2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE)

Utilizing Minecraft Bots to Optimize Game Server Performance and Deployment

Matt Cocar, Reneisha Harris and Youry Khmelevsky
Computer Science Department, Okanagan College
Kelowna, BC V1Y 4X8, Canada
Emails: {matt.cocar, reneisha.harris}@gmail.com, ykhmelevsky@okanagan.bc.ca

***Abstract*—To simulate a realistic game server environment, we utilized open source software libraries to create automated players (bots) for the globally renowned online game: Minecraft. The fairly simple design of the Minecraft server as well as its massive development and support community facilitates considerable research and analysis prospects. As such, the goal of our investigation was to emulate and then analyze the real-world stress that game-players actively create on hosting servers. We achieved this through creating scripted movements of Minecraft characters that are connected to the Minecraft server(s) hosted within our virtual infrastructure. After this was achieved, we explored altering the methods of running the active Minecraft servers to control CPU load; we primarily explored manually setting the CPU affinity of the Minecraft server thread to run on specific virtual cores. Collecting CPU workload data while the bots were running around on our servers gave us consistent and predictable readings that confirmed the success of our methods we used to control performance. Evidence of this is illustrated through the use of graphs and other experimental data outlined in the body of this document.**

I. Introduction

In early 2014, students of Okanagan College chose to experiment with locally hosted Minecraft servers and custom designed bots that used libraries from a community developed protocol implementation named MCProtocolLib [1]. Because Minecraft's architecture is well documented and its community is rich with developer support, it was a prime candidate for experimentation with custom developed tools. Also, players can download the server application so they can host their own worlds, which means we have total control over the infrastructure it runs on.

In this paper we control the Minecraft server threads to expose how scaling from one server to ten servers increases load across the system. To start, we will examine the infrastructure of our environment; it was set up as a virtual network between the server and client(s). In this infrastructure, we have the following configuration: One virtual machine that is used to host the Minecraft servers, and several virtual machines those are used to run bots for every two Minecraft servers.

Each Minecraft server hosts 25 bots, so this translates into one virtual machine running a total of 50 bots. Following from this, testing was done on a total of 10 Minecraft servers. Therefore, this utilized resources of 5 virtual machines. For our evaluations, this gave a semi-accurate representation of real-world players that are on separate hosts and connecting to a Minecraft server. The configuration also helps the bots to run more smoothly due to spreading out their workload.

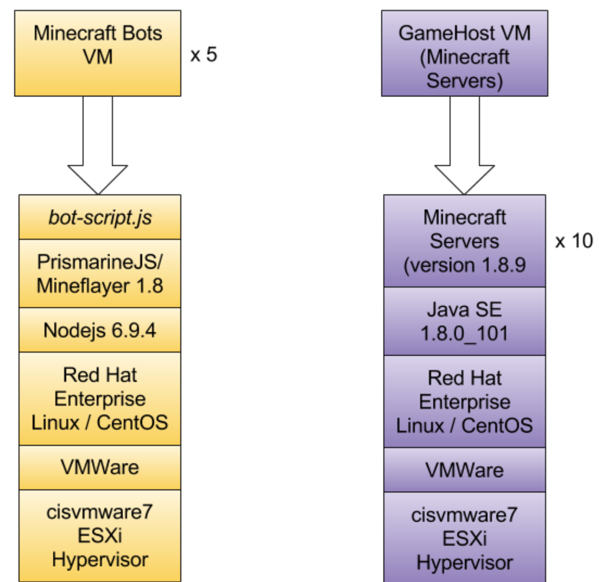

Fig. 1. Current Infrastructure Diagram

The following section has details outlining simulation tools that are currently used to evaluate game performance (see Section II). We then go on to provide a detailed illustration of the set-up of our experimental infrastructure. Accompanying this is a discussion of the results obtained through our performance analysis which also includes data collection, measurements and their associated interpretations. Section III of this document outlines the details of the bot design and its associated applications in our Minecraft server performance analysis. In Section IV, we go on further to discuss the deployment of the Minecraft servers and then in Section V we discuss the process of automating and optimizing the Bot for testing. Section IX outlines our future plans and Section X summarizes our research results.

