Yardstick: A Benchmark for Minecraft-like Services

Jerom van der Sar*
AtLarge Research Team
J.vanderSar@atlarge-research.com

Jesse Donkervliet†*
AtLarge Research Team
J.Donkervliet@atlarge-research.com

Alexandru Iosup†*
AtLarge Research Team
A.Iosup@vu.nl

ABSTRACT

Online gaming applications entertain hundreds of millions of daily active players and often feature vastly complex architecture. Among online games, Minecraft-like games simulate unique (e.g., modifiable) environments, are virally popular, and are increasingly provided as a service. However, the performance of Minecraft-like services, and in particular their scalability, is not well understood. Moreover, currently no benchmark exists for Minecraft-like games. Addressing this knowledge gap, in this work we design and use the Yardstick benchmark to analyze the performance of Minecraft-like services. Yardstick is based on an operational model that captures salient characteristics of Minecraft-like services. As input workload, Yardstick captures important features, such as the most-popular maps used within the Minecraft community. Yardstick captures system- and application-level metrics, and derives from them service-level metrics such as frequency of game-updates under scalable workload. We implement Yardstick, and, through real-world experiments in our clusters, we explore the performance and scalability of popular Minecraft-like servers, including the official *vanilla* server, and the community-developed servers *Spigot* and *Glowstone*. Our findings indicate the scalability limits of these servers, that Minecraft-like services are poorly parallelized, and that *Glowstone* is the least viable option among those tested.

KEYWORDS

Yardstick, benchmark, as a service, Minecraft, distributed systems, online gaming.

ACM Reference Format:
Jerom van der Sar, Jesse Donkervliet, and Alexandru Iosup. 2019. Yardstick: A Benchmark for Minecraft-like Services. In *Tenth ACM/SPEC International Conference on Performance Engineering (ICPE '19), April 7–11, 2019, Mumbai, India.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3297663.3310307

1 INTRODUCTION

Primarily operated on high-performance machines in cluster-based data centers, online computer gaming is a billion-player, multi-billion-dollar industry [14]. The growth potential of this industry depends not only on innovative content, but also on transitioning existing and future games to scalable online services. By offering seamless but *modifiable* virtual environments, the online game Minecraft has become the sixth most-popular game on PC platforms, and is popular also on other platforms.[1] However, the transition of Minecraft to full delivery "as a service" is possibly still hampered by technology. Because Minecraft acts as a distributed simulation and database with high-frequency updates, it is not trivial to understand its performance and scalability. Relatively few performance studies focus on Minecraft-like services [7, 8, 5], and no public benchmarks for Minecraft-like games exist. Addressing this knowledge and tools gap, in this work we design, implement, and use Yardstick, a benchmark for Minecraft-like services.

Online games are now a popular and large market, of over 250 million active users and over $100 billion global revenue [14]. Developed by Mojang AB and acquired by Microsoft in 2014 for $2.5 billion,[2] Minecraft has currently more than 70 million active users and generates over 100 million dollars in sales yearly.[3] The success of Minecraft has triggered game-developers to create tens of Minecraft-like games, many of which are currently distributed through the Steam market for PC-games and, combined, have millions of customers. The modifiable environments of Minecraft are used not only by gamers, but also as environments for education[4] (e.g., in history, anatomy, digital logic, and economics), and have been used for activist approaches to save real-world ecosystems.[5]

To maintain its status in the very competitive market, Minecraft relies not on innovative content, but on technological advances: it operates already across more than 10 gaming platforms, it coordinates the servers operated by community-members, and is offered "as a service" through Minecraft Realms. Technology-related, Minecraft fosters an active and sizable modding community, which has already created over 38,000 customized modules to alter the gameplay of Minecraft.[6] Modding is not supported by the official (*vanilla*) Minecraft distribution provided by Mojang; hence, the modding communities have created several other implementations of the Minecraft server-protocol to support their customizations: the mod-friendly servers include *Spigot* [12], Sponge, and *Glowstone* [16].

Hosting *vanilla* Minecraft servers and various kinds of modded servers is currently an emerging but fast-growing service. Despite the large interest in Minecraft server hosting, the performance of Minecraft-like services, and in particular their scalability, remains poorly understood. Moreover, there currently exists no public benchmark for Minecraft-like services. Therefore, it is challenging

*Delft University of Technology, the Netherlands
†Vrije Universiteit Amsterdam, the Netherlands

ICPE '19, April 7–11, 2019, Mumbai, India

ACM ISBN 978-1-4503-6239-9/19/04…$15.00
https://doi.org/10.1145/3297663.3310307

[1] http://bit.ly/PopularPCGames
[2] http://bit.ly/Minecraft2-5B
[3] http://bit.ly/MostPlayedGames, bit.ly/MojangSales, bit.ly/MicrosoftMC
[4] https://education.minecraft.net/
[5] For example, in Poland, http://bit.ly/MinecraftToSaveForests
[6] https://www.curseforge.com/

to determine, and consequently to improve, the performance of Minecraft-like services. Toward understanding the (relative) performance of Minecraft and Minecraft-like services, in this work we propose Yardstick, a benchmark for Minecraft-like services. Our main contribution is threefold:

(1) We design a system model for the operation of Minecraft-like services (Section 2). Our model provides a reference architecture for Minecraft-like games, which captures the fundamental characteristic of such games: modifiable terrain. The system model also captures the performance of the core Minecraft game-loop. We validate the model with real-world Minecraft and Minecraft-like servers.
(2) We design the Yardstick benchmark for Minecraft-like services (Section 3). Yardstick fulfills a diverse set of requirements, among which Yardstick generates a workload that simulates lifelike player behavior in representative virtual-environments. Moreover, Yardstick defines and monitors various system-, application-, and service-level metrics.
(3) We conduct, through real-world experimentation, an analysis of Minecraft-like services (Section 4). We construct an experiment setup that focuses on three popular server-distributions, including *vanilla* Minecraft, use the realistic workloads derived from real-world communities, and conduct comprehensive parameter exploration to understand the scalability of these services.

This work aligns with our long-term vision, of Massivizing Computer Systems [10]. It aligns with the principle of increased awareness about the emerging properties of ecosystems (P9), by informing ourselves and the community about the evolution of these new computer ecosystems and, more specifically, how we address the need for increased performance among such ecosystems. We specifically tackle in this work the challenge of understanding this emerging ecosystem (challenge C19, "understanding the New World"). This work also aligns well with use case 6.3: Online Gaming. In particular, Minecraft-like games entertains tens of millions of players worldwide at a fraction of the cost.

2 SYSTEM MODEL

In this section, we model the operation of Minecraft-like services. We propose a system model, define the core unit of computation in Minecraft services (the game-loop, or *tick*), and define performance and service metrics for Minecraft-like services. Last, we validate the system model with three Minecraft-like servers, which we benchmark in Section 4.

2.1 Overview

Minecraft is a multi-player gaming system designed around the traditional client-server model. The server performs the majority of computation. All Minecraft users install locally client-software, which connects to the server. Users may download server-software, either *vanilla* or a server produced and modified by the community, and use it to host Minecraft services independently. Alternatively, users may use Minecraft servers as a service; for example, Mojang offers Realms, that is, servers hosted for a monthly fee.

