Efficient Monitoring of socket statistics in a client-server TCP connection

Ahmik Virani (ES22BTECH11001)
Devraj (ES22BTECH11011)
Divra Rajparia (ES22BTECH11013)
Gupta Dhawal Pravin (ES22BTECH11015)
Prashans Chauhan (ES22BTECH11028)
Praveen Prakash Ayila (ES22BTECH11029)

This report has 6 main sections:

1. **Background and Motivation**: This section explains why we chose this particular problem statement. We give a brief background into it, and then we explain what motivated us to work on this problem.
2. **High level Approach:** Which explains our high level approach to the problem statement and the chronological order of steps taken to solve it.
3. **Program Details:** Which explains some assumptions we made, and the steps we took to check correctness of our code.
4. **CPU and Memory Utilization**: This explains how CPU and Memory Utilization is affected in the presence of a user-space monitor program.
5. **Our tool architecture**: Here we try to explain the implementation of our program along with the block diagram and the pseudocode.
6. **Future prospects and Conclusion**: We conclude and reiterate our results along with possible future optimizations.

1 Background and Motivation

In real world network topologies, there is immense amount of data flowing between different nodes. There are multiple sockets, and hence lot of possible points of failure in the topology.

In this scenario, monitoring of different TCP statistics, and maintaining socket logs can really help us in detecting and fixing performance related problems.

In the past, there have been various such monitoring tools, but most of them are really heavy in terms of CPU and memory utlization.

For our project, we have taken inspiration from SNAP(Scalable Network-Application Profiler), a light weight profiler which efficiently collects TCP statistics, and socket logs. We have tried to implement our own tool on similar lines, and demonstrated the results of our experiments in the following sections.

2 High Level Approach

We have implemented a network monitoring tool to collect TCP statistics and socket logs for a connection. Now here's the challenge - most monitoring tools are extremely heavy and utilize high amount of system resources. Here, we have taken inspiration from SNAP(Scalable Network-Application Profiler) and come up with our own monitoring program which efficiently collects statistics, which can be used for downstream tasks such as detection of network performance problems.

2.1 Network Topology

We have implemented monitoring of socket logs and TCP statistics for a **full duplex client server TCP connection**, with loss.

2.2 Monitoring

We have implemented monitoring as a separate script which collects data about the client socket while it is communicating with the server. We have explained the details of the measured statistics, and the commands used to achieve them in the sections to follow.

2.3 System resources utilization

The goal of our experiment was to compare the system resource (CPU and memory) utilization difference when we are monitoring/not-monitoring. We used the vmstat command to observe the CPU and memory utlilization in both cases, and have shown our results below.

3 Program Details

3.1 TCP full duplex

Code from the book *Networking: A Top-Down Approach* was used as a base for the TCP implementation.

Both server and client sides were equipped with functions to send and receive bytes from the connection socket.

The next challenge was to have the server and client programs send and receive data at the same time. This was accomplished with the help of multithreading. Two threads were created in both the server and client programs, one for sending the data and one for receiving the data.

3.2 Monitoring - intervals (PASTA Property)

Instead of calculating the sample statistics continiously, which would always consume the cpu, we could do this over intervals which is defined by a poisson distribution. The arrival rates of packets into a socket could be assimed to follow the PASTA (oisson Arrivals See Time Averages) property. This assumes that the arrivals occur independtly and uniformly. The formula for poisson distribution is

$$p = \frac{e^{-\lambda}\lambda^k}{k!}$$

In our context,

- λ= average time interval between successive acks
- k=number of expected packets/chunk arrivals, here we assume it to window size.

In the above mentioned formula, p is the probability of exactly k packets arriving in the time interval $t = \frac{1}{\lambda}$

Inter-arriving rate changes with time, thus the estimated time interval is dynamic, and we take the average of the inter-arriving times. We observe statistics after every t_i seconds where t_i is the time interval estimated for the i^{th} time.

3.3 Monitoring - Statistics Collected

We collected a total of 8 statistics that we thought are essential to understand any socket.

