DDoS Load-Testing with CowStorm Client-Server Botnet

Keelan Durham and Sam Magid
Williams College
ksd2@williams.edu, sm39@williams.edu

December 14, 2024

1 Introduction

Often, the same processes that allow us to operate powerful systems distributed across a network can also be exploited by bad actors to deny service to certain resources on the web. The use of a botnet to preform Distributed denial-of-service (DDoS) attacks are an example of this. Bots are computers infected with malware that allow machines to be controlled by a user. This allows someone in control of a botnet to generate huge amounts of network traffic effectively denying service to a specific resource. In this project, we use Amazon Web Services (AWS) to create a botnet called CowStorm. We use CowStorm to preform stress testing on the Apache HTTP web-server as well as two Python servers we implemented ourselves: a basic Python web-server, as well as one which performs a SQL read and write command upon every access. To quantify the effects that our botnet has on our server we also wrote an analytic script that logs CPU utilization, connections per second, and bandwidth utilization. Our paper is laid out as follows: In Section 2 (Background) we expand on the motivation for this project covering what botnets are and common types of DDoS attacks. In Section 3 (Design/Implementation) we cover the design and implementation of our CowStorm botnet, bot programs and web-server monitoring programs. In Section 4 (Evaluation) we evaluate the effectiveness of CowStorm and its ability to stress our servers. In Section 5 (Future Work) we cover possible areas of future work, and in Section 6 (Conclusion) we draw conclusions, putting our project in context and connecting it to broader themes of the course. Finally, Section 7 (Appendix) contains all graphs and figures referenced throughout this paper.

2 Background

Before diving into the details of our CowStorm botnet implementation, it is useful to provide some relevant background information. This section introduces two key ideas from distributed computing discussed in this paper—DDoS attacks and botnets—in order to support the reader in understanding the discussion and experimentation which follows.

2.1 Botnets

A botnet is, simply, a network of computers ("bots") under the control of a master computer (the "botmaster") which issues commands to be executed on those bots. Technically, this just gives the botmaster access to greater computing

power distributed across multiple nodes; however, the term "botnet" usually refers to a network of computers which have been unknowingly infected with malware and are leveraged for malicious purposes, such as DDoS attacks.

A botnet can be organized following either a client-server architecture or a peer-to-peer (P2P) architecture—or both! Both types have the advantage of amplified power and increased anonymity—the target of a botnet attack does not easily know where the attack originates from—but a P2P botnet is especially diffuse and difficult to take down. In a client-server model, once the botmaster (or multiple botmasters) is discovered, the attack can be stopped by targetting that botmaster node; in a P2P botnet there is no centralized point of failure, making it much harder to take down.

2.2 DDoS Attacks

DDoS attacks can target several different parts of the 7 layer OSI network stack. HTTP flood attacks, for example, target Layer 7 (the application layer) of the network stack. By using a botnet to create massive amounts of network traffic we can tax server resources degrading or denying use of the service. DDoS attacks can also target the network protocol layer—specifically, SYN flood attacks exploit the TCP three-way handshake, spoofing an IP address, sending a request to connect, and then never responding. When preformed in high enough volumes this can leave a server unable to respond to valid requests. Finally, DDoS attacks can target Layer 1, the physical layer. This approach targets the bandwidth that connects a server to the internet. DNS amplification attacks are a classic example of a physical layer attack. This works by spoofing the IP address and the MAC address of a request for all domains to be that of the target. This request is then sent to an open DNS server which returns a large response to the target overwhelming the bandwidth. The key is a small request triggers a large response overwhelming the bandwidth. A botnet can also be used to attack the physical layer, although having another form of amplification can be crucial to generating the necessary traffic.

We didn't experiment with any techniques that required spoofing either IP addresses or MAC address, in an attempt to keep our project fully within the legal limits of network experimentation. For these reasons we primarily preformed attacks relying on floods of HTTP requests to stress the physical capacity our server. We also monitored our bandwidth utilization but due to our extremely strong Williams College network connection, this was difficult to exploit.

3 Design/Implementation

This section describes the design and implementation of the main three parts to our project: the botnet itself, hosted on several Amazon Web Services (AWS) EC2 VMs; the bot and botmaster programs for sending/receiving bot commands; and the web server and monitoring programs on our target machine.