Figure 1 depicts the system model of Minecraft-like servers. The typical Minecraft-like server adds to the basic client-server model a variety of components, some idiosyncratic (e.g., related to the modifiable virtual-world of Minecraft, or to the Minecraft client-server communication protocol), others generic (e.g., for user authentication, or for data storage). This system model is a generalization of Minecraft(-like) servers commonly used by the community. We describe the six main components below.

2.1.1 Network Manager. The user connects to Minecraft-servers with client-software, which connects to the *Network manager*. The latter accepts the incoming TCP-connection, configures the necessary protocol stack, including layers for encryption and message encoding/decoding, and exposes the messages to the *Game loop* through a message queue.

2.1.2 Authentication Agent and Server. All Minecraft servers support authorization of Minecraft players. In particular, servers use an *Authentication Agent* to communicate with a (developer-controlled) *Authentication Server*.

2.1.3 Server Configuration. This component manages the parameters that control the virtual-world simulation, e.g., the frequency of spawning certain entities in the virtual world, the frequency of persisting the world-state, the maximum message size, and the view distance[7].

2.1.4 One or Several Virtual Worlds. A server may contain several worlds, each containing zero or more players. Players interact with the virtual world, and with other players connected to the same virtual world, through the Minecraft-service (see game loop, Section *2.1.5*). Each virtual world consists of game objects. These objects are either (1) *terrain*, which is a grid of immovable blocks, or (2) *entities*, some player-controlled, which may move freely.

The terrain is segmented into *chunks*, which are discrete cuboids[8] of $l \times m \times n$ *blocks*, with $l = 16$, $m = 16$, and $n = 256$ for the official Minecraft distribution.[9] To give the illusion that the terrain of the world is infinitely large, terrain-chunks are generated using a pseudo-random number generator and a complex algorithm that mimics various geographies and biomes. At runtime, chunks are generated, loaded, and stored, individually, as players join and leave areas of the terrain. The *World loader* component determines which chunks should be loaded and stored to the world database, and ensures the world is persisted between user-sessions.

Entities are all non-block objects in the game, including players, creatures, and certain physics-aware block-like objects such as sand and gravel[7]. Some entities, such as cows and wolves, are controlled by virtual agents that respond to other entities and to the terrain. For example, cows may move toward patches of grass, and wolves may move towards players. When chunks are unloaded, all entities located within are persisted, along with their corresponding metadata (such as health).

2.1.5 The Main Service: the Game Loop. The primary service offered by a Minecraft-server is to operate the game loop for all connected players, independently for each virtual world managed by the server. We model the game loop in Section 2.2. Similarly to the typical game-loop in other games [22], the Minecraft game-loop

[7] Community-led Minecraft wiki, https://minecraft.gamepedia.com
[8] A parallelepiped whose faces are all all rectangles, e.g., as described by http://mathworld.wolfram.com/Cuboid.html

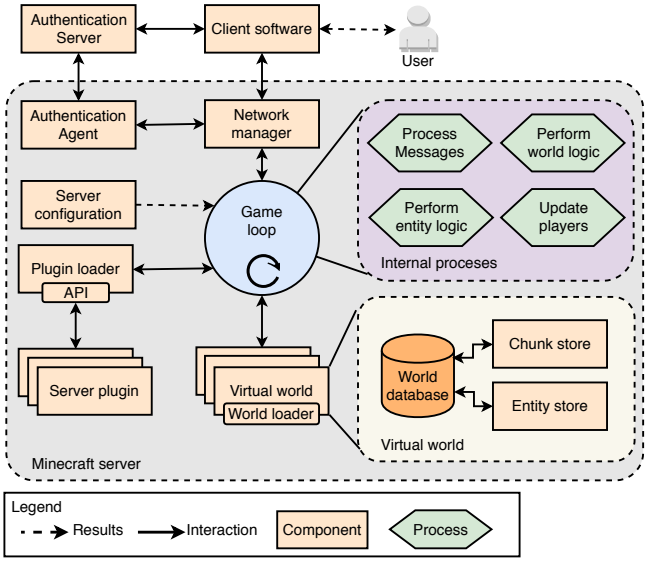


Figure 1: System model for Minecraft-like servers.

performs the majority of the computation in Minecraft servers, and, in particular, four categories of tasks[9]: (1) processing of incoming messages; (2) performing updates regarding the virtual worlds; (3) performing updates regarding the entities (including players) in the worlds; (4) updating players of the new world and entity states.

2.1.6 Server-side Plugins. Minecraft has a considerable community that creates new content, altering the *vanilla* game. Server-side *plugins* can alter the operation of the server, and expose this functionality through APIs. For example, a plugin may allow players to generate complex constructions on the map, without the effort required to manipulate each block; another may generate a map of the region and post it to an online forum outside the game, to give the entire community a 3d-depiction of progress in the collaborative building process. The *Plugin loader* and *Server plugin* components appear only in the systems designed and operated by the community, such as *Spigot* and *Glowstone* [12, 16], but not in the *vanilla* Minecraft-servers. At server startup, the plugin loader dynamically loads the plugins indicated by the server-operator; dynamically, through the API, the server-operator can run the functions offered by plugin as a service.

2.2 The Minecraft Game Loop

Minecraft services perform most computation in the game loop (*tick*). Ticks are started with constant frequency (*tick frequency*) of 20Hz; hence, the interval between the start of consecutive ticks (*tick interval*) is equal 50ms. Processing a tick completes after a variable amount of time (*tick duration*), which depends on the amount of tasks the game-loop has to complete, the amount of computation for each task, and the overhead of the game-loop implementation; for example, many entities converging in the same chunk can lead to substantial amounts of computation, relatively to a chunk with few entities. The overhead is not fixed; although the main operations that occur within a tick are known, the order in which these

[9]Community-led Minecraft wiki, http://technical-minecraft.wikia.com

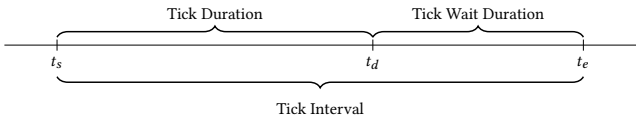


Figure 2: Time line of the Minecraft game loop.

take place and the level of concurrency used depends on the implementation. If the game-loop completes its computation before the tick interval has elapsed, it pauses until the next tick should be performed (the pause is for a variable *tick wait duration*).

Figure 2 illustrates a game tick: t_s indicates the start time of a tick, t_d the start of the tick wait period, and t_e the end of the tick (and the start of the next tick).

2.2.1 Relative Utilization. From the game loop, we derive a new metric, the *relative utilization*, which is for game-servers the analogue of resource utilization for traditional servers. We define the relative utilization as the fraction of the tick duration over the tick interval (Definition 2.1). Under normal operating conditions, $0 \leq U_r \leq 1$. When U_r approaches 1, the tick wait duration is low and there is little leeway for additional server load. When U_r approaches zero, little to none of the capacity of the server is used. This metric effectively allows us to determine when spare capacity remains, and when a server is overloaded ($U_r > 1$, discussed next).

Definition 2.1. The relative utilization of a server is defined as:

$$U_r = \frac{t_d - t_s}{t_e - t_s}$$

where U_r is the relative utilization, and the variables t_d, t_s, and t_e are interpreted as in Figure 2.

