CS655-Network

Programming Assginment

Xueyan Xia U82450191
Ziqi Tan U88387934

Logistics

In this assignment, we write a java program to send message between a pair of processes on different Linux machines. For part 1, we implement a simple echo client-server application. For part 2, we successfully perform round trip test and throughput measurements on this application. To measure RTT, we use TCP to send and receive messages of size 1, 100, 200, 400, 800 and 1000 bytes. To measure throughput, we use TCP to send and receive messages of size 1K, 2K, 4K, 8K, 16K and 32K bytes.

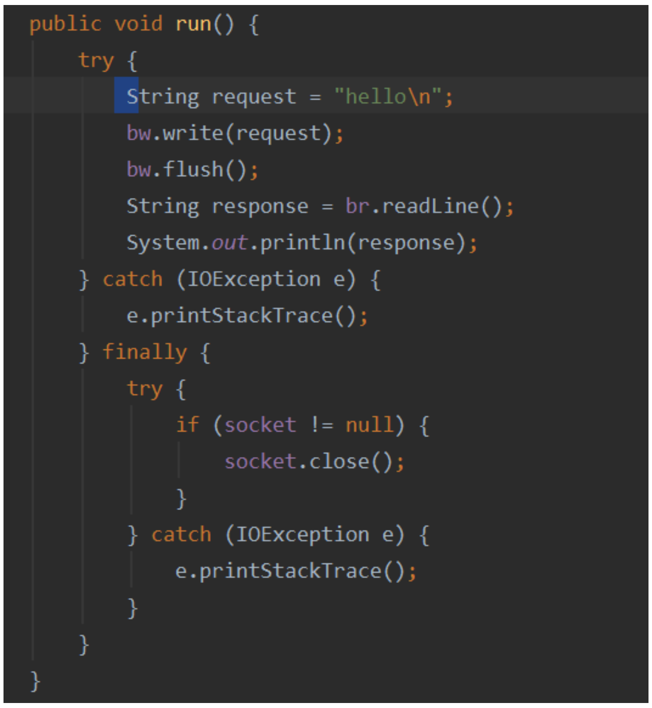
Echoing Works

We test our programs on our Microsoft Windows system with memory of 8 GB.

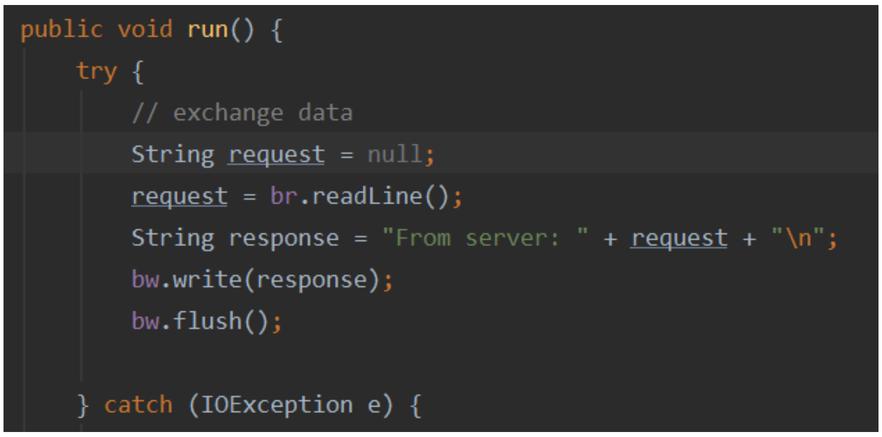
Client IP: 192.168.7.93
Server IP: 192.168.7.96

Code from slient:

We try to send a “hello” message to server.


```java
public void run() {
    try {
        String request = "hello\n";
        bw.write(request);
        bw.flush();
        String response = br.readLine();
        System.out.println(response);
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        try {
            if (socket != null) {
                socket.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Code from sever:


```java
public void run() {
    try {
        // exchange data
        String request = null;
        request = br.readLine();
        String response = "From server:  " + request + "\n";
        bw.write(response);
        bw.flush();