3.1 AWS Botnet

We used free tier AWS services to create and maintain as many as 91 VMs at different times, all located in the Ohio AWS region. The machines all ran Ubuntu and were the AWS t2.micro instance. To effectively manage our machines, we wrote multiple bash scripts to streamline the process of sending and executing code on each machine. The first script we wrote was one to compile the list of IP addresses from our active AWS servers. Each

time we started the VM's they were given new IP addresses so this was an important step. By using AWS scripting we were able to compile a list of IP addresses and their associated keys in a CSV file. The other useful scripts we wrote were `sendFile.sh` and `multissh.sh`. `sendfile.sh` streamlined the process of distributing test code across our machines by copying a file to each IP address in our list of VM's. We used `multissh.sh` to quickly run command line arguments across all the machines. This allowed us to start programs, and configure our machines.

3.2 Bot/Botmaster Program

We implemented two different versions of our botnet. Our first implementation was written in C and essentially ran each of our bots as a server. It would open a port and listen for connections from the bot-master containing instructions. We preformed connections via telnet and wrote script `multitelnet.sh` to efficiently distribute instructions to our bots. If a telnet request started with `execute:` whatever followed would be run in the command line. We quickly realized that having each bot listening on a port for instructions wasn't the direction we wanted to go. It posed a significant security risk since anyone could use telnet to execute commands, and it also had the major disadvantage that it relied on each bot having an open port to listen on—something which we could ensure on our own VMs, but which is unrealistic in a real-world botnet implementation.

This led to the second implementation, detailed in Figure 1. This was written as a bash script which periodically requests an instruction file from a bot master server. The first line contained a boolean which told the bot if it should execute the command that followed. In this way each bot acted as a client instead of a server. This did not rely on opening ports on our bots and it made the packaging of the bot very small, allowing it to easily be installed as malware in the background without a user knowing. The major disadvantage of this system is that the bot master server is a single point of failure and is hard coded into the executable. If this botnet were to be used with malicious intent, once the botmaster IP address was identified taking down the botnet would be relatively simple.

3.3 Web Server and Monitoring

Over the course of our testing, we ran three different web servers on our target machine in an attempt to balance performance, customization, and monitoring capabilities.

We first tried the Apache HTTP Server (version 2.4.52) as the web server on our target machine. This lightweight server provided high performance, but its monitoring capability with the native `mod_status` module was quite limited: its `requests/sec` metric updated slowly and averaged over a long amount of time, the `CPU Usage` metric did not seem to provide realistic numbers, and the module did not provide a way to save performance data for further analysis or visualization. Apache also did not provide us much control over the inner-workings of the server and required using another language like PHP to implement any scripting capability.

Given these limitations, we implemented our own Python web server with the goal of building better performance monitoring and having more low-level control of the server. We initially built a simple threaded server using the `http.server` and `socketserver` packages, modifying the request handler to log visits and adding a background thread to log visit

count and CPU usage using the `psutil` package; however, we found that when we load tested the server at high capacity, the background performance-logging thread would stop functioning. Given this, we decided to move the performance-logging feature to a separate program so it could continue to function under high network traffic. Because we were logging on a separate program, we did not have immediate access to incoming requests on the same port as the server; therefore we used the `scapy` package to sniff network traffic, and recorded every unique request. Separating the program also allowed us to apply the same performance metric tools on our original Apache server.

In addition to our basic Python web server implementation, we implemented one more Python server which performed a read and write command on a backend SQLite 3 database upon every connection, keeping a log of each IP address's visit count and displaying it to the user on the homepage. As we will explain, we built this SQL-enabled server to test our hypothesis that a greater performance cost per connection would make a server easier to take down.

4 Evaluation

Here we evaluate CowStorm's DDoS load testing performance on our three test target servers: the Apache HTTP Server (version 2.4.52), our basic Python web server, and our SQL-enabled Python server. We ran all three severs on port 80, used the same Python-based monitoring program to gather performance metrics, and performed the same HTTP flood attacks against each (10,000 GET messages flooded at 1-bot, 5-bot, and 10-bot tiers). We find that all three servers were impacted under high load, however only the SQL-enabled Python server became unreachable to users under any of these three tests.

4.1 Apache HTTP Server

Figure 3 shows three statistics—bandwidth, CPU utilization, and connections per second—across our three tiers of testing on the Apache server. Of the three servers we tested on our target machine, the Apache server performed the best: during the 5-bot attack, for example, the server managed to handle around 1000 connections per second with a low CPU utilization of around 15%. In comparison, under the same attack the basic Python server handled around 450 connections per second at 20% CPU utilization, and the SQL-enabled Python server only managed to handle around 300 connections per second at 30% CPU utilization.