2.2.2 Server Overload. If, for a given tick, the relative utilization is larger than 1, the server cannot process the next tick at the correct time and must delay the start of the next tick until spare capacity becomes again available. This behavior results in desynchronized updates and can cause unwanted game-behavior such as loss of interactivity; in turn, these effects can quickly sour the gameplay experience [4, 21]. Definition 2.2 captures this situation.

Definition 2.2. A Minecraft tick has the *overloaded property* iff., for that tick, it holds that $t_d - t_s > t_e - t_s$.

Defining server-overloads that cause the degradation of gameplay experience is not straightforward. A server that occasionally experiences an overloaded tick will not reduce the user experience significantly, because the server has the opportunity to compensate for the delay in the ticks that follow, and because the client-software can use simple prediction [15, 24] and more complex techniques [3, 20] to compensate for mismatched server-updates. Only when overloaded ticks occur in bursts over a short timespan it becomes likely for the players to notice undesirable behavior.

Our definition of system overload (Definition 2.3) is based on a sliding-window average (τ). The length of the window is arbitrary, because no formal definition of system overload currently

exists for Minecraft-like services. A short window length may cause false positives. A long window length may fail to identify an overloaded server.

Definition 2.3. A Minecraft server is *overloaded* if more than half of the ticks in the previous τ seconds have the overloaded property. (In this work, we use $\tau = 5s$.)

2.3 Validating the Model

We validate here the (abstract) system model introduced in Section 2.1. To this end, we select popular real-world Minecraft-like servers, and map them to the system model. We focus on each component of the system model, in turn. Overall, we do not find components that are not used by at least one Minecraft-like server, and find different implementations for specific components; we conclude that our system model is a reference architecture for Minecraft-like servers.

Although the *vanilla* Minecraft server does not publish its source code, many community projects try to augment its functionality or recreate its functionality from scratch. We select among these projects two popular Minecraft-like servers, *Spigot* and *Glowstone*. *Spigot* [12] is an adaptation of *CraftBukkit*[10]: it is fully compatible with *vanilla* Minecraft, and also incorporates a plugin system and several performance-oriented extensions. *Glowstone* is an independent re-creation of Minecraft.

2.3.1 Network Manager. *Spigot*, *Glowstone*, and *vanilla* all use Netty[11] for network management. Internally, Netty uses worker threads and asynchronous event groups to accept and configure communication channels.

2.3.2 Authentication Agent and Server. The authentication approach used by *vanilla* Minecraft, Yggdrasil, is based on access-tokens, which the client software requests and the Minecraft server validates.[12] The other servers use the same or a similar authentication approach. For example, *Spigot*'s use of Netty means that, during the authentication phase of the connection, the *Network manager* simultaneously spawns a thread, the *Authentication agent*, which verifies the session key with Mojang-controlled servers.

2.3.3 Server Configuration. The *vanilla* Minecraft server, *Spigot*, and *Glowstone* have separately-stored configurations. The configurations are each loaded on startup and persist for the duration of the process.

2.3.4 Virtual World. The world database uses a proprietary format, Anvil[13] to store *regions*, which are comprised of 32×32 chunks. This format is based on a community-developed approach and has been included in *vanilla* Minecraft since version 1.3. The same format is used in *Spigot*, which further exposes the state of the virtual-world through the Bukkit API, and uses for performance reasons an IO thread-pool to load and store multiple world-chunks concurrently, based on the location of each player. *Glowstone* supports the Anvil file format, with minor incompatibilities.

For procedural world-generation, *vanilla* Minecraft and *Spigot* use the same algorithm, different from *Glowstone*'s.[14]

2.3.5 Plugin Loader and Server Plugins. The *vanilla* Minecraft server does not support plugins. *Spigot* and *Glowstone* expose the Bukkit API (shared by all servers based on *CraftBukkit*). Developers create JVM-compatible binaries that are loaded dynamically through JVM class-loaders. However, the class-loaders are configured hierarchically and thus plugins may access public APIs in other plugins. The Bukkit API is event-driven; to alter real-time in-game behavior, plugins register to it their event-handlers. Additionally, Bukkit features synchronous and asynchronous task schedulers, which enable multi-threaded execution.

2.3.6 Game Loop. All Minecraft servers perform a variety of tasks during the game loop, including processing all player messages, lighting updates, and updating the weather.[15] *Spigot* and *Glowstone* both implement a variant of the Bukkit API. For each task in the game loop, they fire the corresponding API events so that plugins may alter the game behavior. Bukkit is not thread-safe, which means plugins must resynchronize with the game loop after asynchronous computation.

3 DESIGN OF YARDSTICK, A BENCHMARK FOR MINECRAFT-LIKE SERVICES

In this section, we present the design of Yardstick, a benchmarking suite for Minecraft-like services. We define a set of eight requirements, and present the high-level design of Yardstick and the details that help fulfill these requirements.

3.1 Requirements

We consider for our benchmark both requirements applicable for all types benchmarks [23] and domain-specific requirements. We discuss both classes of requirements, in turn.

3.1.1 Requirements Applicable to Many Types of Benchmarks. We adopt here five common criteria [23]:

R1 *Fairness:* The benchmark should provide a fair performance assessment for compatible systems. In particular, benchmark developer should limit bias to one specific system as much as possible.

R2 *Ease of use:* The benchmark should easy to set up, configure, and use so that obtaining results for new applicable systems is simple.

R3 *Clarity:* The benchmark should choose and present results in a manner that logically characterizes performance.

R4 *Representativeness:* The benchmark should be comprised of tasks and metrics suitable to the target systems.

R5 *Portability:* should run on different host-platforms.

3.1.2 Benchmarking Minecraft-like Services.

R6 *Relevant and Realistic Workloads and Metrics:* Players emulated by the benchmark should imitate true player behavior such that the workload created by the emulated players is similar to real-world workloads. Similarly, the virtual world

[10] *CraftBukkit* is now closed, https://bukkit.org/
[11] https://netty.io/
[12] Minecraft Coalition, http://wiki.vg.
[13] https://minecraft.gamepedia.com/Anvil_file_format.
[14] http://bit.ly/GlowstoneMinecraft.
[15] http://technical-minecraft.wikia.com/wiki/Tick.

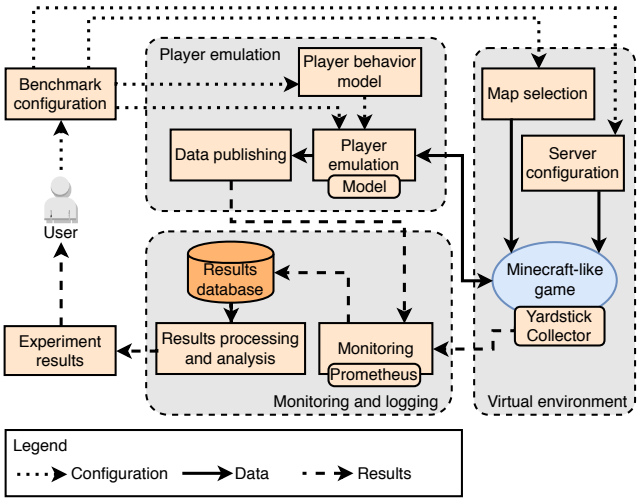


Figure 3: Architectural Overview of the Yardstick Benchmark.

of the benchmark should resemble popular commercial or community-driven worlds. The metrics used by the benchmark to describe system performance and scalability should provide insight into how real players experience these properties.