    } catch (IOException e) {
```

Result from client:

We get back the same message “hello” from the server successfully.

Error Conditions

At the server end, we would test the following error cases, if any happens, response 404 to client, and once the client gets any 404 response, it end the connection and close the socket.

Setup connection:
(1) The request message should be split to 4 or 5 elements by delimiter “ ”.
(2) The first element should be “s”
(3) The second element should be “rtt/tput”.

Measurement:
(1) The request message should be split to 3 elements by delimiter “ ”.
(2) The first element should be “m”.
(3) The second element should be the number of sequence and the value should be the expected one.

End Connection:
(1) The request message should be “t”.

Methodology

The socket program is implemented in Java with usage of Socket interface. By the accept () function, ServerSocket listens an assigned port and waits for clients to connect. Client could connect to the server directly by IP address (as server ‘s hostname) and the port. Then we could exchange data in a very simple and intuitive way.

We use BufferWriter and BufferReader as IO Steam manager to avoid transport congestion.

We use Java’s API **System.currentTimeMillis()** to calculate RTT, starting from the moment we send data and ending at the moment we receive data. Sum up these values on each probe and divide by probe number = 10. Take this time as variable AveRTT (ms).

For the throughput, since the server would send the message back to clients, we send 2 * Message_size bytes data in total.

Throughput = 2 * MESSAGE_SIZE * 8 / AveRTT (kbps)

Server:

The server creates a new socket object and the corresponding thread for each client.

Its task contains the three following steps.

handleSetupConnectionRequest();
handleMeasurement();
handleEndConnectionRequest();
In each step, it would check the format of request message. If any error or exception occurs, return false. And then it while breaks while loop (for measurement probes) and directly go to “finally” codes segment. If everything goes well, it would also reach “finally” code segment and

close the socket for the specific client. In the end, the server thread completes its task and terminates.

Client:

Its thread, the corresponding task contains the five following steps.

```
setUpConnection();
getMeasureMentMessage();
measurement();
endConnection();
outputMeasurementInfo();
```

In the first three steps, every time it send a request to the server, it would check the response. If the response starts with “404” or any exception occurs, it would directly go to “finally” codes segment. If everything goes well, it would also reach “finally” code segment and close its socket.

Graphs and Discussion

We both test program on Linux System and get the following results.

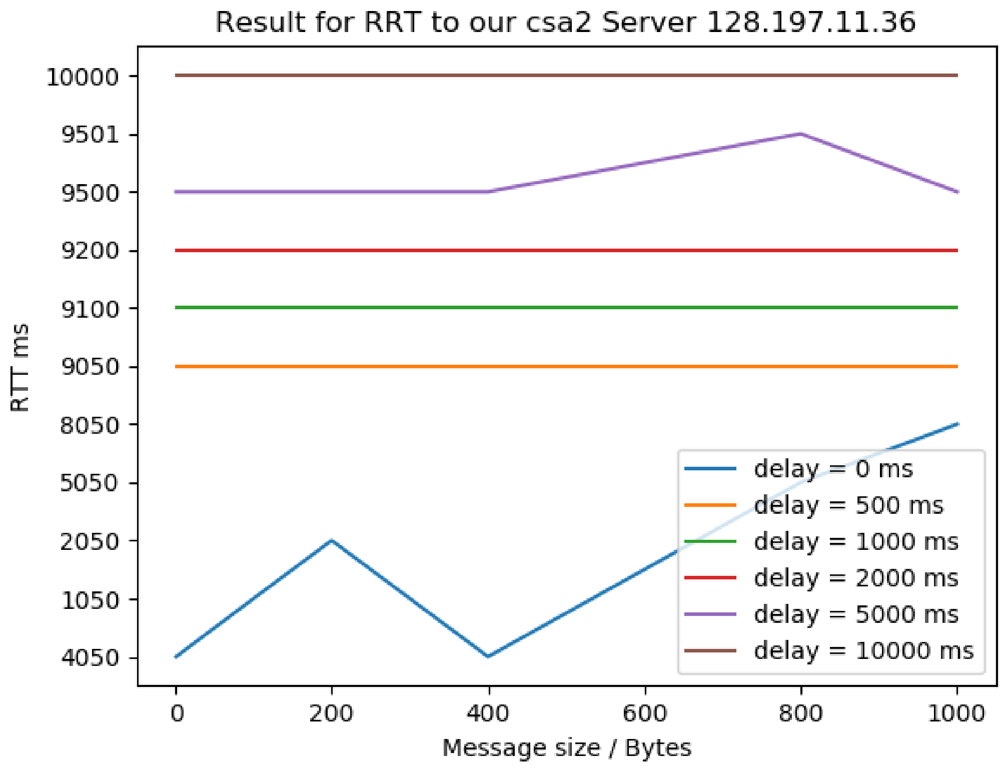
(1) Our own implements of server and client

Location: Server: ziqi1756@csa2.bu.edu, IP: 128.197.11.36 Client: ziqi1756@csa1.bu.edu
Memory: 16GB
Number of probes = 10
RTT MessageSize = {1, 100, 200, 400, 800, 1000}; (bytes)
Delay = {100, 500, 1000, 2000, 5000, 10000}; (ms)

(The result is not in order at first because of muti-threading and we sort them for the sake of plotting.)