We found that the Apache server experienced some bottleneck or hit some limit between our 5-bot and 10-bot test: both CPU utilization and clicks per second increased significantly between the 1-bot and 5-bot tests, but relatively little between the 5-bot and 10-bot tests; bandwidth, however, scaled linearly with the number of bots used in the attack. This was a general theme across our three severs: both connections per second and CPU utilization tended to approach some maximum, increasing less and less as the number of bots increased.

4.2 Python HTTP Server

Our basic Python web server performed very similarly to the Apache server, with slightly higher CPU utilization and slightly lower connections per second. Under our 1-bot test, the basic Python server handled around 150 connections per second under 9% CPU load (compared with 200+ connections per second and 7.5% CPU load on the Apache server), increas-

ing to around 450 connections per second and 20% CPU utilization on our 5-bot test, and finally 475 connections per second and 26% CPU load during our 10-bot test (during which the Apache server handled more than twice as many connections with significantly less CPU usage).

These performance differences make sense: the Apache web server is written in C, a much more efficient language than Python. However, both servers are essentially performing the same task—serving a static `index.html` webpage on several. Given that both of our basic web server implementations remained online under all three tiers of testing, we hypothesized that a greater cost-per-connection on the server side might be the key to a successful DDoS attack.

4.3 Python HTTP Server with SQL

keelan

4.4 Comparison

Figure 2 shows the CPU usage across our three servers, under an HTTP flood attack with 10 bots each sending 10,000 requests. As we can see, the Apache server maintained the lowest CPU utilization, with the basic Python server following, and finally the SQL-enabled Python server performing the worst. This makes sense: the Apache server was written in C, a much faster and lightweight language than Python, and we would similarly expect added cost of a read and write SQL query to worsen performance. However, given that even the SQL-enabled Python server did not completely overwhelm the CPU, it seems that the most significant factor in our ability to overwhelm it was the server's lack of capacity to handle simultaneous write SQL requests. It makes sense, then, that we see a significantly longer processing time on the SQL-enabled Python server (almost 400s) compared to the two non-SQL severs (Apache and Python) which completed processing requests in under 100s.

4.5 Epilogue

At the end of the project we decided to preform a test with as many bots as we could muster. Between our two AWS accounts we were able to preform a test with 91 bots. Our tests are detailed in Figure 4. We preformed tests on both our Apache server and our SQL Python server. Our SQL server was inaccessible for the duration of the test and our target `indie2.cs.williams.edu` was unresponsive. The Apache server was inaccessible during portions of the test. We are not exactly sure what to make of the large swings in bandwidth usage, CPU utilization and connection, but it was very exciting to finally break the Apache web server!

5 Future Work

Future work on this project falls into three main categories. The first is a few small things we would have liked to complete to but didn't have time to. The second category further explores the design and implementation of a botnet, and the third involves experiments into different types of attacks.

We would have liked to have configured the Apache web server to make read and write queries to an SQL database as a comparison to the Python server. We hypothesize that, given the Apache web server would experience the same thread-protected write queue effect on the SQL database, it would also be rendered inaccessible given a relatively low-number of bots executing an HTTP flood attack.

We also would have liked to create real time graphs of CPU utilization, bandwidth usages and connections per second, in the form of a web app that would also allow a user to save performance data easily to a `.csv` file. An additional metric that would have been interesting to add would have been to issue a `curl` request periodically to `indie2.cs.williams.edu` and record the response time, allowing us to quantify more clearly how a user would experience server degradation.

Peer-to-peer frameworks make systems that can be near impossible to take down. Further work on this project could be to implement a P2P botnet where bots propagate commands among themselves. This would eliminate the single point of failure in our current implementation and make a robust and scalable botnet resistant to take down. However, it would be harder to prevent truly malicious actors from issuing their own commands to the botnet.

How each layer of the network can be exploited to deny service is another aspect of this project that could be explored further. Attacks such as SYN flood attacks or slow post attacks can be effective with smaller numbers of machines in the botnet. Our analytics could easily be adapted to quantify the effect of these attacks.

6 Conclusion

In this project, we implemented a client-server botnet, called CowStorm, which we used to perform DDoS-like load testing on our own target server. At its peak, CowStorm included 91 AWS EC2 bots (see Figure 4), but in order to stay mostly within the AWS free tier, we performed most of our tests using 10 or fewer bots.