R7 *Reconfigurability:* The benchmark should be adaptable to various Minecraft-like services. If a new type of Minecraft-like service arises, the benchmark should require few alterations to support the new service. Likewise, the benchmark should be reconfigurable such that presenting results in another form is possible.

R8 *Low Overhead:* The benchmark needs to interact with the Minecraft-like service to collect application-level metrics. The overhead of this interaction needs to be low.

3.2 Design Overview

All benchmarks, and so also Yardstick, define [9, Ch.1]: (1) a process to conduct the benchmark, including how a single benchmarking experiment should run, and how and what to monitor about the system under test (described in this section, with parameters detailed in Section 3.3), (2) the workload given as input to the system under test (Sections 3.4 and 3.5), (3) the metrics to assess the output of the system under test (Section 3.6).

3.2.1 The Benchmarking Process. The overall process of Yardstick addresses in particular the requirements **R1**, **R3**, **R4**, **R6**, and **R7** (see Section 3.1). Yardstick evaluates the performance of the server using specific configurations of these parameters, grouped into *experiments* (**R1**).

Yardstick subjects a Minecraft server to workloads determined by the virtual world and a set of emulated players. Both are parameterizable, leading to infinitely many possible workloads. A Minecraft service can be configured to generate procedurally its virtual world or to use an existing pre-generated world; the latter option allows different Minecraft-like servers to run the same workload. The workload generated by emulated players is specified by selecting a built-in player behavior (**R4**). The resulting workloads are independent of the tested Minecraft-server (**R1**) and selected to cover a diverse set of operational conditions (**R7**).

Yardstick monitors both the machine running the Minecraft server application and the machines that run the emulated players. After an experiment is complete, both raw and processed data are made available to the Yardstick user to analyze server performance (**R1**,**R3**). During an experiment, Yardstick measures both system-level metrics such as CPU, RAM, and network usage, and application-level metrics such as tick duration, number of transmitted messages, and the relative utilization of the server (**R4**).

3.2.2 The Benchmark Architecture. Figure 3 depicts the architecture of Yardstick. Yardstick consists of three main components (the shaded boxes in the figure). The *Virtual environment* component includes the Minecraft service, that is, server and APIs, alongside the current configuration and the virtual-world, plus the Yardstick Collector component that monitors the activity of the service.

The *Player emulation* component connects emulated players to the system under test, that is, players are emulated by an AI-algorithm, according to a behavioral model specified by the user (**R7**, see Section 3.5).

The *Monitoring and logging* component monitors both the emulated players and the virtual environment, processes the results that are communicated to the user, and logs these to the *Results database*.

3.2.3 The Benchmark Implementation. The implementation of Yardstick addresses in particular the requirements **R2**, **R5**, and **R8** (see Section 3.1). Yardstick uses for the Monitoring and logging component the Prometheus monitoring tool,[16] a popular monitoring tool for real-time measurement of time-series data. Prometheus is portable (**R5**) and low-overhead (**R8**), and could be replaced by similar monitoring systems, e.g., Ganglia and Nagios. We have deployed Yardstick in DAS-5 [2] (see Section 4.1.2), which is a multi-cluster infrastructure for computer science that shares common features available in modern multi-cluster datacenters (relevant here, x86-based CPUs, various types of fast memory and storage devices, and Ethernet and Infiniband high-speed networks).

Yardstick collects server performance metrics through the *Yardstick Collector* component. Collectors serve two key key goals: measure tick-related data, and publish this data to the Yardstick monitoring subsystem. In general, collectors are designed such that they only expend low computational effort (**R8**) and most data processing occurs in the monitoring component. To achieve low overhead, Yardstick also uses simple counters to measure application level metrics on the system under test, and only communicates these counter values to the monitoring subsystem. Only after the experiment is complete, data is further aggregated and processed.

The implementation of the *Yardstick Collector* component is specific to the system under test. Currently, Yardstick includes a collector for each of *vanilla*, *Spigot*, and *Glowstone*. These collectors consist of minimal modifications to the server source-code, to collect information with little implementation effort (**R2**), but it may also be possible to use external tools, such as drop-in Java agents.

[16] https://prometheus.io/

Table 1: Overview of supported Yardstick experiments. Emphasized values indicate constants in other experiments. Acronyms: W—Workload, C—Configuration.

ID	Type	Parameter Description	Values
1	W	Input world complexity	High, *Medium*, Low
2	W	Connected players, count	25, *50*, 75, …, 250, 275, 300
3	W	Player behavior model	*SimpleWalk*, WalkModify
4	W	Join strategy	Linear join, *Fixed*
5	C	View distance	*10*, 14, 18, 22, 26, 30, 32
6	C	Entity spawning	*true*, false

3.3 Overall Process: Configuration Parameters

Yardstick currently supports five main configuration parameters that guide how the benchmark operates. For each parameter, Yardstick pre-defines several distinct, relevant, and realistic values, which are used to assess the impact of the parameter on the performance and scalability of the system under test, whilst keeping other parameters constant.

Table 1 summarizes the parameters and their pre-defined values. For each parameter, one value is emphasized. This value indicates the default setting for the parameter. For example, for the experiment with input-world complexity set to low, we test with *50* players and entity-spawning set to *true*.

The six main parameters supported by Yardstick are:

(1) *Input World Complexity:* This parameter defines the complexity of the input virtual world. The complexity levels correspond to the virtual worlds listed in Table 2.
(2) *Number of Connected Players:* This parameter defines the number of connected bots to the server. In conjunction with the join strategy, this number specifies how many bots are on the server at any given time.
(3) *Player Behavior Model:* The player behavior model specifies how the simulated players interact with the environment. In this experiment, we test two player behavior models, these are further described in Section 3.5.
(4) *Join Strategy:* The join strategy defines how bots join the server throughout the experiment. Yardstick supports two join strategies: 1) *Linear join:* 10 players join the server every 120 seconds. 2) *Fixed:* 5 players join the server every second until a multiple of 25 is reached. The server keeps running for one hour, after which it is restarted and the process repeats for the next multiple of 25 players.
(5) *View Distance:* This indicates the number of chunks sent to each player, measured as a radius in chunks around the player. By default, the Minecraft server imposes a value-range between 2 (low load, low visibility) and 32 (high load, excellent visibility). The default value is set to 10, which is commonly used in practice.
(6) *Entity Spawning:* The entity spawning parameter is a combination of three application settings: *spawn-animals*, *spawn-npcs*, and *spawn-animals*. When each is set to *true*, the Minecraft server will spawn game entities of various types that move around in the world. For each of these entities, the server computes walk paths, and player interactions. Thus, this parameter may impose an additional computational cost in the game loop.

Table 2: Virtual worlds used by Yardstick.