```
DELAY: 0 MESSAGE SIZE: 1 RTT: 4050 ms
DELAY: 0 MESSAGE SIZE: 100 RTT: 1050 ms
DELAY: 0 MESSAGE SIZE: 200 RTT: 2050 ms
DELAY: 0 MESSAGE SIZE: 400 RTT: 4050 ms
DELAY: 0 MESSAGE SIZE: 800 RTT: 5050 ms
DELAY: 0 MESSAGE SIZE: 1000 RTT: 8050 ms
DELAY: 500 MESSAGE SIZE: 1 RTT: 9050 ms
DELAY: 500 MESSAGE SIZE: 100 RTT: 9050 ms
DELAY: 500 MESSAGE SIZE: 200 RTT: 9050 ms
DELAY: 500 MESSAGE SIZE: 400 RTT: 9050 ms
DELAY: 500 MESSAGE SIZE: 800 RTT: 9050 ms
DELAY: 500 MESSAGE SIZE: 1000 RTT: 9050 ms
DELAY: 1000 MESSAGE SIZE: 1 RTT: 9100 ms
```

```
DELAY: 1000 MESSAGE SIZE: 100 RTT: 9100 ms
DELAY: 1000 MESSAGE SIZE: 200 RTT: 9100 ms
DELAY: 1000 MESSAGE SIZE: 400 RTT: 9100 ms
DELAY: 1000 MESSAGE SIZE: 800 RTT: 9100 ms
DELAY: 1000 MESSAGE SIZE: 1000 RTT: 9100 ms
DELAY: 2000 MESSAGE SIZE: 1 RTT: 9200 ms
DELAY: 2000 MESSAGE SIZE: 100 RTT: 9200 ms
DELAY: 2000 MESSAGE SIZE: 200 RTT: 9200 ms
DELAY: 2000 MESSAGE SIZE: 400 RTT: 9200 ms
DELAY: 2000 MESSAGE SIZE: 800 RTT: 9200 ms
DELAY: 2000 MESSAGE SIZE: 1000 RTT: 9200 ms
DELAY: 5000 MESSAGE SIZE: 1 RTT: 9500 ms
DELAY: 5000 MESSAGE SIZE: 100 RTT: 9500 ms
DELAY: 5000 MESSAGE SIZE: 200 RTT: 9500 ms
DELAY: 5000 MESSAGE SIZE: 400 RTT: 9500 ms
DELAY: 5000 MESSAGE SIZE: 800 RTT: 9501 ms
DELAY: 5000 MESSAGE SIZE: 1000 RTT: 9500 ms
DELAY: 10000 MESSAGE SIZE: 1 RTT: 10000 ms
DELAY: 10000 MESSAGE SIZE: 100 RTT: 10000 ms
DELAY: 10000 MESSAGE SIZE: 200 RTT: 10000 ms
DELAY: 10000 MESSAGE SIZE: 400 RTT: 10000 ms
DELAY: 10000 MESSAGE SIZE: 800 RTT: 10000 ms
DELAY: 10000 MESSAGE SIZE: 1000 RTT: 10000 ms
```



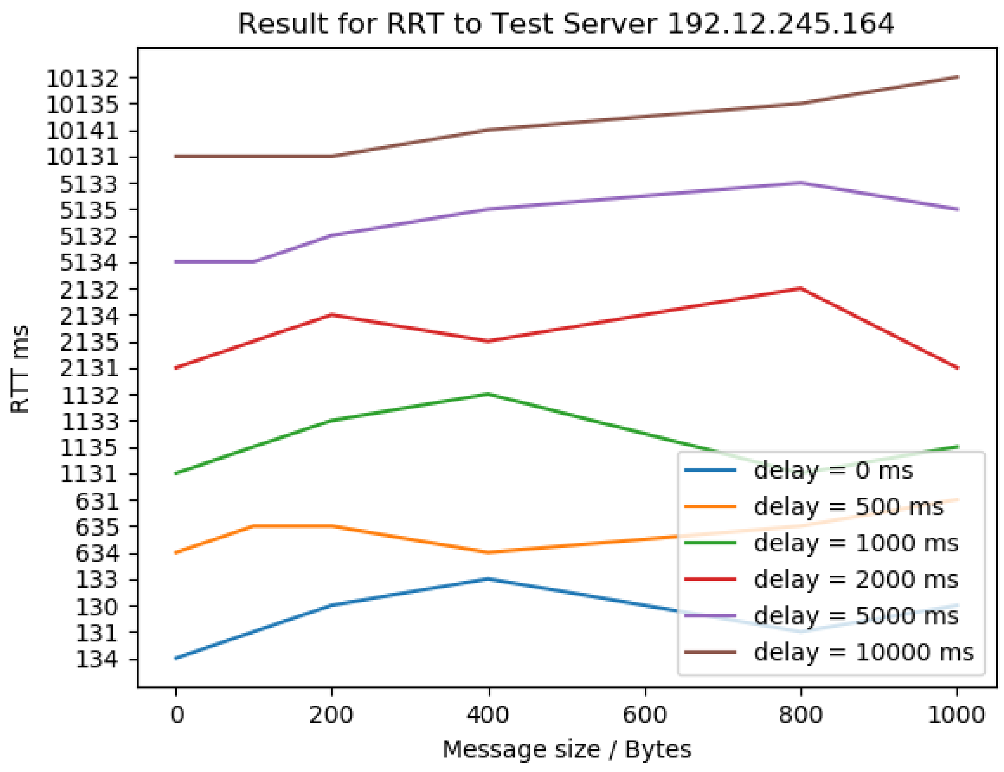