1. **CurAppWQueue:** This statistic measures the number of bytes in the sender's queue. We used the `netstat -tn` command to get the `SEND-Q` statistic.
2. **MaxAppWQueue:** This measures the maximum number of bytes in the sender's queue throughout the lifetime of the socket. To do this, we simply took the `SEND-Q` values collected over the Poisson distribution and then took the maximum value over it. There is a little room for error here (in the case that the maximum queue occurs at an instant that we did not monitor the socket), but the chance of error is minimized because of the **PASTA property**.
3. **#FastRetrans:** To find the number of fast retransmissions, we used the `cat /proc/net/netstat` command to read the system TCP files and get the number of retransmissions. Please note, this is a cumulative statistic.
4. **#Timeouts:** Same as above, reading the TCP files using `cat /proc/net/netstat` gave us this statistic. This is also cumulative.
5. **SampleRTT:** This measures the total number of RTT samples (only acked ones) in a certain time interval. We measured it by taking a sample of packets from a TCP dump and analyzing them. The command to capture the TCP dump in a `pcap` trace file was:

```
sudo tcpdump -i h1-eth0 tcp port 20001 -w capture.pcap
```

Now, we used this command to analyze all the RTTs in the dump:

```
tshark -r capture.pcap -Y 'tcp.analysis.ack_rtt' \
-T fields -e tcp.analysis.ack_rtt
```

6. **SumRTT:** This takes the sum of all the samples we collected. Basically, the sum of RTTs which we obtained in the previous command for **SampleRTT**.

7. **SentBytes:** This stores the number of bytes that have been sent in the lifetime of the socket. We calculate it with the help of the `segs_out` parameter of `ss -i`.

   Please note, SentBytes is generally directly available in `ss -i`, we do not need to do segs_out * bufferSize. But here, in our virtual machine, this statistic was not showing, probably due to an older version of kernel. Thus, we had to do this. But in real deployment, `ss -i` can be directly used.

8. **Cwin:** This stores the size of the current congestion window. We have calculated it with the help of the command `ss -i`, which gives the value of `cwnd`.

9. **Rwin:** This stores the size of the announced receiver window. We calculated this by using the TCP dump and sampling the receiver window value for a random packet. The command used to analyze the `pcap` trace file (generated earlier) is:

   ```
   tcpdump -nn -tt -r capture.pcap 'tcp and src port 20001'
   ```

4 CPU and Memory Utlilization

The reason we are dividing the polling process into intervals was to ensure the heavy work of CPU to collect socket statistics is divided and at the same time ensure that we are able to monitor the required data in frequent intervals.

To monitor the CPU and Memory utilization, we have used the command 'vmstat' and 'free -k' respectively.

The command 'vmstat' returns a collection of hardware related utilization parameters, one of which is CPU idle (cpu_id) value in a range of 100. To find the CPU utilization percentage, we have subtracted CPU idle value from 100:

$$cpu\_util = 100 - cpu\_id$$

. This is the CPU utilization percentage.

Among the values that the command 'free -k' returns, the ones of our interest are the total memory space (tot_mem) and available memory space (ava_mem). From these two metrics, the memory utilization (mem_util) can be calculated as follows:

$$mem\_util = \frac{tot\_mem - ava\_mem}{tot\_mem} * 100$$

This gives the memory utilization in percentage.

Since we have implemented the statistics command in the user space, as expected the CPU utilization will be more when we use our monitoring program versus when we do not use our monitoring program. Also, since our monitoring program requires to store some and observe socket related metrics, thus memory utilization is also expected to increase.

We can observe these in the graphs below.