Name (official name of community map)	Size [MB]	Downloads (in 1,000s)	Complexity (relative)
Vertoak City	62	556	High
World of Worlds	53	244	Medium
Castle Lividus of Aeritus	36	307	Low

3.4 Input Workload: Virtual World

The Minecraft game always starts from a virtual-world map, which is either generated procedurally, or generated by other players (possibly, from a previously generated world). Virtual worlds may have varying size and processing complexity, based on whether the terrain is walkable, on the material types used, on meta-data associated with certain materials, and on the size of the world. Moreover, the complexity of virtual worlds impacts the performance of Minecraft-like games [1]. Thus, Yardstick must define the starting virtual-world.

Yardstick natively supports virtual-worlds generated procedurally starting from the same seed of the pseudo-random number generator; sharing the seeds is sufficient to ensure the same virtual-worlds are generated (**R5**). However, to ensure the relevance of the starting virtual-world (**R6**), Yardstick uses publicly available virtual-worlds generated by the community. The three virtual-worlds, whose characteristics are summarized in Table 2, have different complexity and size, and are some of the most popular virtual-worlds in the community (as indicated by their download counts).

3.5 Input Workload: Emulation Models for Player Behavior

Any benchmark for a class of interactive systems must define the interaction model predicting how clients use the system. Currently, no such interaction-model exists for Minecraft, although many exist for games with different characteristics [11, 21, 19]. Yardstick employs two interaction-models based on an existing model for Second Life [11], which is the online game or environment closest to Minecraft that has a public interaction-model.

Yardstick's interaction-models predict player behavior in-game, and are based on the observation that players either perform a complex activity in a narrow area, for example, building a house, or are moving towards such an area, for example, exploring the world. The Yardstick models combine these activities, subject to the parameters summarized in Table 3. The *SimpleWalk* model lets players walk to close-distance or long-distance locations, and idle momentarily. The *WalkModify* model generates short- and long distance traveling, and also regularly lets the players place and break blocks in the virtual world.

Table 3: Player behavior models supported by Yardstick.

Parameter	Value
SimpleWalk Model	
α: Ratio of long-distance targets to short-distance targets	$\alpha = 1/9$
WalkModify Model	
α: Ratio of long-distance targets to short-distance targets	$\alpha = 1/9$
β: Ratio of break/place tasks to walk tasks	$\beta = 1/3$
γ: Amount of blocks to break/place	$\gamma = 3$

3.6 Metrics for Minecraft as a Service

During an experiment, Yardstick measures a variety of metrics, both system- and application-level. Yardstick also derives service-level metrics, such as Relative Utilization (see Section 2.2.1). Table 4 summarizes all metrics used in this work.

3.6.1 Collection of System-Level Metrics. During experimentation, each machine executes an instance of the monitoring sensor (here, the Prometheus metrics-server), which automatically collects and stores system-level metrics such as CPU load, RAM usage, disk usage, and network usage. The monitoring system is hierarchical, with the root (primary) configured to periodically poll all other monitoring-sensor instances. Data samples are stored in-memory, or on disk in a custom binary format[17], until the end of the experiment. Using a query language exposed through the CLI, the Yardstick tools extract and process the data after the experiment is complete.

3.6.2 Collection of Application-Level Metrics. Yardstick captures several application-level metrics: the number of connected players, number of disconnects, and details of network messages are captured by the *Player emulation* component (see Figure 3); and the tick-length and stage are captured by the *Yardstick collector* component. These components publish data periodically to a fast intermediary cache; Yardstick currently uses Prometheus Pushgateway.[18] The *Player emulation* component also captures application-level messages transmitted between bots and the Minecraft-like server, including the time of transmission, and message type, size, source, and destination. To keep the network usage low (**R8**), these detailed records are not published to the intermediary cache, but stored on-disk in compressed (GZIP) CSV format.

3.6.3 Derivation of Service-Level Metrics. Yardstick derives several service-level metrics from the system-level and application-level metrics it collects. Firstly, from the message data, Yardstick determines the relative frequency, average packet size, rate of each message type, and various basic statistics (e.g., the quartiles). Message-sizes collected by Yardstick represent the application-level data, and not the total size of IP packets; thus, they are irrespective of the underlying transport-protocol. From the game-loop data, Yardstick derives the tick frequency, which should be approximately equal to 20Hz for a Minecraft service that is not overloaded. Last, Yardstick computes the relative utilization (see Section 2.2.1).

[17] https://prometheus.io/docs/prometheus/latest/storage/
[18] https://prometheus.io/docs/instrumenting/pushing/

Table 4: Overview of performance metrics obtained by Yardstick. Acronyms: S—System, A—Application, D–Derived, Y—Yardstick, C—Collector, Tp—Type, Src—Source.

Name	Tp	Src	Description
RAM usage	S	S	RAM usage of the service
CPU load	S	S	CPU load total and per-core
Disk usage	S	S	Rate of disk R/W, in bytes
Network usage	S	S	Rate of incoming/outgoing bytes
Player count	A	Y	Current connected players, count
Disconnects	A	Y	Server-wide disconnects, count
Messages, I/O	A	Y	Timestamped message-log
Tick length	A	C	Time spent processing each tick
Tick stage	A	C	Time spent and order of processing
Messages Frequency	D	Y	Number of messages per second
Message Freq., %	D	Y	Relative frequency, by msg. type
Message Weight, %	D	Y	Fraction of bytes, by msg. type
Tick frequency	D	C	Frequency of game loop updates
Relative Utilization	D	C	See Section 2.2.1

4 EXPERIMENT RESULTS

This section discusses the experiment results obtained from using the Yardstick benchmark on *vanilla*, *Spigot*, and *Glowstone*. The main findings are based on the scalability experiment using both join strategies (ID = 2 and ID = 4 respectively in Table 1); the results of all other experiments are available in a technical report on arxiv.org [18]. Our main findings are:

MF1 Minecraft-like services scale to hundreds of players. For higher numbers of players, the Minecraft services become overloaded.

MF2 Minecraft-like services are poorly parallelized. Although these services try to exploit parallelism, none of the Minecraft services ever fully utilizes the CPU during the experiments.

MF3 Minecraft-like services transmit a vast amount data which increases linearly with the number of players.

MF4 Position updates are the most frequently sent type of updates. The majority of the data sent by the server concerns the terrain of the world.

MF5 Different Minecraft-like servers have different performance profiles. *Glowstone* delivers the worst performance among the tested services. The *vanilla* server performs best.

4.1 Experiment Setup

In this section, we present the experiment setup used throughout our experiments. Using Yardstick, we assess and compare in each experiment the performance of the three systems under test, running independently in the same environment.

4.1.1 System Under Test. This experiment tests several popular Minecraft server implementations. In particular, we investigate the default *vanilla* Minecraft server provided by Mojang, the modded variant *Spigot*, and the open-source reimplementation of the Minecraft server *Glowstone*.

Table 5: Minecraft-like servers used in our experiments.

System	License	Version	Release date	URL
Minecraft	Commercial	1.11.2	Dec 21, 2016	[13]
Spigot	Community	3fb9445eca	Apr 30, 2017	[12]
Glowstone	Community	2017.6.0-94e2efd	Jun 9, 2017	[16]

Spigot is currently the most popular Minecraft server distribution[19] and was forked from *CraftBukkit*, which was previously the most popular variant. Our experiments only use installations of the Minecraft servers, with the customized Yardstick collector included.

