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: fmatt.cocar, reneisha.harrisg@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

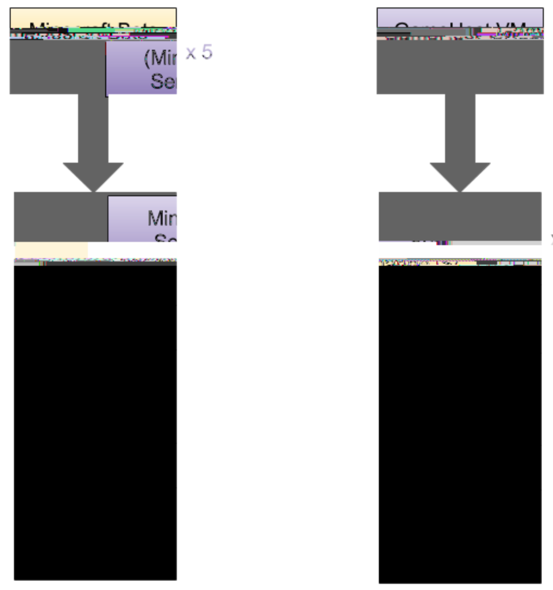

Fig. 1. Current Infrastructure Diagram

to a Minecraft server. The configuration also helps the bots to run more smoothly due to spreading out their workload.

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

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 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

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.

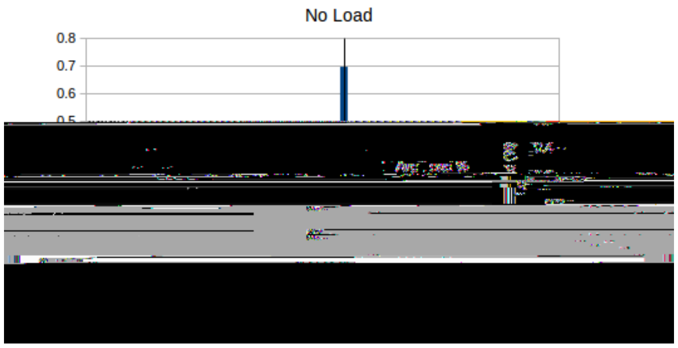

Fig. 2. No Minecraft servers running. It represents our host while it is completely idle (the baseline).

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

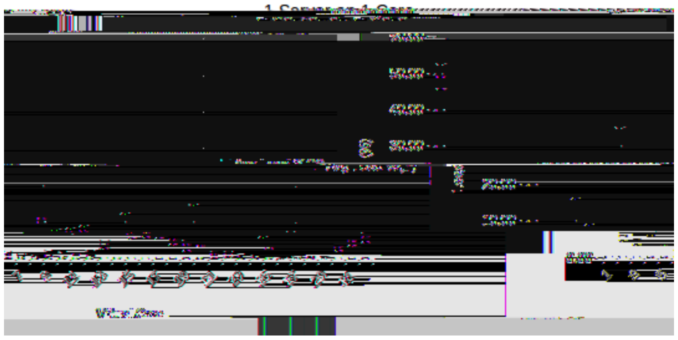

Fig. 3. 1 Minecraft server thread running on virtual core 1.

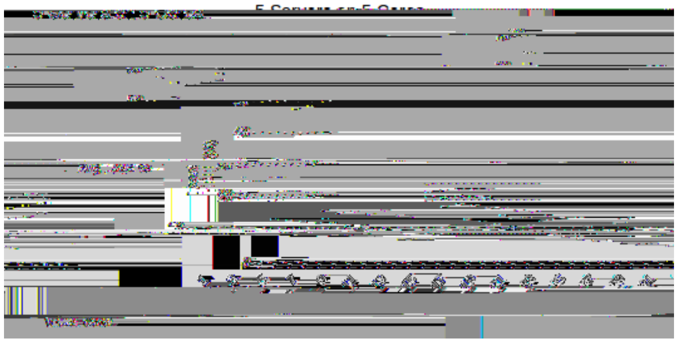

Fig. 4. 5 Minecraft server threads running on separate virtual cores 1 to 5.

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,

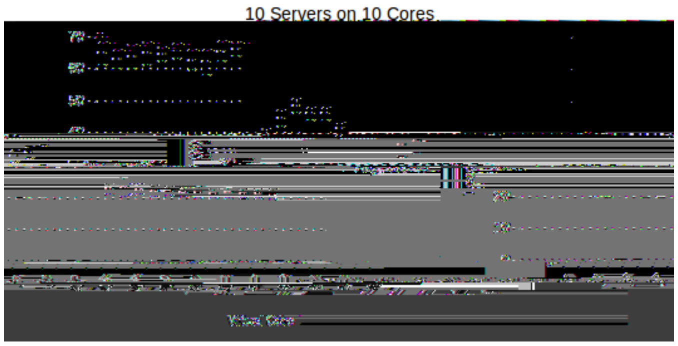

Fig. 5. 10 Minecraft server threads running on separate virtual cores 1 to 10.

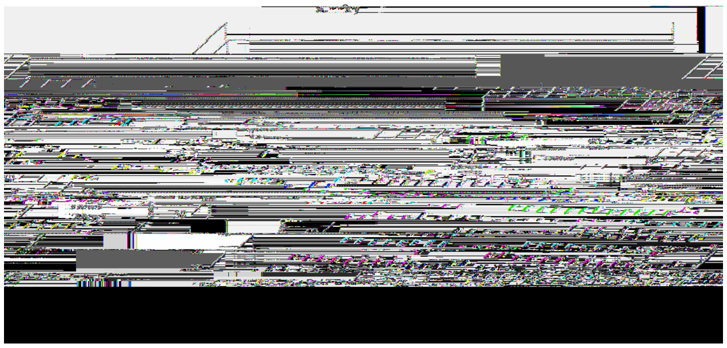

Fig. 6. System workload during each test configuration (10 in total), and the baseline for a point of reference.

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