On our target server, we experimented with three separate web server implementations: the Apache HTTP server, a basic Python web server, and a Python web server which performed a SQL read and write command upon every connection. We also implemented a program in Python to monitor various useful server statistics, which we used to evaluate the performance of our servers under different attacks. Under our basic 1-bot, 5-bot, and 10-bot flood attacks, the Apache server consistently performed the best, and only the SQL-enabled Python server became inaccessible to users, both at both the 5-bot and 10-bot tiers. This is likely due to the inability of our SQL database to execute simultaneous write commands. Notably, our final experiment with 91 AWS bots was able to take down both the Apache and Python basic web servers; however we did not perform extensive testing at this tier to avoid expensive AWS charges.

This project provided useful experience in large-count VM management, bash scripting, and cyber attacks. The experience gained in this project could be used to better understand and prevent cyber attacks from truly malicious actors. The project also covered many aspects of our course material, in a way that built upon and made us appreciate the extent of our learning this semester. We worked and gained knowledge at all levels of the network stack—sniffing network packets; crafting websites and HTTP requests; and implementing, reimplementing, and modifying many of the course projects we had done over the course of the semester. We also had a ton of fun doing it.

With an even greater appreciation for distributed system computing, and armed with new experience and understanding of cyber attacks, we promise to use our new skills for good!

7 Appendix

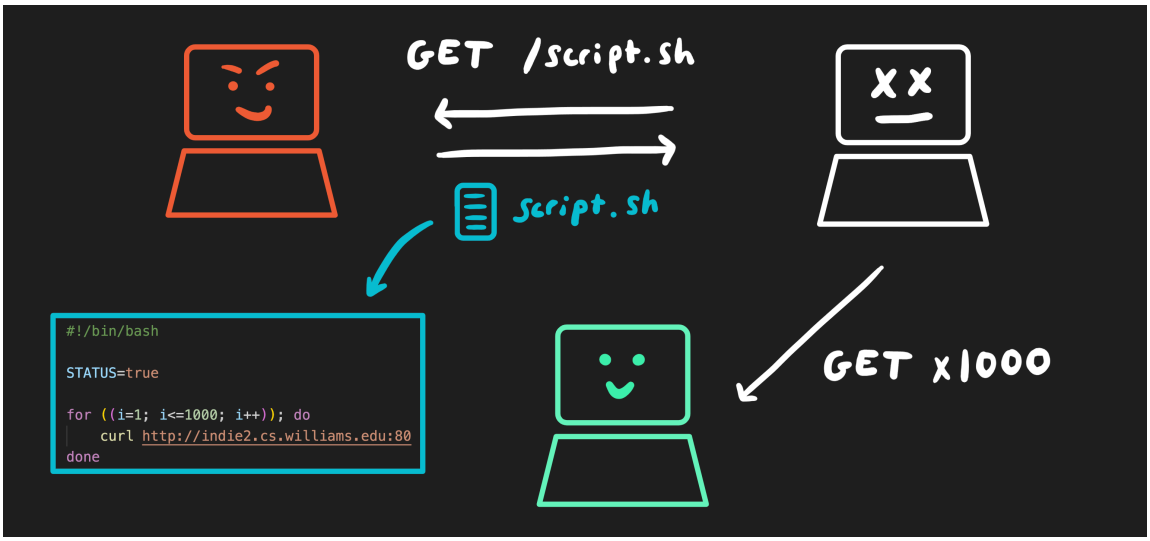

Figure 1: Diagram of our second botnet implementation, with `script.sh` serving commands.