4.1.2 Environment. All our experiments use the DAS-5 multi-cluster system [2]. We reserve 7 compute nodes for each sub experiment. One node downloads and executes the Prometheus server and push gateway. This node will collect metrics from the other reserved nodes. This node also uses the Yardstick benchmark tools to measure and obtain the relative utilization per-tick. Yardstick pushes data to the Prometheus push-gateway, which caches metrics temporarily. The Prometheus server collects and stores this data at a constant interval.

One other node is designated to be the server node. This node obtains the Minecraft server software modified with the integrated Yardstick metrics hook.

The remaining 5 nodes are used by the Yardstick framework as client nodes. Each node is loaded with the Yardstick software to connect to the Minecraft server and emulate player behavior. Client nodes also push metrics to the Prometheus push gateway.

4.2 Minecraft-like Services Only Scale to Hundreds of Players

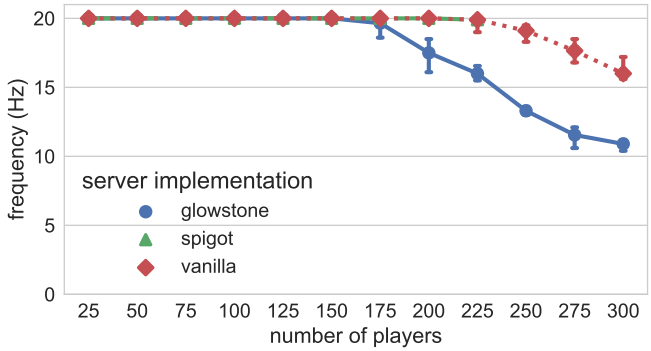
Modern MMO games, such as World of Warcraft and Runescape, scale to thousands of players per service-instance. Can Minecraft-like services scale similarly?

We conduct experiments running workloads with increasing players ("linear join" in Section 3.3) and fixed players ("fixed"), in turn. Figure 4 depicts the tick frequency of the Minecraft-like services *vanilla*, *Spigot*, and *Glowstone*. A decrease in tick frequency causes the simulation of the virtual world to slow down, decreasing the overall game speed, and thus introducing update latency (see Section 2.2).

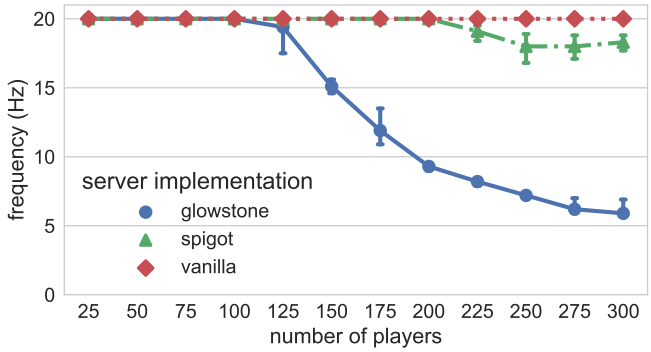
Figure 4a indicates the tick frequency from both *Glowstone* and *vanilla* drops below 20Hz during the increasing players workload. The tick frequency of *Glowstone* drops below 20Hz when connecting 175 players or more, and the tick frequency of *vanilla* drops below 20Hz when connecting 225 players or more. The tick frequency from *Spigot* does not drop below 20Hz, but *Spigot* does not successfully connect more than 225 players (it crashes). Figure 4b shows lack of scalability for *Glowstone* and *Spigot*, but not for *vanilla*. *Glowstone* drops below 20Hz when connecting 125 players or more, and *Spigot* drops below 20Hz when connecting 225 players or more.

Which services drop their tick frequency below 20Hz depends on the workload: for 150 players, the tick frequency of *Glowstone* is approximately 15Hz with the fixed players workload, but for the increasing players workload the frequency is still 20Hz. Similarly, for 275 players, the tick frequency of *vanilla* is approximately 18Hz,

[19] Statistics available at https://bstats.org and http://mcstats.org/global/. (The latter site appeared to be down around February 8, 2019.)



(a) Effect of the increasing players workload. Horizontal axis shows number of connected players as measured by Yardstick.



(b) Effect of the fixed players workload. Horizontal axis shows the number of players that Yardstick tries to connect.

Figure 4: Tick frequency of Minecraft-like game servers when varying the number of players. (Horizontal axis shows the number of players that Yardstick tries to connect. Markers indicate the median value. Whiskers indicate a 95% confidence interval.)

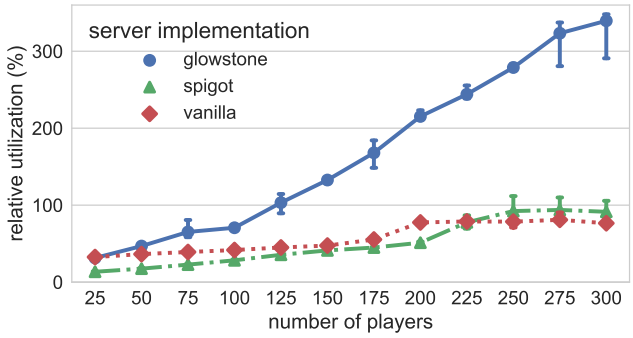


Figure 5: Relative utilization of Minecraft-like game servers with the fixed players workload. (Horizontal axis, markers, and whiskers as in Figure 4.)

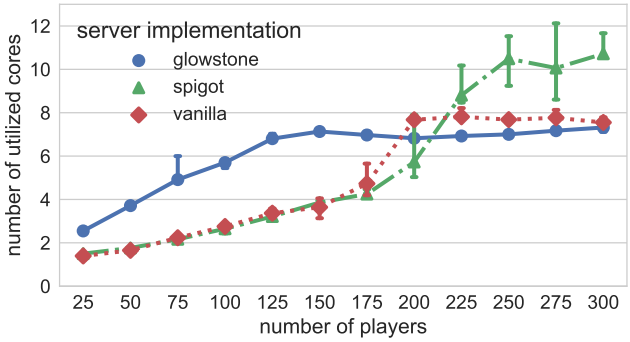


Figure 6: CPU utilization of Minecraft-like game servers with the fixed players workload. (Horizontal axis, markers, and whiskers as in Figure 4.)

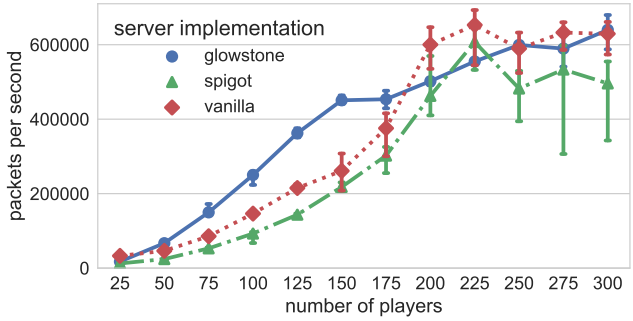


Figure 7: Outgoing packet throughput on Minecraft-like game servers when with the fixed players workload. (Horizontal axis, markers, and whiskers as in Figure 4.)

but it remains 20Hz for the fixed players workload. This indicates that Minecraft-like services react differently to different workloads.