5 Our tool architecture

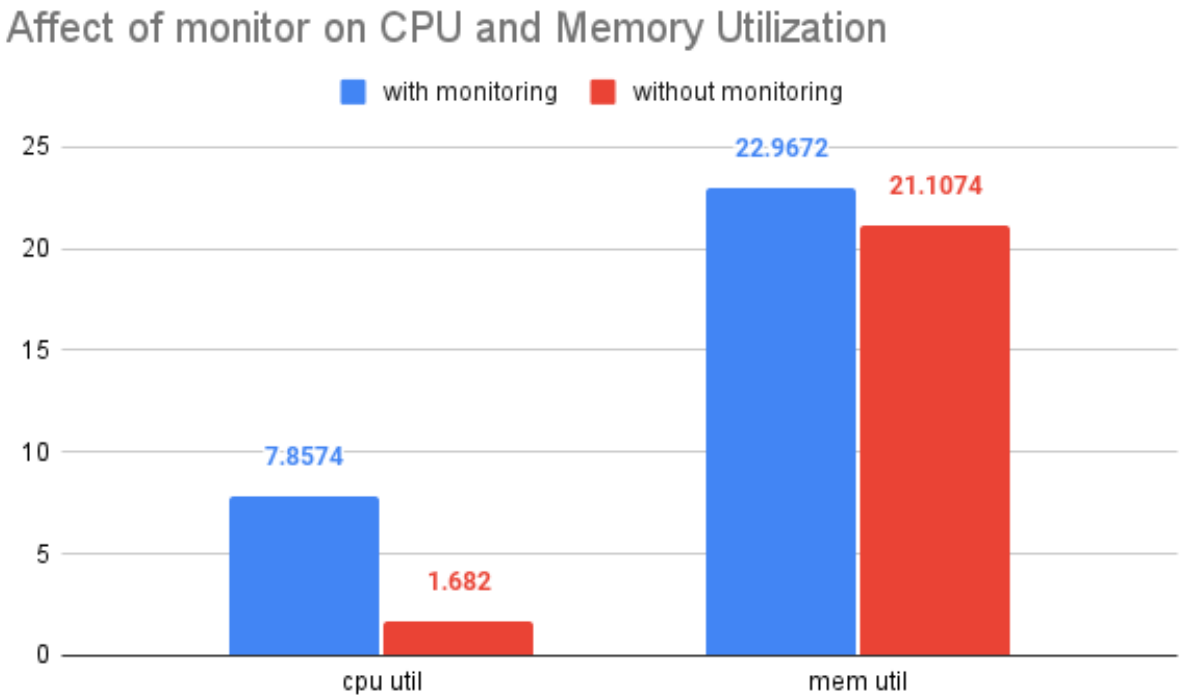

Figure 1: Block Diagram

From the graph we can see that the CPU utilization percentage changes from 1.68% to 7.85% and the memory utilization percentage changes from 21.11% to 22.96% when we use our monitor program. Now this adds a tradeoff between using a monitor to better navigate traffic in the transport and network layer versus giving more CPU utilization to the client and allow them to enjoy application layer services better. The monitor program we are using is not affecting memory too much as it does not require us to store all the data, one optimization is perhaps have a limited space of memory particularly allocated for this operation and could be in the form of a circular buffer that allows us to overwrite the data. The CPU utilization is on the other hand a little bit more significant, but in latest processors we have several type of CPU cores, which include performance (little) and efficiency (big)

cores. The little cores could be given such operations which do not require CPU intensive work and at the same time consume less energy.