From the above result we could notice in the scenario of muti-threading, the RTT seems to be stable. This may be because Java would do some optimization and make use of resource like CPU and memory sufficiently. This different result (compared to test server may be because of scheduling of threads on the server end.

(2) Our own implements of client and test server

Location: ziqi1756@csa1.bu.edu Server IP (hostname): 192.12.245.164
Memory: 16GB
Number of probes = 10
RTT MessageSize = {1, 100, 200, 400, 800, 1000}; (bytes)
Delay = {100, 500, 1000, 2000, 5000, 10000}; (ms)

```
DELAY: 0 MESSAGE SIZE: 1 RTT: 134 ms
DELAY: 0 MESSAGE SIZE: 100 RTT: 131 ms
DELAY: 0 MESSAGE SIZE: 200 RTT: 130 ms
DELAY: 0 MESSAGE SIZE: 400 RTT: 133 ms
DELAY: 0 MESSAGE SIZE: 800 RTT: 131 ms
DELAY: 0 MESSAGE SIZE: 1000 RTT: 130 ms
DELAY: 500 MESSAGE SIZE: 1 RTT: 634 ms
DELAY: 500 MESSAGE SIZE: 100 RTT: 635 ms
DELAY: 500 MESSAGE SIZE: 200 RTT: 635 ms
DELAY: 500 MESSAGE SIZE: 400 RTT: 634 ms
DELAY: 500 MESSAGE SIZE: 800 RTT: 635 ms
DELAY: 500 MESSAGE SIZE: 1000 RTT: 631 ms
DELAY: 1000 MESSAGE SIZE: 100 RTT: 1131 ms
DELAY: 1000 MESSAGE SIZE: 1 RTT: 1135 ms
DELAY: 1000 MESSAGE SIZE: 200 RTT: 1133 ms
DELAY: 1000 MESSAGE SIZE: 400 RTT: 1132 ms
DELAY: 1000 MESSAGE SIZE: 1000 RTT: 1131 ms
DELAY: 1000 MESSAGE SIZE: 800 RTT: 1135 ms
DELAY: 2000 MESSAGE SIZE: 1 RTT: 2131 ms
DELAY: 2000 MESSAGE SIZE: 100 RTT: 2135 ms
DELAY: 2000 MESSAGE SIZE: 200 RTT: 2134 ms
DELAY: 2000 MESSAGE SIZE: 400 RTT: 2135 ms
DELAY: 2000 MESSAGE SIZE: 800 RTT: 2132 ms
DELAY: 2000 MESSAGE SIZE: 1000 RTT: 2131 ms
DELAY: 5000 MESSAGE SIZE: 1 RTT: 5134 ms
DELAY: 5000 MESSAGE SIZE: 100 RTT: 5134 ms
DELAY: 5000 MESSAGE SIZE: 200 RTT: 5132 ms
DELAY: 5000 MESSAGE SIZE: 400 RTT: 5135 ms
DELAY: 5000 MESSAGE SIZE: 800 RTT: 5133 ms
DELAY: 5000 MESSAGE SIZE: 1000 RTT: 5135 ms
DELAY: 10000 MESSAGE SIZE: 1 RTT: 10131 ms
```