II. Existing Works

The previous research comprised an investigation into the maximum possible workload that could persist on CentOS 6.5

Authorized licensed use limited to: The University of British Columbia Library. Downloaded on March 03,2024 at 05:25:41 UTC from IEEE Xplore. Restrictions apply.

978-1-5090-5538-8/17/$31.00 ©2017 IEEE

and CentOS 7.0 virtual servers; this workload consisted of our custom Java-based bots [1].

There are some discussion by authors surrounding interactive online games, with particular focus on "First Person Shooters" (FPS) genre [2], [3] and the accompanying network traffic for these games [4]. Their investigations explore the impact of the network on the games and also looks at realistic traffic generators.

A technical report obtained from IBM [5] demonstrates that "rapid system response time, ultimately reaching sub-second values and implemented with adequate system support, offers the promise of substantial improvements in user productivity". It is "better to implement sub-second system response for their own online systems" and usually computers are not well balanced. The system response time was divided in two categories that were deciphered as critical components such as communication time and computer response time.

One of the contributors to the library (mentioned in the next section) has programmed a bot to be able to find and navigate to another player using A* path finding [6].

III. Implementation of the Bot

In this research, the Minecraft players' (bots) actions were developed using a NodeJS library called PrismarineJS/mineflayer. We utilized their abstraction of the Minecraft network protocol: PrismarineJS/node-minecraft-protocol. This is made incredibly simple to do through use of the Mineflayer API. Some code snippets that highlight key features are provided below (see Listing 1) and illustrate some chief functionality that supported our efforts to achieve scripted player movements that allowed us to stress test our server VM.

Listing 1. createBot Function

```
...
bot.on('spawn', function () {
  var timer = 1000;
  bot.setControlState('forward', true);
  for (var i = 0; i < ITERATIONS; i++) {
   setTimeout (function () {
    bot.entity.yaw = (Math.PI * 0.5);
   }, timer );
  ...
  bot.setControlState ('forward', false);
  }, timer);
});
```

The first part of the Create Bot function connects a player to the Minecraft server. The bots we spawn are considered unregistered players, which means they need special server parameters or they will not be able to connect.

A listener is attached to the bot object and waits until the spawn event occurs, then signalling that there is now a Minecraft character placed in the game world. The bot's control state is set to forward, which tells the server that the bot is walking. The for-loop controls when movements occur. Specifically, for each second, we tell the bot to turn 90 degrees, making it walking in a square pattern.

Listing 2. makeBots; The Main function

```
...
while (bot_count < NUM_BOTS) {
  createBot(BOTNAME_PREFIX +
  bot_count);
  bot_count++;
}
```

The main function in Listing 2 invokes *createBots*. It controls how many bots the script will spawn in its instance. A separate BASH script executes the bot script 25 times to take advantage of multiprocessing.

IV. Deployment of the Minecraft servers

The version of the Minecraft server was selected based on the compatibility with Mineflayer. They include: PrismarineJS/mineflayer 1.8.0 and Minecraft server 1.8.9. We used a fairly simple approach to set up 10 Minecraft servers on one host; a base server was created and its configuration was customized for all the other servers to be cloned from it. The procedure for this includes the following steps:

- First, the server.jar file has to be run: The *nogui* argument is needed for environments without GUI environments, so in our case, this argument is necessary. The *server.properties* file is automatically generated so the user can configure server options. Its main options are denoted in Listing 3:

  Listing 3. The main server options

  ```
  generator-settings=FLAT
  level-type=FLAT
  max-players=100
  server-port=25565
  spawn-animals=false
  generate-structures=false
  online-mode=false
  ```

  where *generator-settings*, *level-type*, and *generate-structures* are for making the generated Minecraft world flat, and without any buildings to get in the way of the bots. Animal spawning (*spawn-animals*) is turned off because they spawn randomly around the map and can affect performance readings for certain cores that are assigned certain Minecraft servers exhibiting this case. *online-mode* is turned off to allow unregistered players (the bots) to connect to the Minecraft servers. Lastly, *server-port* is used to control which port a Minecraft server listens on. This option is particularly important to us because we are running all of the servers on one host, creating the need for multiple servers listening on different ports.
- After the *server.properties* are set up, deleting the created */world/* directory is necessary, so a fresh, flat world gets created. *server.jar* creates many files, so it is also necessary to have separate folders for each Minecraft server. At this point, the base server setup is ready to be

Authorized licensed use limited to: The University of British Columbia Library. Downloaded on March 03,2024 at 05:25:41 UTC from IEEE Xplore. Restrictions apply.

cloned. Simply copying the entire folder and renaming it creates the clones.

- Lastly, the new clones need to have their *server.properties* altered by setting *server-port* to a different port than the other servers. The firewall has to have all of those ports open for bots or players to connect.

V. Automating and Optimizing the Bot for Testing

Two BASH scripts were written to automate spawning and de-spawning the bots. Early tests results showed that if we used only one NodeJS process to spawn bots, they would misbehave by not always turning when they were supposed to, or sometimes they would delay for a few seconds and not move forward. Spawning one bot per NodeJS process gave better results and as a consequence of this, server load averages across cores with server affinity were more consistent when different tests were done to compare results. Below is an illustration of the scripts created for the Bot Spawner as well as the Bot Destroyer.

1) Bot Spawner: The spawning script, *bot_spawner.sh* takes in

```
<script.js> <list-of-ips> <iterations>
<bots_per_instance> <num_instances>
```

as command line arguments. *script.js*, is a JavaScript file that uses PrismarineJS/mineflayer. For our tests, we used the previously mentioned script (Listing 4).

Listing 4. scripts.js

```
for ip in $(cat $serverIPsTxt)
do
 host=$(echo $ip | awk -F: '{ print $1 }')
 port=$(echo $ip | awk -F: '{ print $2 }')
 echo "Spawning bots on $host:$port"

for group in $(seq 1 $numberOfInstances)
do
 #launches node as a background process
 node $mineflayerScript $host $port
 $iterations $botsPerInstance
 grp-$group- &
done
done
```

The first part of the code reads a list of Minecraft server IPs and ports. Each IP is a server that this script will spawn bots on. For our testing, we spawned 1 NodeJS process per bot for the best stability. *$numberOfInstances* is given 25, and *$botsPerInstance* was set to 1 (CL arguments). Each instance of NodeJS is spawned as a background process so the script can continue looping. Persistent processes caused trouble after finishing a test, as there was no easy way to kill the spawned background processes other than using *top* or *kill pid*.

2) Bot Destroyer: The bot destroyer, aptly named *bot-destroyer.sh* is a script that automates the task of killing NodeJS processes. Essentially, it makes cleaning up the bots up after tests a lot quicker:

```
ps no pid,command > user.pids
echo -n > botpids.pids
```

The records that *ps* generates are formatted ⟨pid,command⟩. *command* is the line of *BASH* that spawned the process (we want to filter out non-NodeJS processes):

```
while read proc
do
  echo $proc | grep "[n]ode.*\.js"
  | awk '{ print $1 }' >> botpids.pids
done < user.pids
```

Each record is filtered by *grep* using a tailored RegEx pattern to filter out only the NodeJS processes that we spawned. *awk* is used to output only the PID of the filtered lines:

```
rm user.pids # cleanup
while read pid
do
  kill $pid
done < botpids.pids
```

After some cleanup of the lint that previous code created, the script reads all of the freshly filtered PIDs and runs *kill* with the ith PID as an argument.

