Implementing Dynamic Conits for The Glowstone Minecraft Server

Student:

Christopher Esterhuyse

Supervisor:

Dr. ir. Alexandru Iosup

March 2019

Executive Summary

This report describes the design, implementation and testing of a dynamic consistency mechanism for the open-source Minecraft server: Glowstone. Dynamic conits were found to be able to reduce the total clientward messages by up to 97% without significantly impacting the client's game experience. However, overall work time of Glowstone was reduced by only 6%. Rather than being an inherent limitation of the continuous consistency paradigm, our results show that these shortcomings are a consequence of the way Glowstone mutuates its state and prepares update messages. We discuss future implementation changes that would complement continuous consistency.

Keywords

consistency, bounds, optimization, networking

Contents

1 Introduction

Minecraft is a proprietary single- and multi-player game which offers players a vast sandbox to explore and modify. Minecraft, as proprietary software owned by *Mojang AB*, does not disclose its game source code, making it difficult to understand its scalability limitations. As such, Glowstone is a Minecraft server written from scratch to be compatible with the Minecraft server-client protocol. The number of clients connected to a game server severely affects its workload. Any game can only support a limited number of players before crashing or becoming *unplayable*. Previous research by van der Sar et al. [1] has shown that for Glowstone, this number is several hundred players.

The *continuous consistency* paradigm [2] of Yu et al. provides a means of increasing the performance of a distributed system by conditionally allowing nodes to de-synchronize their views on the system state. A continuously consistent system relies on *conit* (consistency unit) structures to measure and repair inconsistencies such that the measurements never exceed some predefined numeric *bound*. The definition of this bound depends on the system. The work of van der Sar et al. investigated the use of continuous consistency for Minecraft-like games, finding entity-movement messages[1] to comprise the majority (approximately 89%) of network traffic in Minecraft sessions with numerous clients [1].

This report details the design, implementation and experimentation on a variant of the Glowstone implementation – henceforth referred to as *modified Glowstone* – with an additional continuous consistency mechanism. This mechanism facilitates conditionally suppressing the transmission of entity-movement update messages to clients. This allows the server to temporarily desynchronize the views of clients within an acceptable bound, saving work overall in the process.

Following some background information and definitions, given in Section 2, the design of the continuous consistency mechanism for Glowstone is described in Section 3. Section 4 discusses how this design is realized in the implementation. Section 5 explores the effects of using the continuous consistency mechanism has on Glowstone. Primarily, we wish to determine whether and under what conditions the conit mechanism is able to improve the performance of Glowstone such that it can support more simultaneous players, or support the same players more efficiently. In Section 6, the experimental findings are discussed, as well as how they pertain to continuous consistency in general. Section 7 concludes the report with reflection on our important findings.

2 Background

This section lays the groundwork for the sections to follow. Sections 2.1–2.2 briefly explain the current states of Minectaft and Glowstone. Sections 2.3–2.6 provide definitions and terminology relevant to the rest of the report.

[1]'entity movement' is used as an umbrella term for all five five classes of update messages that mutate the position, rotation, head rotation or velocity of a single entity.

2.1 Minecraft

Minecraft[2] is a video game designed by Markus Perrson and owned by *Mojang AB*. For our purposes, we focus on its technical properties. The game places players in a three-dimensional simulated game world in control of a singular human-like player avatar. At the core of the experience is the procedurally-generated game world composed entirely out of different kinds of *blocks*, stretching infinitely in the two horizontal dimensions, and to a finite height in the vertical dimension. Blocks can be created or destroyed as a result of various player actions. The world is populated by *entities* not limited to human-controlled players such as cows or skeletons that can move around autonomously and interact with blocks or other entities. Each play session has the potential to generate blocks in entirely different arrangements. Players can interactin a shared game world if they connect as *clients* to some Minecraft *server* which hosts a Minecraft game in which clients can participate simultaneously.

2.2 Glowstone

As described in Section 2.1, for several Minecraft players to play together, they must coordinate through a shared server. Many server implementations exist to fulfill this purpose. In this work, we concentrate only on one such Minecraft server: Glowstone[3]. This server is entirely open-source, enabling us to change its behaviour to suit our needs.

To host its Minecraft game, Glowstone must operate in the manner common to many game engines. Computation is conducted in a potentially endless *game loop*, one iteration of which is called a *tick*. The role of a tick is to advance the state of the game-world's simulation in accordance with the passage of wall-time. The server continuously mutates the game world's data as defined by the logic of the game (for example, gravel blocks fall vertically while unsupported by a 'solid' block from below). *Play* emerges as a consequence of the client being able to *participate* in these events. This necessitates the server receiving *request* messages from clients, and sending *update* messages to clients. Together, servers and clients comprise a *distributed system*.

