral
- File length: bits
- Signal power: db10

There is a simple input check mechanism. Client would count the number of input arguments and exit if it's less than 4. `argv[0]`~`[4]` save all the inputs.
A TCP socket is made after the check. This socket, `socket_aws`, is to connect to the AWS server, send and receive message. Notice that for c++ socket, a socket without `bind()` will not be bind to a certain port number, i.e., the port number would be autoly and dynamicaly assigned.
The `msg_send` comes from the inputs. It is generated by the following rules. If the operation is write(`argv[1]`), then get all inputs like `meg_send = argv[0]&argv[1]&argv[2]&argv[3]&argv[4]&`, which is `<write>&<BW>&<Link length>&<Velocity>&<Noise power>&`.
If it's compute, the message is `<compute>&<Link id>&<File length>&<Signal power>&<0>`. The mark `&` is for sepration.
The receive from AWS is also simple. Just check the first unit of the receive message.

AWS(Amazon Web Server)

AWS is the most important main server in the project. It receives and sends message from client, server A and B, and monitor.
In my code, all messages shown on monitor is created by AWS and sent to the monitor.
A TCP socket is created, binded, and listened. Every time a connection from client, the message is decoded by function `decode_msg()`. The message should be the following format:

```
<write>&<BW>&<Link length>&<Velocity>&<Noise power>&
<compute>&<Link id>&<File length>&<Signal power>&<0>
```

Arguments should be in such units:

- BW: Mhz
- Link length: km
- Velocity: km/s

- Noise power: db10
- Link id: integral
- File length: bits
- Signal power: db10

For write, the AWS directly send receive message to server A to store and then receive the acknowledge from server A.
On the other hand, if it is compute, the AWS send message to A anyway and receive link information from A. Then 2 messages are packet to this:

```
<compute>&<Link id>&<File length>&<Signal power>&<0><Linkid>&<BW>&<Link length>&
<Velocity>&<Noise power>&
```

and sent to server B and receive the compute result.

Server A(storage)

Server A is a storage server. It provide two functions: write message and search.
A UDP socket is created and binded to its own ip address and port number. It keep trying to receive message from the socket. The message should be the following format:

```
<write>&<BW>&<Link length>&<Velocity>&<Noise power>&`
<compute>&<Link id>&<File length>&<Signal power>&<0>`
```

Arguments should be in such units:

- BW: Mhz
- Link length: km
- Velocity: km/s
- Noise power: db10
- Link id: integral
- File length: bits
- Signal power: db10

The first argument is checked to decide its function. If it is write, the message would be save to `database.txt` by the function `write_link()`. The initial link id, if not exsist, is 1. Or it will find out the greatest link id, if exists, and save the new link with link id + 1.
Samilarlly, function `search_link()` is for searching the given link id and return the link information, if exsists. If the link is found, the message for AWS is `<Linkid>&<BW>&<Link length>&<Velocity>&<Noise power>&`. Else it is `0&1&2&3&4&`, for the first `0` tells the AWS that LINK NOT FOUND.
In the case of compute, the units of arguments are not required.

Server B(computing)

Server B is a computing server, and it's structure is simple. It passively opens.
Like server A, a UDP socket is created and binded to its own ip address and port number. The

message server B received should be like

```
<compute>&<Link id>&<File length>&<Signal power>&<0><Linkid>&<BW>&<Link length>&
<Velocity>&<Noise power>&
```

Arguments should be in such units:

- BW: Mhz
- Link length: km
- Velocity: km/s
- Noise power: db10
- Link id: integral
- File length: bits
- Signal power: db10

Then the transmission delay, propagation delay, and end-to-end time is calculated and resent to AWS.
Caution, the input messages are `char[]`. Transformation to `float` is needed.

Monitor

Monitor is to display information the AWS get from client or server A and B.
It starts with seting up a `socket_aws`, just like what client does, for connection with AWS. It keep trying to connect to AWS and receivd msg from AWS to show on the screen.

Reused Code

None.