VI. Running Minecraft Servers for Testing

When the Minecraft servers run as background processes, they stall. We used the GNU Screen sessions to efficiently manage more than one running server. After the sessions are created, we attach to the first session and start a Minecraft server. Then, we detach from that session back to the main session. Next, we need to find the PID of the main Java process running the Minecraft server to run *jstack* on:

```
$ ps -n
...
24135 pts/1     Sl+
15:50 java -jar server.jar nogui
...
$ jstack 24135
```

jstack lists lots of information about each thread in a JVM instance. There is one in particular that we are interested in setting CPU affinity for:

```
$ jstack 24135 | grep ''Server thread''
Server thread ... ... nid=0x5e77
```

The $nid = 0x5e77$ part is the PID (in hex code) of the actual game server thread. To set its CPU affinity, we need the decimal conversion:

```
$ taskset -cp 0 $((0x5e77))
pid 24183's current affinity list: 0-31
pid 24183's new affinity list: 0
```

Authorized licensed use limited to: The University of British Columbia Library. Downloaded on March 03,2024 at 05:25:41 UTC from IEEE Xplore. Restrictions apply.

Now, the Minecraft server thread is running only on CPU 0. This process is repeated for every Minecraft server after startup using subsequent virtual cores. It should be noted that there are other threads involved in the JVM stack, like garbage collection and socket connections, which is still managed automatically. Only the server thread is affected, since we suspect that it creates the most workload.

VII. Collecting Host Workload Data

The monitoring tool used to gather data is *sar* from the *sysstat* package. Our *sar* configuration is set to poll for data every 1 second for 40 seconds. The *sar* output file is in an unreadable object format, so using *sadf*, the formatter, is necessary to create Comma Separated Value records for easy parsing and/or importing into DBMS. An example of output:

```
1. # hostname;interval;timestamp;CPU;%user;
%nice;%system;%iowait;%steal;%idle
2. 00.gameserver.SysCon2017;1;2017-02-09
23:10:19 UTC;0;0.00;0.00;0.00;0.00;0.00;
100.00
3. ...
```

The testing and data collecting process happens each time a server is added, with 25 bots spawned on all servers.

VIII. Hosting Machine Performance Analysis

The goal of this experiment was to collect CPU workload data of server hosts that are running multiple, active Minecraft servers. That is, Minecraft servers with 25 players connected to each of them. Our attempts at controlling the workload generated from these game servers is graphically depicted below.

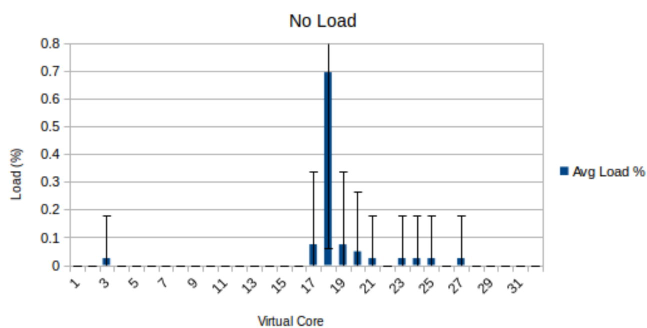

Fig. 2. No Minecraft servers running. It represents our host while it is completely idle (the baseline).

First, we show a baseline graph. It shows workload data of the host with *no Minecraft servers* running. Pay attention to the y-axis scaling, as the bar heights are slightly misleading at first glance.

Fig. 3 clearly demonstrates that our attempt at controlling workload of the Minecraft servers is successful. It shows that our first server, which has its affinity set to *virtual core 0 (represented as 1 in the figure)*, is creating much more workload on that core than the rest of the cores. There are

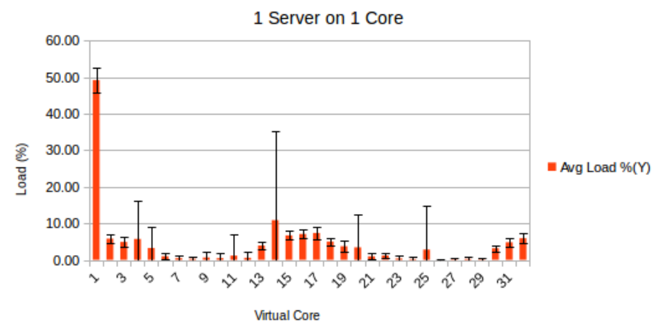

Fig. 3. 1 Minecraft server thread running on virtual core 1.