2.3 Continuous Consistency

In the context of distributed systems, a system is *consistent* when all nodes reach consensus on their shared state [3]. Discrepancies between states of nodes are here called *inconsistencies*. A distributed system is able to function correctly even if inconsistencies cause its node-states to diverge, as long as the inconsistency is managed. The extent to which this is possible depends on the nature of the system.

A distributed system is considered continuously consistent if it preserves a well-defined *consistency bound*. Such a bound can be understood in terms of the various *states* the system can be in throughout its runtime. The bound provides a mapping

[2]Homepage: `https://minecraft.net/`

[3]Homepage: `https://glowstone.net/`

to some partially-ordered *bound domain* (for example, the domain of real numbers: $\mathbb{R}$) such that the system is never in a state that maps to a value above[4] the bound. Accordingly, the bound must be defined *in terms of* some primitive unit that facilitates the mapping from the state- to bound-space. Many such mappings are possible, but we are particularly interested in mappings that preserve a notion of 'measuring' the inconsistency in the distributed system. These primitives thus represent granular *relationships* between system values, and are appropriately named *conits* ('consistency units') by Yu et al [2].

To follow, Section 2.4 outlines how continuous consistency compares with other consistency models for distributed systems. Section 2.5 explores the requirements for conits and bounds that can be considered to be *dynamic*.

2.4 Consistency Models

Depending on the nature of the distributed system, many approaches to managing inconsistencies between nodes may be possible which all preserve functional correctness, which is considered to be a requirement at all times. However, these different consistency *models* or *paradigms* may differ in terms of non-functional properties. For example, one model may differ from another by dedicating increased server resources to keeping track of the client's view of the state. Precisely which models are applicable depends on the nature of the system.

The space of possible consistency models can be arranged according to which *bounds* they are able to guarantee on the inconsistency present between nodes at runtime. At one extreme is *strong consistency*, which provides the tightest bounds on inconsistency possible: trivial bounds; node states are never allowed to diverge in the first place. Achieving such perfect consistency imposes extreme restrictions on what each node can do at any given time. By contrast, *optimistic consistency* exemplifies an approach often allows indeterminate state divergence, since inconsistencies are only repaired as consequential conflicts arise. In these cases, there is only the infinite bound to inconsistency. Depending on the circumstance, the knowledge that some bound is never exceeded may have intrinsic value. For example, a user of a system may require that their perceived bank balance is an accurate representation of the true balance no more than five seconds ago.

Continuous consistency generalizes this notion of *inconsistency bound* to a domain not necessarily limited to the two extremes chosen by strong and optimistic consistency. Thus, this paradigm is useful for distributed systems for which the optimal choice of bound necessitates a non-trivial trade-off between extremes of the spectrum; important consistency invariants may be preserved, while relaxing other requirements, usually to gain performance from the resulting change in the system's behaviour. *Knowing* the bound also often facilitates static analysis of interesting system properties, such as bounds to the *staleness* of some retrieved datum.

[4]The distinction between 'above' and 'below' is not meaningful until this bound is made concrete.

2.5 Dynamic Continuous Consistency

At first glance, the idea of a 'dynamic bound' may seem counter-intuitive; how can we assert some value is never exceeded if the threshold is able to change? A system has a dynamic bounds if is preserves a bound that is defined *in terms of* system properties that change over time. Furthermore, it is often useful to distinguish cases where the system whose bound is able to change in response to *external* influence.

As conits can be defined for arbitrary distributed systems, whether a system is considered 'dynamic' depends on the outcome of a semantic argument; two opposing views on dynamic consistency differ only in where they draw the line between *external* and *internal* influences. For our purposes, conits and bounds are dynamic if they meet either of two equivalent requirements: (a) they are defined in terms of factors that change in response to external influence, or (b) they are able to re-define their bound definition at runtime as a function of external influence.

Section 3 explores how the modified Glowstone is able to employ dynamic consistency to facilitate desirable flexibility. Section 5.2.3 provides experimental results that demonstrate its dynamism in action.

2.6 Message Properties

Here we introduce two useful properties that may be present for a class of messages in a distributed system: *erasing* and *fusing*. Section 4.1 explains the relevance of these properties to the modified Glowstone server.

Erasing We consider a class of message to be *erasing* if for any two messages x, y arriving at some node N in sequence, the impact on state at N after y is always entirely unaffected by omission of x. In other words, the arrival of y has *erased* any impact x might have had on the state at N. *Erasing* carries over to sequences, resulting a property where *only the most recent message* is reflected in the state at N after all messages have arrived. Intuitively, this hints to the utility of this property: in some cases, the omission of a message does not change the outcome.

Note that even if x erases y, the state in-between may show the effects of x. The class of messaging only reasons about the state *after* y.