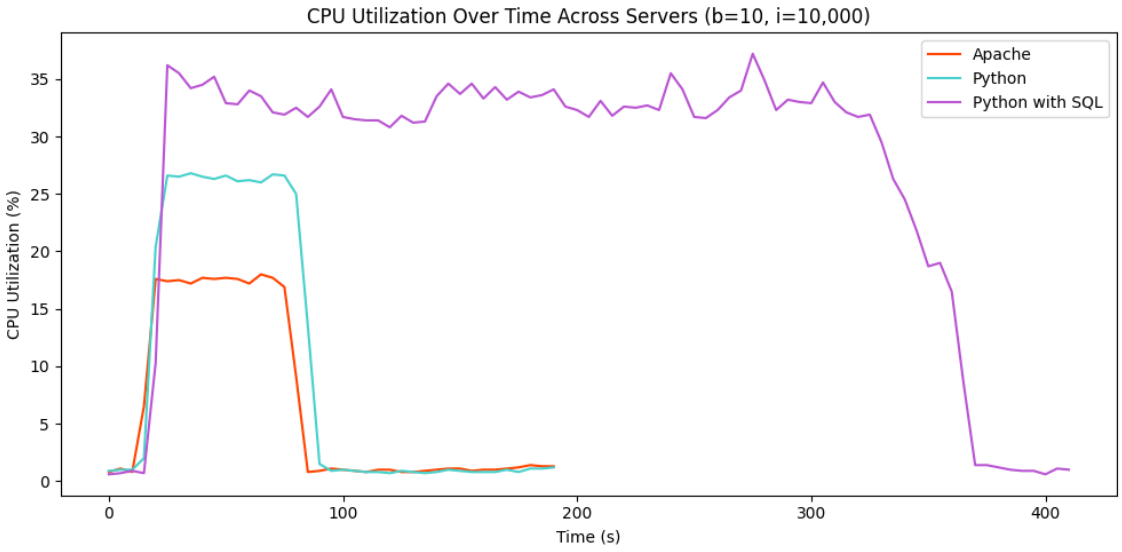

Figure 2: Graph of CPU utilization across our three test server programs during a 10-bot attack.

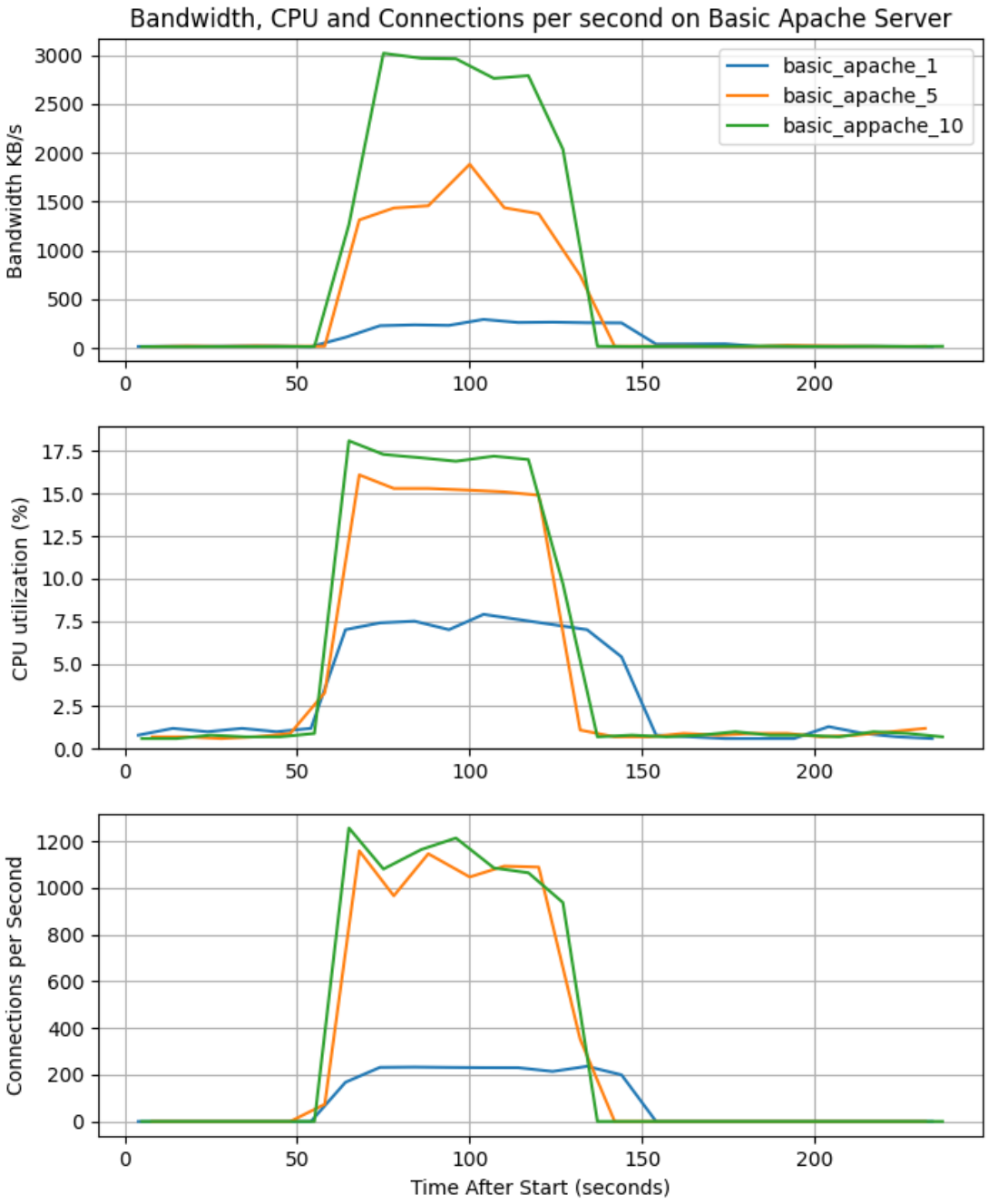

Figure 3: Graph of CPU utilization, bandwidth, and connections with 1, 5, and 10 bots