Figure 5 depicts the relative utilization of the Minecraft-like game servers for the fixed players workload. The horizontal axis depicts the number of players connected to the server and the vertical axis depicts the relative utilization (see Section 2.2.1). We see that the relative utilization exceeds 100% for *Glowstone* when connecting 125 players or more, while for *Spigot* this only occurs when connecting 250 players or more. Likewise, for 200 players, *Glowstone* is running at approximately 220% relative utilization and is therefore overloaded, while *vanilla* is not overloaded, at approximately 80% relative utilization. This indicates that different Minecraft-like services react differently to equal workloads.

Combining Figure 4b and Figure 5 shows that the decrease in tick frequency coincides with exceeding a relative utilization of 100% for each of the Minecraft-like services. This suggests that the tick frequency of the servers decrease because the duration of each individual tick is larger than its maximum duration, delaying the execution of the next tick, hereby reducing the tick frequency.

Overall, we conclude that Minecraft-like services can currently scale up to hundreds of players per server-instance, whereas state-of-the-art MMO games, such as World of Warcraft and Runescape, can currently scale to thousands of players per server-instance. This gap, of an order of magnitude, provides a clear motivation for researchers to look for novel techniques to increase the scalability of Minecraft-like services.

4.3 Minecraft-like Services are Poorly Parallelized and this Forms a Bottleneck

The level of parallelism a service can use gives often an explanation for the scalability of the service.

Figure 6 shows the number of used cores vs. the number of players, using the *fixed join* strategy (ID = 2 and ID = 4 in Table 1). Between 25 and 150 connected players, all games show a trend of increasing CPU utilization for an increasing number of players. Between 150 and 300 players, the CPU utilization of both *Glowstone* and *vanilla* stops increasing. The CPU utilization of *Glowstone* keeps increasing until 150 players. For a larger number of players the CPU utilization seems to be roughly constant at a value between 7 and 8 cores. The CPU utilization of *vanilla* keeps increasing until 200 players. For larger numbers of players, the CPU utilization stays slightly below 8 cores. None of the tested services uses all of the available cores at any time during the experiment.

Combined, Figures 5 and Figure 6 indicate that, whereas the relative utilization exceeds 100% for all games and continues to increase with the number of players, the number of utilized cores does not, and never reaches 32 (the total number of cores in the node). Hence, the game is spending more time on each game tick without consuming all available computational resources. This suggests that Minecraft is poorly parallelized. It is possible that the tested services do not fully parallelize the separate tasks in the game loop discussed in Section 2.2.

4.4 Network Traffic from Minecraft-like Services Increases with the Number of Players; Is Limited by CPU Utilization

We now analyze if the cap in parallelism observed in the previous section is due to network or CPU bottlenecks.

Figure 7 shows the number of packets per second transmitted by the Minecraft-like game server over the number of players while under the fixed players workload. The horizontal axis shows the number of players, and the vertical axis shows the number of packets per second sent by the server. All games show an increasing number of packets sent for an increasing number of players. The number of packets per second sent by *Glowstone* keeps increasing with the number of players. The number of packets per second sent by *Spigot* and *vanilla* also increases with the number of players, but stops doing so after 225 players.

Combined, Figures 5 and Figure 7 indicate the reduced increase in number of packets sent per second by *Glowstone* coincides with the game exceeding 100% relative utilization. The reduction in number of packets sent by *Spigot* also coincides with the game exceeding 100% relative utilization. The reduction in packets sent by *vanilla* does not coincide with the game exceeding 100% relative utilization.

However, when combining Figure 6 and Figure 7 we observe that the reduction in number of packets sent coincides with an upper bound in the CPU utilization of the game. *vanilla* seems unable to use more than 8 CPU cores. This can be seen in Figure 6, where

Table 6: Network packet distribution, *vanilla* experiment with 200 players using the fixed players workload. *freq*: frequency of occurrence. *potb*: percentage of total bytes. *aps*: average packet size. In bold-face font, the most important percentages. Asterisk (*): server-bound message, otherwise: client-bound.

Message Type	*freq* (%)	*potb* (%)	*aps* [B]	σ [B]	Size Distribution [B] 25%	50%	75%	Max
EntityPosition	**45**	2	9	35	9	9	9	32k
EntityHeadLook	**18**	0	3	38	3	3	4	33k
EntityPositionRotation	**16**	1	11	40	11	11	12	32k
EntityVelocity	**5**	0	8	146	7	8	8	64k
EntityTeleport	**5**	1	29	66	29	29	29	32k
EntityMetadata	3	0	25	459	7	7	32	63k
PlayerPosition*	3	0	25	72	25	25	25	32k
EntityStatus	1	0	5	0	5	5	5	34
SpawnObject	1	0	68	663	55	55	55	62k
PlayBuiltinSound	1	0	23	176	22	22	22	32k
ChunkData	1	**93**	32k	9k	31k	31k	32k	252k
MultiBlockChange	1	0	37	284	20	26	41	33k
BlockChange	1	0	11	147	10	10	10	32k

for 200 players or more *vanilla* is very close to, but stays below, a utilization of 8 cores. We conclude the CPU does act as bottleneck.

4.5 World Data is Responsible for Most Network Traffic; Player Position Updates Are Most Frequent

Table 6 shows a summary of the network activity from one of the repetitions of the fixed players workload. Most network traffic is caused by the server communicating player location data to the clients. Only packets that have a frequency of 1% or more have been included in the table. The first row of the table shows that 45% of the packets exchanged between the server and clients are `EntityPosition` packets. These packets communicate the location of an entity from the server to a client. The top five rows in the table show packets related to entity positioning. These packets account for almost 90% of all network traffic between the server and the clients. Whereas packets related to entity positions are the most frequently sent, the `ChunkData` packet, responsible for communicating the layout of the world to the clients, is responsible for the largest amount of network traffic generated by the server. From the obtained network trace, 93% of the bytes sent over the network belong to such a packet.

Overall, Minecraft-like games transmit a large number of packets which increases steadily with the number of players. All servers show an increasing number of packets sent per second until they become overloaded, at which point the servers show erratic behavior.

4.6 Glowstone Delivers the Worst Performance; Vanilla Performs Best

Figure 4a shows that *Glowstone* drops the tick frequency below 20Hz after connecting more than 150 players, while *vanilla* and *Spigot* can connect 225 players before reducing server frequency or limiting the number of players. Similarly, Figure 4b shows that *Glowstone* performs worst for a fixed number of players as well. Moreover, Figure 5 shows that Glowstone uses more of the CPU time available for each tick, regardless of the amount of players.

In contrast, *vanilla* performs best throughout the experiments. Figure 4a shows that *vanilla* can support more players than *Glowstone* before reducing server tick frequency, and that it does not limit the number of players as opposed to *Spigot*. Figure 4b shows that *vanilla* is the only server implementation that does not decrease the server tick frequency throughout the entire experiment. Finally, Figure 5 shows that the relative utilization of *vanilla* does not surpass the threshold of 100% and thus the server is never overloaded.

5 RELATED WORK

In this section we survey the body of related work, which we divide across traditional (generic) benchmarks and Minecraft-specific performance studies.