Fusing Consider a sequence of two arbitrary messages x, y from class C sent to some node N. Let S_N be the state at N after receipt of both messages. If there exists some message z such that z could have been inserted in place of the x, y sequence, and N would still have ended up in state S_N, then C is *fusing*. This property holds even if the message z is arbitrarily large or complex. A trivial example exists where z represents x and y's changes as a list. However, we restrict our focus to cases where z is from class C, and has a fixed size. With this added restriction, we observe that x and y have been *fused* into z, resulting in a reduction in both the number of messages transmitted as well as the number of bits. As with *erasing*, this property is able to propagate through a sequence of messages $M_0, M_1, ..., M_N$ such that M_N erases $M_0, M_1, ..., M_{N-1}$.

3 Design

This section explains the intuition and design behind the continuous consistency mechanism for Glowstone.

3.1 Approach

As described in Section 2.2, all clients communicate with the Glowstone server to facilitate a shared game world. In short, clients send *requests* of state-changes to the server, and the server sends *updates* of game-state changes to keep the *views* of the shared state consistent between clients. As such, Glowstone must potentially update *every* client in response to *any* incoming request. Findings of van der Sar et al. confirm our intuition that with many clients, these update messages begin to dominate network traffic [1]. Entity movement is a common occurence in Minecraft, resulting in entity-movement type messages comprising upwards of 97% of total messages by count in some experimental simulations [1]. We observe that many of these updates can be omitted (a) without harm to the correctness[5] of the system, and (b) without significantly impacting the quality of play[6].

Sections 3.2–3.3 to follow describe the design of a *continuous consistency* mechanism for Glowstone. The resulting implementation should avoid sending unnecessary update messages, saving the difference in *work time.* As defined in Section 2.3, implementation of the continuous consistency paradigm requires the definition of *conits* and an *inconsistency bound* for the application in question. Sections 3.2 and 3.3 respectively provide these definitions for Glowstone.

3.2 Conit Definition for Glowstone

As per their definition in Section 2, conits must be *defined* for an application to allow the system to *measure* its inconsistency according to some partially-ordered domain. Below, the fundamental design decisions for these conits are enumerated:

1. **One conit per client**
   Many consequences follow from the choice of the granularity of the conit mechanism, namely: which different states can and cannot be distinguished by weight. We choose to assign one conit per client[7]. The reasons for this are two-fold: (a) Glowstone's implementation is already centered around the *collection-pattern*, in which movement messages for a client are all generated and transmitted at once, with control flow revolving around the client. (b) Tracking inconsistency per client assures that the server results in distribution of inconsistency *fairly*, as inconsistency per conit is managed independently. More details are provided in Section 4.1.

[5]In this context, the game simulation is *correct* if it still adheres to the game logic of Minecraft.

[6]'quality of play' is a non-functional requirement that attempts to capture what influence on system correctness is *perceptible to the player.*

[7]One could argue that our definition maps one conit per *pair of entities.* This is a matter of the way the original description of Yu et al. is interpreted. Therefore, we prefer the granularity which maps most neatly to the implementation described in Section 4.

2. **Conit measures entity-movement state only**
   Based on the findings of van der Sar et al. [1], this work targets what was found to comprise the overwhelming majority of network traffic: clientward entity-movement updates. Glowstone is able to reduce this traffic, due to the observation that clients often receive updates more quickly than would be necessary for a acceptable game experience. Additionally, these messages are all either *erasing* or *fusing*, reducing the information that must be transmitted to synchronize the client and server. The details of synchronization is described further in Section 4.

3. **Incremental divergence**
   The server introduces inconsistency per client incrementally by *capturing* (intercepting and destroying) messages bound for the client. Messages are *weighed* before being destroyed, producing a value in $\mathbb{R}$. The section to follow explains the *inconsistency bound* for Glowstone in detail. Here, it suffices that this bound reasons about the *sum* of these $\mathbb{R}$ values *per entity*, ie weights of messages carrying information about entity x to client C are *not* summed with weights for messages carrying information about entity y to C when $x \neq y$. At any time, such a sum of weights represents the *distance* between C's and the server's views of some entity.

4. **Atomic synchronization**
   Synchronization ensures that for some entity x, the *view* of client C matches that of the server. In contrast to the gradual increase in *distance*, synchronization resets this value to zero all at once. After synchronization, the server and client's views of the entity are identical. This cycle of gradual increase and sudden decrease result in synchronization events being somewhat evenly-distributed over time. This is experienced by clients as low *jitter*.

3.3 Inconsistency Bound for Glowstone

The *inconsistency bound* of the modified Glowstone server is a *safety property* of Glowstone, ie. the bound constrains which game-states are reachable at runtime. This bound is defined in terms of the chosen definition of conits, described in Section 3.2 above. Section 2.3 explains why this is the case for continuously consistent application in general. The following is invariant at runtime[8]:

$$\forall p, e\,(\,P(p) \wedge E(e) \implies d(p,e) \leq b(p,e)\,)$$

where:

- $P(p)$: p is a player-entity.
- $E(e)$: e is an entity.
- $d(p,e)$: b is a function mapping an entity pair to $\mathbb{R}$, assuming that p corresponds to a player with some client C. It represents the *inconsistency distance* between the server's and C's views of e.