```
DELAY: 10000 MESSAGE SIZE: 200 RTT: 10131 ms
DELAY: 10000 MESSAGE SIZE: 400 RTT: 10131 ms
DELAY: 10000 MESSAGE SIZE: 100 RTT: 10141 ms
DELAY: 10000 MESSAGE SIZE: 800 RTT: 10135 ms
DELAY: 10000 MESSAGE SIZE: 1000 RTT: 10132 ms
```



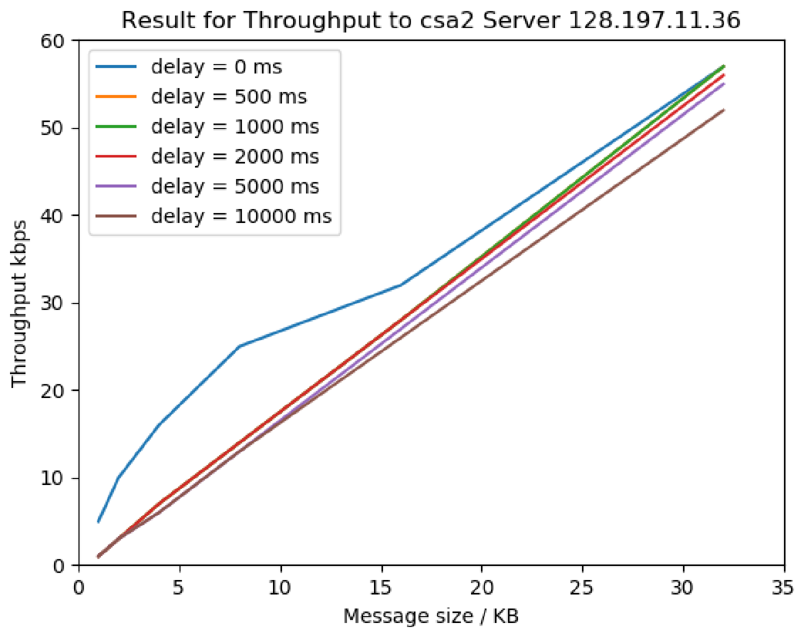
As increasing of message size, the RTT seems become larger although there are still some exceptions which is up to condition of network. Obviously, more data need more time to transport. RTT without delay of server is about 100-150ms.

3. Result of Throughput

(1) Our own implements of server and client