The many traditional benchmarks and tracing utilities typical in cluster environments, e.g., the SPEC and TPC consumer and database benchmarks, the NPB and HPCC parallel benchmarks, and the tracing utilities developed for the large data centers of Google and others, do not address the specific challenges of online games. In particular, previous work in this category lacks representative workloads and service-level metrics. Yardstick complements this body of work.

Considering only Minecraft-related work, relatively few performance studies exist to-date [7, 8, 5], and no benchmark has been proposed. In contrast, Yardstick extends and complements these studies, and proposes a benchmark.

Closest to our work, Alstad et al. [1] experiment with lifelike bots the performance of *vanilla* Minecraft. In contrast to Yardstick, they do not investigate: (1) application- and service-level metrics, (2) different Minecraft-like servers.

Likewise, Cocar, Harris, and Khmelevsky [5] investigate the impact of CPU-core affinity on the performance of *vanilla* Minecraft, with similar limitations

Manycraft [7] is a Kiwano-based [6] distributed architecture aiming to scale Minecraft. Relatively to Yardstick, the Manycraft experiments: (1) currently support only non-modifiable Minecraft environments, (2) are not compatible with the *vanilla* clients and thus requires installing additional software on the client machine, (3) suffer from the same drawbacks as Alstad et al.

Similarly to Manycraft, but with more drawbacks due to the early stage of the project, Koekepan [8] distributes Minecraft, but so far lacks extensive performance experiments.

6 CONCLUSION AND ONGOING WORK

Among the workloads typical in high-performance data centers, Minecraft-like gaming services are increasingly more popular, but their performance and scalability are still not well-understood. To address this problem, in this work we have designed, implemented, and used the Yardstick benchmark for Minecraft-like services.

At the core of the Yardstick benchmark is our system model for the operation of Minecraft-like services, which captures salient characteristics of Minecraft-like systems and the performance of

the core game-loop. Yardstick proposes a benchmarking system that subjects the Minecraft-like service to the test of a realistic workload, and produces system-, application-, and service-level performance metrics that follow the system model. The Yardstick workload simulates lifelike player behavior.

We have used Yardstick on the DAS-5 cluster environment to conduct real-world experiments with three Minecraft-like services. Our main findings include:

MF1 Minecraft-like services can scale to hundreds of players.
MF2 Minecraft-like services are poorly parallelized.
MF3 The Minecraft protocol leads to large amount of data, linearly proportional with the number of players.
MF4 Position updates dominate in frequency, but volumetrically, terrain data is responsible for most network traffic.
MF5 The Minecraft-like servers have different performance profiles.

We are currently exploring the implications of the performance and scalability limitations of Minecraft-like services; we aim to design new scalability techniques for this domain. In the future, we aim to perform more extensive parameter exploration on other distributed Minecraft-like services.

ARTIFACTS FOR REPRODUCTION

The Yardstick project adheres to the 2019 reproducibility standards of ACM and IEEE. For this project, we release all the output as free open-access data and free open-source software (and related documentation):

Data Available on Zenodo [17]
Software https://github.com/atlarge-research/yardstick

ACKNOWLEDGEMENTS

We thank ACM SIGSOFT (the CAPS program) for their travel grant. Projects Vidi, MagnaData, and COMMIT/ co-support this work.

REFERENCES

[1] Trevor Alstad et al. "Game network traffic simulation by a custom bot". In: *SYSCON*. 2015, pp. 675–680.
[2] Henri E. Bal et al. "A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term". In: *COMPUTER* 49.5 (2016), pp. 54–63.
[3] Ashwin R. Bharambe et al. "Donnybrook: enabling large-scale, high-speed, peer-to-peer games". In: *SIGCOMM*. 2008, pp. 389–400.
[4] Mark Claypool and David Finkel. "The effects of latency on player performance in cloud-based games". In: *NETGAMES*. 2014, pp. 1–6.
[5] Matt Cocar, Reneisha Harris, and Youry Khmelevsky. "Utilizing Minecraft bots to optimize game server performance and deployment". In: *CCECE*. 2017, pp. 1–5.
[6] Raluca Diaconu and Joaquín Keller. "Kiwano: A scalable distributed infrastructure for virtual worlds". In: *HPCS*. 2013, pp. 664–667.
[7] Raluca Diaconu, Joaquín Keller, and Mathieu Valero. "Manycraft: Scaling Minecraft to Millions". In: *NETGAMES*. 2013, 1:1–1:6.
[8] Herman Arnold Engelbrecht and Gregor Schiele. "Koekepan: Minecraft as a Research Platform". In: *NETGAMES*. 2013, 16:1–16:3.
[9] Jim Gray, ed. *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993. ISBN: 1-55860-292-5.
[10] Alexandru Iosup et al. "Massivizing Computer Systems: a Vision to Understand, Design, and Engineer Computer Ecosystems through and beyond Modern Distributed Systems". In: *CoRR* abs/1802.05465 (2018).
[11] Huiguang Liang et al. "Avatar Mobility in Networked Virtual Environments: Measurements, Analysis, and Implications". In: *CoRR* abs/0807.2328 (2008).
[12] SpigotMC Pty. Ltd. *Spigot*. 2018. URL: https://www.spigotmc.org/ (visited on 05/07/2018).
[13] Mojang. *Minecraft*. URL: https://minecraft.net/en-us/ (visited on 05/07/2018).
[14] Newzoo. *2016 Global Games Market Report*. Annual report of trends, insights and projections for the global games market. 2016.
[15] Lothar Pantel and Lars C. Wolf. "On the suitability of dead reckoning schemes for games". In: *NETGAMES*. 2002, pp. 79–84.
[16] Glowstone project. *Glowstone*. 2018. URL: https://glowstone.net/ (visited on 05/07/2018).
[17] Jerom van der Sar, Jesse Donkervliet, and Alexandru Iosup. *Yardstick Software, Tools, and Documentation*. Feb. 2019. DOI: 10.5281/zenodo.2511818.
[18] Jerom van der Sar, Jesse Donkervliet, and Alexandru Iosup. *Yardstick: A benchmark for Minecraft-like Services [Extended Technical Report]*. Feb. 10, 2019. URL: https://atlarge-research.com/pdfs/yardstick19techrep.pdf.
[19] Siqi Shen and Alexandru Iosup. "Modeling Avatar Mobility of Networked Virtual Environments". In: *MMVE*. 2014, 2:1–2:6.
[20] Siqi Shen et al. "Area of Simulation: Mechanism and Architecture for Multi-Avatar Virtual Environments". In: *TOMCCAP* 12.1 (2015), 8:1–8:24.
[21] Mirko Suznjevic and Maja Matijasevic. "Player behavior and traffic characterization for MMORPGs: a survey". In: *MMS* 19.3 (2013), pp. 199–220.
[22] Luís Valente, Aura Conci, and Bruno Feijó. "Real Time Game Loop Models for Single-Player Computer Games". In: *SBGames*. 2005.
[23] Reinhold Weicker. "Benchmarking". In: *PERFORMANCE*. 2002, pp. 179–207.
[24] Yi Zhang, Ling Chen, and Gencai Chen. "Globally synchronized dead-reckoning with local lag for continuous distributed multiplayer games". In: *NETGAMES*. 2006.