[8]We do not consider exceptional events that would also impede the functionality of original Glowstone, a TCP session between client and server failing for example.

- $b(p, e)$: d is a function mapping an entity pair to $\mathbb{R}$, assuming that p corresponds to a player with some client C. It represents the maximum distance allowed between C's and the server's views of entity e. At this granularity, this value is called the *bound component*, to suggest its relationship to the singular *inconsistency bound* of Glowstone at large.

The functions b and d are concretely defined at initialization time, after which the computation of a specific value is referred to as a *query* of d or b. Their definitions are formulated in terms of variable properties of game data at runtime. As such, Glowstone exhibits a *dynamic inconsistency bound*, as it is defined in Section 2.5. Concretely, these parameters are combined into *polynomials*, whose various scalar components are exposed for specification in a configuration file. Table 1 lists these variables and which function they can influence.

For function	Description
$d(p, e)$	constant
$d(p, e)$	constant if message encodes *movement*
$d(p, e)$	multiplier of distance moved by e if message encodes *movement*
$d(p, e)$	constant if message encodes *rotation*
$d(p, e)$	constant if message encodes *head rotation*
$d(p, e)$	in-game distance between p and e
$d(p, e)$	multiplier if p is facing away from e
$b(p, e)$	constant
$b(p, e)$	distance between p and e
$b(p, e)$	distance between p and e squared
$b(p, e)$	multiplier if e is out of sight of p

Table 1: List of the runtime variables accessible to variables d and b, described in Section 3.3. The concrete definitions of these functions is determined by how Glowstone is configured.

4 Implementation

This section explains how the design described in Section 3 is concretely implemented as changes to the Glowstone source code, written in the Java programming langauge. At this level, almost all novelties are introduced by the addition of a new structure associate with each client: `Conit`. 'Conit' is henceforth used for both the *logical* and *structural* units where the term is disambiguated by context.

4.1 Conit Mechanism

The modified Glowstone server shares most of its behavior with that of the original, as it outlined in Section 2.2. The versions begin to deviate within the `GlowPlayer` class. In modified Glowstone, an entity-movement message m carrying update information of target entity e bound for client C with player-entity p are *captured* by the associated client's server-side conit before reaching the network. This begins by invoking *weigh* paramaterized with the pair (e, m).

- **Weigh**
  weigh takes as input a pair (e, m) with e representing some *entity identifier* and m representing an entity movement update message. *weigh* defines a partial function of its input to $\mathbb{R}$. If *weigh*(m) returns a value, it is interpreted as the 'degree' to which omission of m increases the inconsistency between the states of the server and C. *weight* can be thought of as the manifestation of the *logical conit* in the implementation. In this case, m is consumed and *increment* is invoked paramaterized with the pair $(e, weigh(m))$.

  A *weigh* mapping is absent as a configurable function of the *entity type* of e. This allows further customizability of the modified Glowstone server at runtime. For example, the result of *weigh* may differ depending on whether e represents another player, a cow, or a skeleton. If m is not mapped, *weigh* returns `null`[9]. Such unmapped messages *fall through* the conit mechanism and are subsequently sent to C as they would be with original Glowstone. If the modified Glowstone is provided the *trivial configuration* (in which all *weigh* calls return `null`), this added check is the only difference in behavior from that of original Glowstone.

- **Increment**
  increment is provided input (e, w) with entity identifier e and $w \in \mathbb{R}$. The conit of C associates entity identifiers with *distances* in $\mathbb{R}$. The value associated with e, written $d[e]$, is updated to $d[e] + w$, representing the updated sum of all un-synchronized inconsistency-weights related to C's view of e. *check* is invoked with $(e, d[e])$ as input. If `true` is returned, *synchronize* is invoked with input e, and $d[e]$ is overwritten with zero[10].

- **Check**
  This procedure takes as input (e, D) with e, an entity identifier, and D in $\mathbb{R}$, and returns boolean `true` if $D > b(p, e)$ and `false` otherwise. This represents checking whether C's view of entity e has become inconsistent enough that *not* reducing the distance immediately would violate Glowstone's inconsistency bound, defined in Section 3.3.

- **Synchronize**
  synchronize takes as input e, an entity identifier. The conit constructs two update messages by accessing the internal state of Glowstone's game data. Data associated with the position, rotation and head-rotation of entity e are retrieved and used to generate two entity-movement update messages: `ServerEntityHeadLookPacket` and `ServerEntityPositionRotationPacket`, together comprising a payload of only 14 bytes[11]. These messages are immediately sent out over the network, bound for C.