6 Our tool architecture

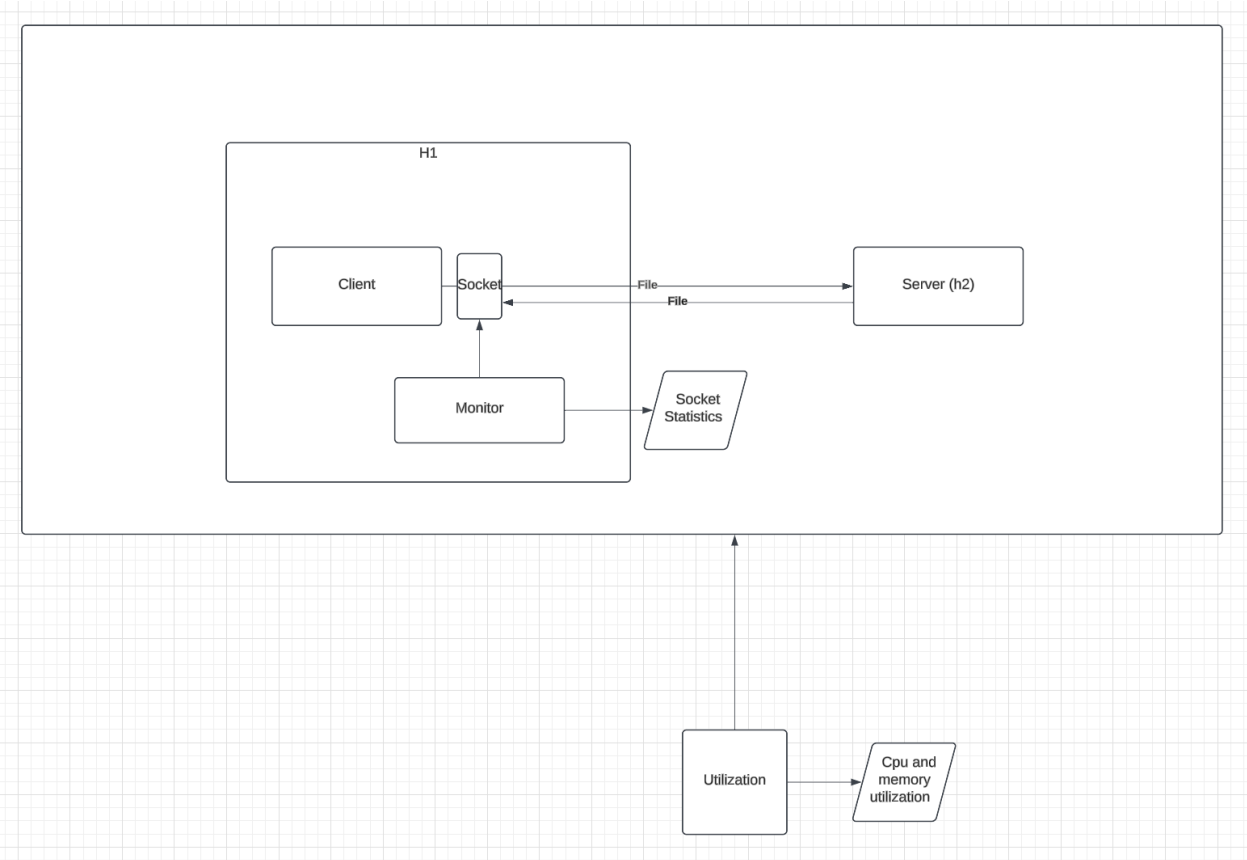

Figure 2: Block Diagram

In this section we will go through the actual implementation of our program.
Our submission consists of 4 python files which act as:

- Client
- Server
- Monitor
- CPU and Memory utilization

6.1 Experiment Configurations

All the experiments in this paper have been done on the Mininet VM version x86_64. This is the same one which has been followed for the assignment in the Computer Networks coursework.

The file used for sending is a 64.4MB video (mp4) file and the underlying channel configurations are set as:

- Bandwidth: 10Mbps
- Propagation delay: 10ms
- Packet loss rate: 10%

6.2 Client-Server Model

When we establish the connection between the client and the server, both the sides will first communicate over their sockets, and have their respective buffers. Our interest lies in monitoring the client side connection. As the two sides communicate with each other, the monitor will store the necessary statistics as mentioned in Section 3.3.

This entire process is looked over by the CPU and Memory Utilization program, which calculates their respective hardware level statistics periodically.

Here, the monitor can be turned on or off as per the need, and the effect of the monitor is show in Section 4.

6.3 Pseudocode

```
Client():
    Request connection to server
    Send files and Listen to server simultaneously

Server():
    Accept connection
    Send files and Listen to server simultaneously

Monitor():
    Find_pooling_time():
        time = Check the amount of time between two acks
        k = number of expected packets in next time interval
        pooling_time = k * avg(time)

    Find_socket_statistics():
        extract CurAppWQueue
        extract MaxAppWQueue
        extract #FastRetrans
        extract #Timeouts
        extract SampleRTT
        extract SumRTT
        extract SentBytes
        extract Cwin
        extract Rwin

    Find_socket_statistics()
    wait(pooling_time)
```

```
28
29 cpu_mem_util():
30     Calculate the cpu and memory utilization
```

7 Future prospects and Conclusion

Our implementation here was on a simple client-server topology. But in real life, data centres are massive, with lots of such nodes. Can our solution be scaled to a larger data centre?

Our monitoring tool is a simple per-socket tool. Thus, scaling it up is very easy, it just has to be installed on all the nodes whose sockets we want to monitor. Now, we would get statistics for all of these sockets, and we can use that data to make useful downstream decisions.

Thus, we conclude by stating that it is possible to monitor sockets to gain useful insights and detect possible performance problems - that too in a lightweight manner. Our results show results here are indicative of this fact, and this might be something that can really help data centres improve their performance in the future.