```
DELAY: 0 MESSAGE SIZE: 1024 Throughput: 5 kbps
DELAY: 0 MESSAGE SIZE: 2048 Throughput: 10 kbps
DELAY: 0 MESSAGE SIZE: 4096 Throughput: 16 kbps
DELAY: 0 MESSAGE SIZE: 8192 Throughput: 25 kbps
DELAY: 0 MESSAGE SIZE: 16384 Throughput: 32 kbps
DELAY: 0 MESSAGE SIZE: 32768 Throughput: 57 kbps
DELAY: 500 MESSAGE SIZE: 1024 Throughput: 1 kbps
DELAY: 500 MESSAGE SIZE: 2048 Throughput: 3 kbps
DELAY: 500 MESSAGE SIZE: 4096 Throughput: 7 kbps
```

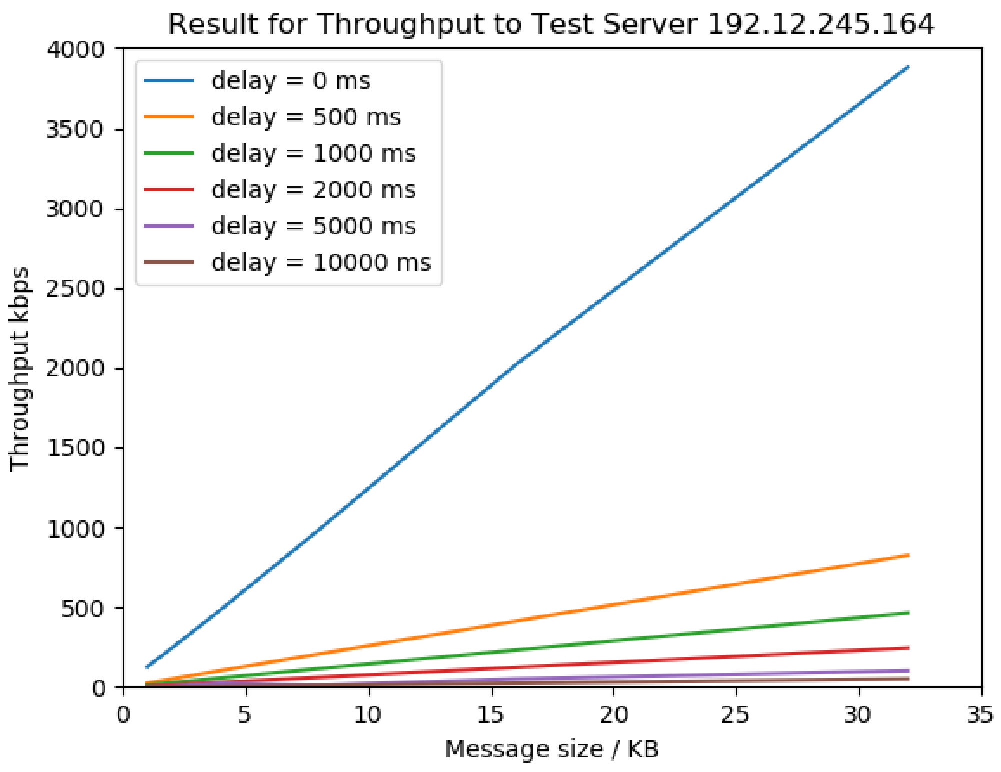
```
DELAY: 500 MESSAGE SIZE: 8192 Throughput: 14 kbps
DELAY: 500 MESSAGE SIZE: 16384 Throughput: 28 kbps
DELAY: 500 MESSAGE SIZE: 32768 Throughput: 57 kbps
DELAY: 1000 MESSAGE SIZE: 1024 Throughput: 1 kbps
DELAY: 1000 MESSAGE SIZE: 2048 Throughput: 3 kbps
DELAY: 1000 MESSAGE SIZE: 4096 Throughput: 7 kbps
DELAY: 1000 MESSAGE SIZE: 8192 Throughput: 14 kbps
DELAY: 1000 MESSAGE SIZE: 16384 Throughput: 28 kbps
DELAY: 1000 MESSAGE SIZE: 32768 Throughput: 57 kbps
DELAY: 2000 MESSAGE SIZE: 1024 Throughput: 1 kbps
DELAY: 2000 MESSAGE SIZE: 2048 Throughput: 3 kbps
DELAY: 2000 MESSAGE SIZE: 4096 Throughput: 7 kbps
DELAY: 2000 MESSAGE SIZE: 8192 Throughput: 14 kbps
DELAY: 2000 MESSAGE SIZE: 16384 Throughput: 28 kbps
DELAY: 2000 MESSAGE SIZE: 32768 Throughput: 56 kbps
DELAY: 5000 MESSAGE SIZE: 1024 Throughput: 1 kbps
DELAY: 5000 MESSAGE SIZE: 2048 Throughput: 3 kbps
DELAY: 5000 MESSAGE SIZE: 4096 Throughput: 6 kbps
DELAY: 5000 MESSAGE SIZE: 8192 Throughput: 13 kbps
DELAY: 5000 MESSAGE SIZE: 16384 Throughput: 27 kbps
DELAY: 5000 MESSAGE SIZE: 32768 Throughput: 55 kbps
DELAY: 10000 MESSAGE SIZE: 1024 Throughput: 1 kbps
DELAY: 10000 MESSAGE SIZE: 2048 Throughput: 3 kbps
DELAY: 10000 MESSAGE SIZE: 4096 Throughput: 6 kbps
DELAY: 10000 MESSAGE SIZE: 8192 Throughput: 13 kbps
DELAY: 10000 MESSAGE SIZE: 16384 Throughput: 26 kbps
DELAY: 10000 MESSAGE SIZE: 32768 Throughput: 52 kbps
```



The throughput just increase with message size and delay time could not cause huge difference under muti-threading circumstance.

(2) Our own implements of client and test server