[9]Concretely, this function returns a Java object of class `Float`. The value represents a reference to some floating-point numeric primitive. Java references can be assigned the special `null` value, representing the *absence* of an object. `null` and numeric zero and distinguishable.

[10]Concretely, the hash map backing these associations *evicts* the mapping for e. Section 4.2 explains why this is preferable and still correct.

[11]11 bytes for the former plus 3 bytes for the latter. This does not include the typical 28 bytes of TCP/IP headers which differ depending on the operating system, network characteristics, etc.

4.2 Conit State

In this section we explain how *structural conits* are represented in the implementation, and which data they manage. As explained by Section 3.2, conits only influence the transmission of messages associated with entity movement. Shown in Section 4.1, performing synchronziation thus only requires data that is already stored in the current game state of Glowstone, ie. a `Conit` does not need to store anything further to perform synchronization. However, this is not the case for determining *when* to do so, which requires computing the outputs of two functions, d and b, for various entity identifier inputs. Conits store *hash maps* to facilitate both of these functions. Below, we differentiate these structures.

4.2.1 Tracking Distance Values

Function d is defined in terms of *weights* of previously-sent messages. This concept is newly-added in Glowstone, and thus is only available if the conit explicitly stores message weights. The chosen inconsistency bound (given in Section 3.3) allows Glowstone to store only the *sum* of message-weights per entity-pair, called *distance*.

Modified Glowstone uses hash map D_{fresh} for retrieving and storing outputs of function d. At the end of every Nth tick, where N is a configurable parameter, D_{fresh} is purged of all mappings that map to identical values as a second hash map, D_{stale}. D_{stale} is then updated to be a replica of D_{fresh}. Intuitively, D_{fresh} is routinely purged of values that have not changed for N ticks. If a value is needed, but no longer stored in D_{fresh}, the conit must act *conservatively*, and consider the value to be *infinity*, certainly triggering synchronization.

Note that this mechanism of purging mappings is vulnerable to the *ABA problem*. Consider some mapping with value x in tick t. This value can increase enough to trigger synchronization, be reset to zero, and then increase such that it has value x again in tick $t + N$. This mapping may be purged, mistaken for a mapping whose value has not changed for N ticks. These cases result in unnecessarily early synchronization, which is not a threat to correctness. By its nature, such cases can happen at the most once per conit, per entity, per N ticks. However, most conit-configurations experience large variability in message weights, making these cases extremely infrequent.

4.2.2 Tracking Bound Component Values

Function b, returning the *bound component* for a given entity pair, requires only values already stored in Glowstone. Therefore, b does not require the conit to store any additional data. Nevertheless, the modified Glowstone server *caches* b outputs in a hash map which is evicted every N ticks, where N is a configuration parameter. Even if $N = 1$, this caching is beneficial, as numerous messages may invoke b with the same parameters. Within a tick, all messaging operations occur *after* the entire state has been mutated, ensuring that every message will compute the same outputs for b for the same entity identifiers. A user may opt to increase N further to use the cache more extensively, further reducing the number of b invocations. Note that if $N > 1$, queries may return values computed in a prior tick. Pragmatically, the difference is insignificant, and thus uninteresting. Theoretically, we can solve this problem by generalizing the *inconsistency bound*'s definition, stipulating that d and b outputs are permitted to be based on a game state for any of the previous $N-1$ ticks. This definition coincides with the existing one when $N=1$.

A further optimization exploits the *symmetry* of the bounds between any two entities a and b. Queries for the *bound component* of (b,a) are are translated to queries for (a,b) if $a<b$. This ordering is trivially achieved, as Glowstone uses the `Integer` type for entity identifiers. If a and b correspond to players, this simple modification halves the number of *bound component* computations. However, this operation has the consequence of separate conit structures *sharing* information, as clearly the conits for two players with entity identifiers a and b where $a<b$ will both access the mapping for (a,b). In the general case, this introduces a *causal relationship* between the queries of different conits; creating data races if queries are not protected by some concurrency primitive (which would add non-trivial overhead to this frequent procedure). Fortunately, parallelism is irrelevant in this case, as the only data race possible in this implementation would result in the bound component for some (a,b) being overwritten with a duplicate value, which does not impact correctness, and results in a cheap solution.

4.3 Encoding Staleness

Staleness models the intuition that the passage of time itself should increase the *weight* of existing inconsistencies. In the implementation described thus far, the conit mechanism will delay synchronization of a client's state indefinitely if no new inconsistencies are introduced. This is undesirable, as we assume that minor inconsistencies are more noticeable (and thus, more greatly impact a player's experience) the longer the player is able to observe them. For example, consider a cow at rest on a grass block after having jumped, while the client views it floating just above the ground. The server is aware of the small inconsistency, but as neither the cow nor the player are in motion, neither the *distance* nor the *bound component* for these entities will change, and thus no synchronization will occur. After a while, the player is sure to notice the cow appears to be grazing mid-air. The client is given an undesired view of an arbitrarily stale (old) state of the game.