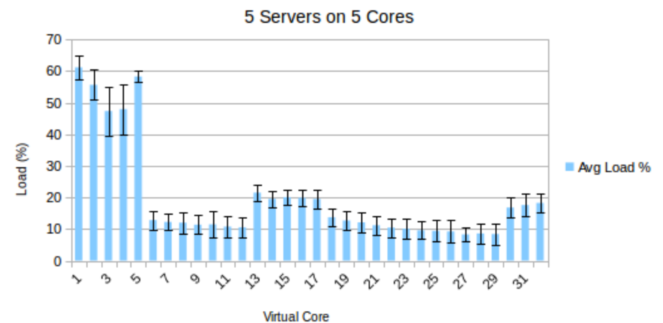

Fig. 4. 5 Minecraft server threads running on separate virtual cores 1 to 5.

interesting sections of the graph from virtual cores 13-20 and 30-32 that starts appearing in this test and subsequent tests. There is uncontrolled workload apparent in these sections.

Halfway through our tests, depicted in Fig. 4, our controlling efforts remain successful. The workload from controlled servers on their virtual cores is much higher than the rest of the cores. Although, the interesting trend discovered within the last graph is now more evident. All of the uncontrolled virtual cores have a much higher load when compared with previous tests. Uncontrolled cores 13-17 and 30-32 still have the most outstanding readings.

Our final test (Fig. 5) shows more of the same. However,

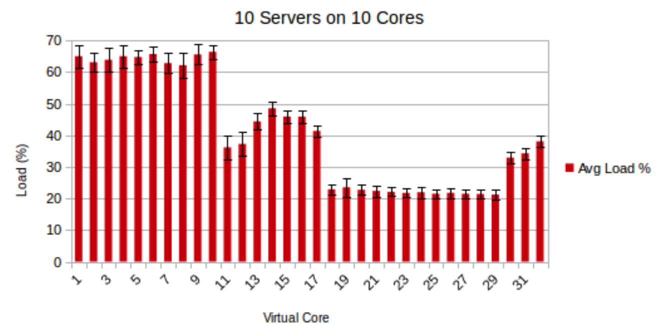

Fig. 5. 10 Minecraft server threads running on separate virtual cores 1 to 10.

Authorized licensed use limited to: The University of British Columbia Library. Downloaded on March 03,2024 at 05:25:41 UTC from IEEE Xplore. Restrictions apply.

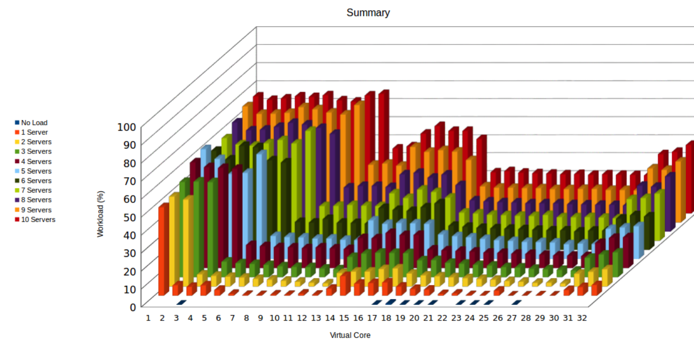

Fig. 6. System workload during each test configuration (10 in total), and the baseline for a point of reference.

uncontrolled cores are now displaying a substantial amount of load, even when compared to controlled cores. The average load of uncontrolled cores is 30.43%, while controlled cores average 64.45%.