```
DELAY: 0 MESSAGE SIZE: 1024 Throughput: 126 kbps
DELAY: 0 MESSAGE SIZE: 2048 Throughput: 244 kbps
DELAY: 0 MESSAGE SIZE: 4096 Throughput: 485 kbps
DELAY: 0 MESSAGE SIZE: 8192 Throughput: 985 kbps
DELAY: 0 MESSAGE SIZE: 16384 Throughput: 2016 kbps
DELAY: 0 MESSAGE SIZE: 32768 Throughput: 3883 kbps
DELAY: 500 MESSAGE SIZE: 1024 Throughput: 25 kbps
DELAY: 500 MESSAGE SIZE: 2048 Throughput: 51 kbps
DELAY: 500 MESSAGE SIZE: 4096 Throughput: 103 kbps
DELAY: 500 MESSAGE SIZE: 8192 Throughput: 207 kbps
DELAY: 500 MESSAGE SIZE: 16384 Throughput: 412 kbps
DELAY: 500 MESSAGE SIZE: 32768 Throughput: 825 kbps
DELAY: 1000 MESSAGE SIZE: 1024 Throughput: 14 kbps
DELAY: 1000 MESSAGE SIZE: 2048 Throughput: 28 kbps
DELAY: 1000 MESSAGE SIZE: 4096 Throughput: 57 kbps
DELAY: 1000 MESSAGE SIZE: 8192 Throughput: 115 kbps
DELAY: 1000 MESSAGE SIZE: 16384 Throughput: 231 kbps
DELAY: 1000 MESSAGE SIZE: 32768 Throughput: 463 kbps
DELAY: 2000 MESSAGE SIZE: 1024 Throughput: 7 kbps
DELAY: 2000 MESSAGE SIZE: 2048 Throughput: 15 kbps
DELAY: 2000 MESSAGE SIZE: 4096 Throughput: 30 kbps
DELAY: 2000 MESSAGE SIZE: 8192 Throughput: 61 kbps
DELAY: 2000 MESSAGE SIZE: 16384 Throughput: 123 kbps
DELAY: 2000 MESSAGE SIZE: 32768 Throughput: 245 kbps
DELAY: 5000 MESSAGE SIZE: 1024 Throughput: 3 kbps
DELAY: 5000 MESSAGE SIZE: 2048 Throughput: 6 kbps
DELAY: 5000 MESSAGE SIZE: 8192 Throughput: 25 kbps
DELAY: 5000 MESSAGE SIZE: 4096 Throughput: 12 kbps
DELAY: 5000 MESSAGE SIZE: 16384 Throughput: 51 kbps
DELAY: 5000 MESSAGE SIZE: 32768 Throughput: 102 kbps
DELAY: 10000 MESSAGE SIZE: 1024 Throughput: 1 kbps
DELAY: 10000 MESSAGE SIZE: 2048 Throughput: 3 kbps
DELAY: 10000 MESSAGE SIZE: 4096 Throughput: 6 kbps
DELAY: 10000 MESSAGE SIZE: 8192 Throughput: 12 kbps
DELAY: 10000 MESSAGE SIZE: 16384 Throughput: 25 kbps
DELAY: 10000 MESSAGE SIZE: 32768 Throughput: 51 kbps
```



The result of throughput is similar to the above result on server under the circumstance of without delay time. However, compared to delay time, message size could not affect throughput that much.