Modified Glowstone avoids staleness by increasing every *distance* for every conit at a regular period of ticks. This is achieved by invoking *increment* (described in Section 4.1) but with a value computed in the absence of a message. Glowstone already tracks which entities are *known* to each player, excluding entities that are not close enough to the player to require their client be sent updates about them. The conit mechanism leverages this collection to avoid considering entities for which the staleness computation would be unnecessary.

5 Experimentation

This section details the experimentation on Glowstone to determine the effects of the continuous consistency mechanism.

5.1 Preliminaries

This section details the experimental setup for the testing Glowstone. Additionally, we define terminology that is referred to in Section 5.2, which provides experimental results.

5.1.1 Testing Environment

The purpose of the experiments is to observe the behavior of Glowstone when subjected to interactions with clients in as realistic a manner as possible. *Yardstick MC* by van der Sar et al. is able to generate network data which, when received by a Minecraft server, mimics the behavior of several connected clients engaged in play modeled after realistic movement from the game *Second Life* [1]. In this case, five nodes each host a single instance of Yardstick MC, which distribute the work[12] of simulating the clients evenly. Both client and server actors in this system are modified to make measurements of various predefined metrics, reporting them to a node running the *Prometheus* monitoring toolkit[13]. The experiment concludes when the last Yardstick instance completes the predefined interaction, after which the server and monitor nodes are terminated. A 'run' script collects and aggregates the Prometheus database alongside all relevant configuration, logging and temporary files into an archive, whose contents are later inspected. The details of this pipeline are elaborated in Table 4, available in the appendix.

All experiments were performed on 7 of the 68 distributed compute nodes of the DAS5 (Distributed Ascii Supercomputer cluster, generation 5) at the Vrije Universiteit, Amsterdam. Each Node has a dual 8-core 2.4Ghz processors, with 64Gb of memory, 128Gb of storage (with further storage on external hard disks). Nodes are inter-connected by a 1 Gbit/s Ethernet network.

Prometheus Key	Description
bots_connected	Count of the number of simulate clients connected to the server
bytes_in	Network traffic bytes incoming to the Yardstick instance
bytes_out	Network traffic bytes outgoing from the Yardstick instance
entity_position_updates	Sum of position updates for entities arriving on the client-side
no_sync	Sum of outgoing movement messages that did not trigger synchronization
yes_sync	Sum of outgoing movement messages that triggered synchronization
neither_sync	Sum out outgoing movement messages that were not captured by the conit
sendtime	Total time spent on sending movement messages on the server side
streamBlocks	Total time spent streaming chunk data to clients on the server side
tick	Total server time spent working on game ticks (and not sleeping)
tick_jobs	Total server time spent working on dispatch jobs within ticks
tick_network	Total server time spent collecting incoming network messages
tick_worlds	Total server time spent on world traversal and update messaging

Table 2: Metrics used by Yardstick MC and Glowstone programs. All but `bots_connected` are of the summary type, being recorded as a pair of sequences $*\_count$ and $*\_sum$.

[12]We use multiple nodes for clients as we do not wish for the machines hosting them to become overburdened and influencing the behaviour of the server under test.

[13]Homepage: `https://prometheus.io`

5.1.2 Metrics

The Prometheus monitor collects and aggregates information from all nodes in the simulation environment. By default, Prometheus employs a *node exporter* program to facilitate the collection of metrics that are common to such systems. Of these metrics, only the number of network packets in and out are reflected in our findings, though per-core CPU uptime and count of disk I/O operations were useful for preliminary, exploratory experimentation. Table 2 provides a full enumeration of all custom metrics used in collection of data that is represented in Section 5.2.

5.1.3 Conit Configurations

Modified Glowstone facilitates a trivial configuration (called C_0) which is able to avoid the continuous consistency mechanism, reverting to the behavior of *original Glowstone*. This is explained further in Section 4.1. Preliminary experimentation confirmed that this configuration's performance characterized original Glowstone indistinguishably. As such, C_0 is used interchangeably with original Glowstone in the experiments.

The behaviour of the Glowstone server is a function of its conit configuration. The implementation exposes a large number of parameters, resulting in highly customizable characteristics at runtime. There is no objective choice for the configured values that can be fairest or most-representative of everyone's interests. The configuration space is too large to explore exhaustively. Instead, this work selects three configurations, $\{C_{low}, C_{med}, C_{high}\}$ to represent a reasonable spectrum of non-trivial trade-offs between consistency and performance. These configurations are identical except for a 9× , 3× and 1× scaling of the inconsistency bounds and recalculation rate parameters respectively, relative to those of C_{high}[14]. C_{med} aims to represent a 'reasonable' configuration, whose parameters were selected for the desirable properties listed below. Note that these properties are expressed in terms of a real player's experience of playing Minecraft. As such, some properties are inherently subjective. C_{med} is provided in Listing 2 in the appendix.