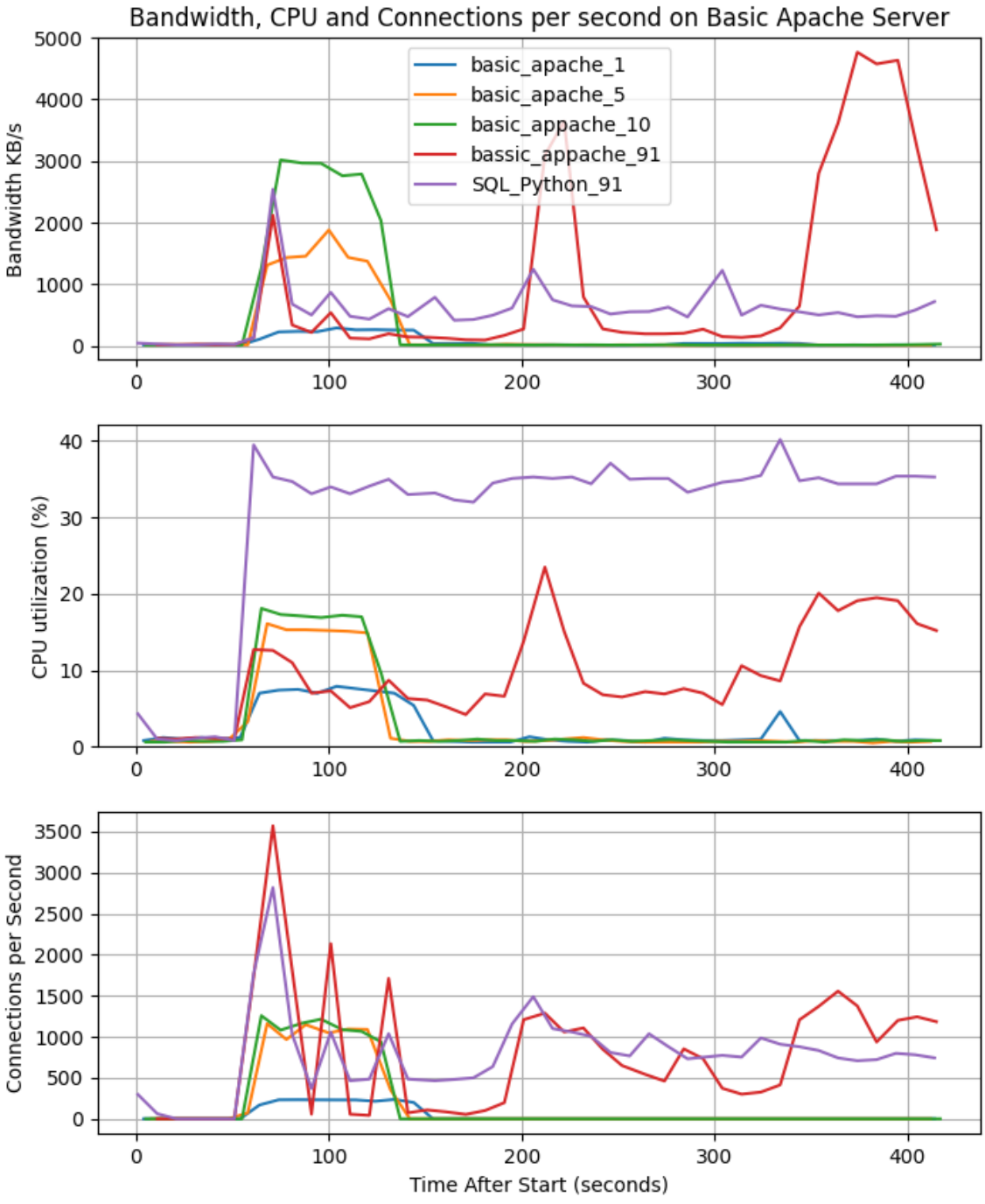

Figure 4: System usage including final test with 91 bots. At times the Apache server was inaccessible. The Python-SQL server was inaccessible for the entire test duration.