A forest graph (Fig. 6) was created to show contrast between all test configurations. The uncontrolled cores display an upward trend of workload as the number of active Minecraft servers increases. The middle and end sections of the cores discussed previously still have unusually high workloads when compared to others of the same nature. However, they too display the same upward trend. These uncontrolled workloads were consistent throughout the polling of CPU data. For example, average standard error of the readings of uncontrolled cores is quite low: 1.91 in Fig. 5 and 3.03 in Fig. 4.

IX. Future Work

Our future work will be related to Minecraft and other game servers performance investigation within Rocket under CoreOS and Docker's Containers.

The game servers can be dynamically deployed based on the workload of actively running host machines. With the use of existing container, sand boxing, and virtualization software, this deployment process is able to be streamlined. Investigation of using Docker containers to achieve that is now a priority. Games with similar technical architecture to Minecraft will be considered for further exploration in automated stress testing. If a game has an openly documented networking protocol, there are prospects for automated stress testing using custom bots.

X. Conclusion

Experiments with the new Mineflayer NodeJS Bot combined with a custom game server deployment procedure yielded not only consistently predictable performance measures, but also demonstrated the effect of adding more than one Minecraft game server on a host machine. The compounding workload of the uncontrolled threads of n concurrent, active Minecraft servers can adversely affect the next deployed on the same host, since workload is evidently not evenly distributed across the virtual cores. Further testing to find performance ceilings needs to be conducted as a result. These limits also need to be taken into consideration when forecasting future performance needs on the fly. Now that we have reliable metrics and a deeper knowledge of the workload that Minecraft servers expose to their hosts, this data can be used for determining the future ampleness of existing infrastructures that host Minecraft servers.

For game server hosting companies of any kind, staying ahead of the curve of player-base growth by intelligently deploying resources can greatly increase player experience and overall satisfaction. In the age of distributed computing and massive server farms, these companies must take advantage of smarter deployment solutions or they risk being left behind.

XI. Acknowledgment

The research project was supported by NSERC's grant CCI ARD Level 1, 465659-14: GPN-Perf: "Investigating performance of game private networks" & CCI ARD 2, 477506-14: GPN-Perf2: "Game private networks and game servers performance optimization" in 2016–2017.

The authors are grateful to the anonymous referees and the anonymous reviewers for their helpful feedbacks which improved the quality of the paper.

References

[1] T. Alstad, J. R. Dunkin, S. Detlor, B. French, H. Caswell, Z. Ouimet, and Y. Khmelevsky., "Game network traffic emulation by a custom bot." in *2015 IEEE International Systems Conference (SysCon 2015) Proceedings*, ser. 2015 IEEE International Systems Conference. IEEE Systems Council., April 13-16 2015.

[2] P. A. Branch, A. L. Cricenti, and G. J. Armitage, "An ARMA (1, 1) prediction model of first person shooter game traffic," in *Multimedia Signal Processing, 2008 IEEE 10th Workshop on*. IEEE, 2008, pp. 736–741.

[3] A. L. Cricenti and P. A. Branch, "A generalised prediction model of first person shooter game traffic," in *Local Computer Networks, 2009. LCN 2009. IEEE 34th Conference on*. IEEE, 2009, pp. 213–216.

[4] Q. Zhou, C. Miller, and V. Bassilious, "First person shooter multiplayer game traffic analysis," in *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, May 2008, pp. 195–200.

[5] W. Doherty and A. Thadhani. (1982) The economic value of rapid response time (IBM Technical Report GE20-0752-0). [Online]. Available: http://www.vm.ibm.com/devpages/jelliott/evrrt.html

[6] Andrewrk. (2016) Prismarinejs/mineflayer-navigate. GitHub. [Online]. Available: https://github.com/PrismarineJS/mineflayer-navigate

Authorized licensed use limited to: The University of British Columbia Library. Downloaded on March 03,2024 at 05:25:41 UTC from IEEE Xplore. Restrictions apply.