1. A client's view of some entity e is more consistent if e has a smaller Euclidean distance in-game from the client's player. This results in Glowstone dedicating more system resources to synchronizing the states of entities with which players are more likely to interact.

2. Player within bow-shooting distance at equal height (used as an approximation of 'max interaction distance') synchronize frequently enough that the standard Minecraft client is able to interpolate the walking animation for both players.

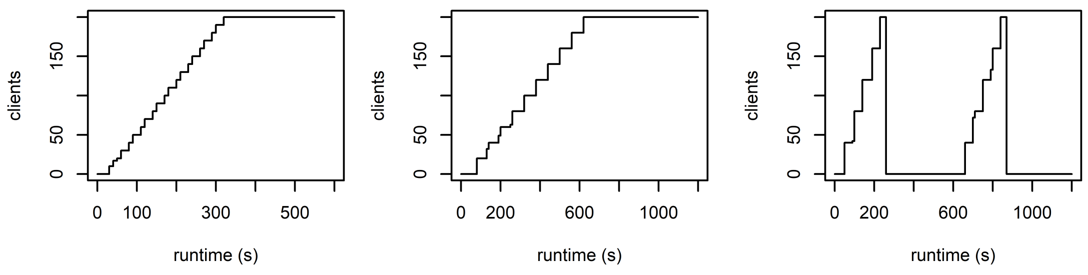
5.1.4 Experimental Workloads

Three workloads were selected for experimentation. Table 3 provides their definitions. Figure 1 shows the number of connected clients in one run for each workload, to give an overview of how these workloads appear in practice.

[14]Named `bounds:constant` and `bounds:ticks-per-recompute` in the file respectively.

Workload name	Definition
Workload 1	10 clients join every 10 seconds to a maximum of 300 (at time=300). At time=1200, all clients disconnect.
Workload 2	20 clients join every 60 seconds to a maximum of 200 (at time=600). At time=1200, all clients disconnect.
Workload 3	40 clients join every 45 seconds to a maximum of 200 (at time=260). At time=600, all clients disconnect. The process repeats from times 1000 to 2000.

Table 3: Definition of the three workloads used throughout experimentation.



(a) Workload 1 (b) Workload 2 (c) Workload 3

Figure 1: For each workload defined in Table 3, an example run is shown as the measured number of connected clients plotted over time.

5.2 Results

This section provides the results of experimentation on Glowstone. Figures provide a digestible view of properties of the server under various interesting combinations of *workload* and *conit configuration.* Additionally, surface-level descriptions and observations accompany the figures to provide context and aid in interpretation. This section ultimately makes a number of important observations:

1. Even the 'moderate' conit configuration C_{med} was able to reduce the total sum of sent clientward entity-movement messages by 97%.

2. The majority of tick-time is spent in a sub-task involving the recursive walk and mutation of the world-data.

3. The difference in total work time per tick was observed to range from 3.4% to 7.8% for the three selected static-conit configurations. This was not enough to significantly change the client capacity of Glowstone.

5.2.1 Glowstone without Conits

To establish a baseline, the trivial configuration C_0 (without continuous consistency) is subjected to each of the three experimental workloads[15]. Workload 1, shown in Figure 1a, puts Glowstone under mounting stress as it gradually accumulates more and more connected clients.

[15]For brevity, configuration names such as C_0 also refer to Glowstone when using them.

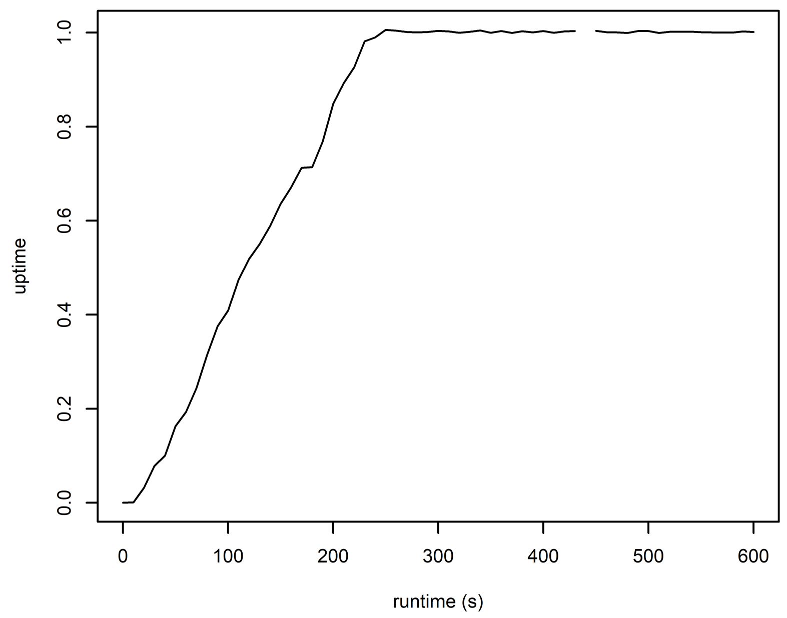


Figure 2: Total server uptime of C_0 for workload 1. Measurements shown are the mean of 4 runs.

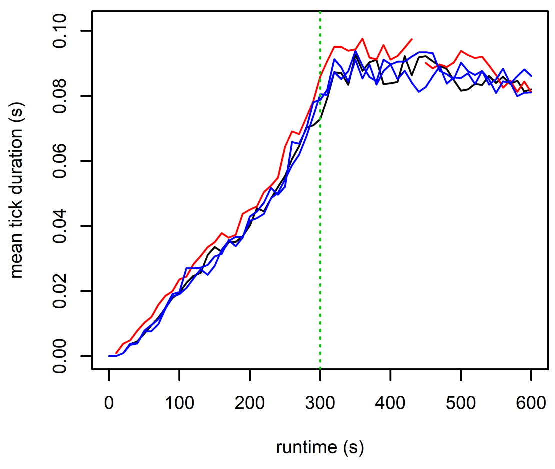
In Figure 2, we observe that the server reaches 100% uptime[16] after approximately 150 of the 200 clients have connected. Note that throughout this section, plots appear to be missing data points, manifesting as *holes*. These are an artifact of the clocks of Glowstone and Prometheus drifting slightly out of sink, resulting in a *scrape* being performed before Glowstone has submitted its measurements.

Figure 3 shows the total tick duration at each moment of the experiment. Note how the duration continues to climb beyond the 100% uptime mark is reached. Where is the extra time coming from? As seen in Figure 4, the server will postpone the next ticks until work for the current tick has been completed, effectively slowing down time within the simulated game world. While this mechanism is elegantly adaptable to changing workloads, it raises the question of whether the server ever reconciles the lost ticks.

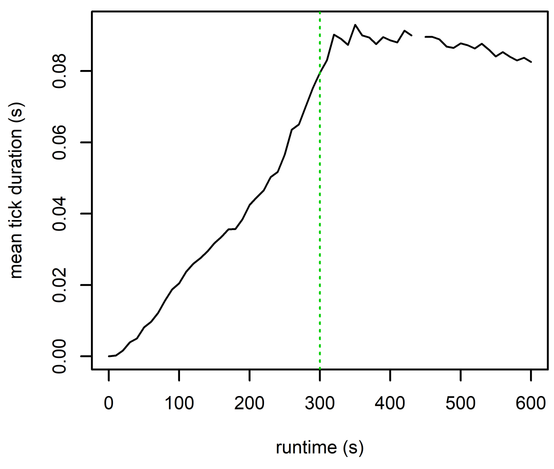
Workload 3, seen in Figure 1c, exposes Glowstone to a fluctuating workload; clients connect and disconnect repeatedly. The tick rate of the server throughout this experiment is shown in Figure 5. As before, the rate drops in response to the server being overburdened. However, we see here that the rate recovers almost immediately once the clients are gone. Curiously, the figure seems to suggest that the tick rate spikes up sharply at the moment the server has a chance to get ahead. Figure 6 gives another view of the same occurrence: the server is catching up with the missed ticks all at once, restoring synchronicity with wall-time. We conclude that the tick duration is strongly tied to the server's immediate client load, being able to accelerate once the load is diminished.

Figure 7 shows the uptime of the server during experiment 3. The order of tasks

[16]As with 'workload', 'uptime' has multiple similar definitions. In this work it is defined as the *proportion* of wall-time spent active ('up'), as opposed to being idle.

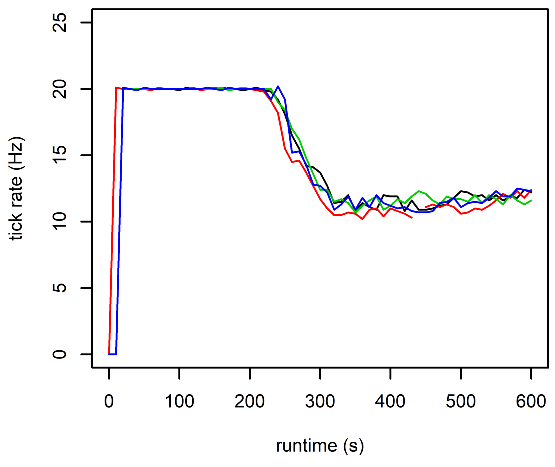


(a)

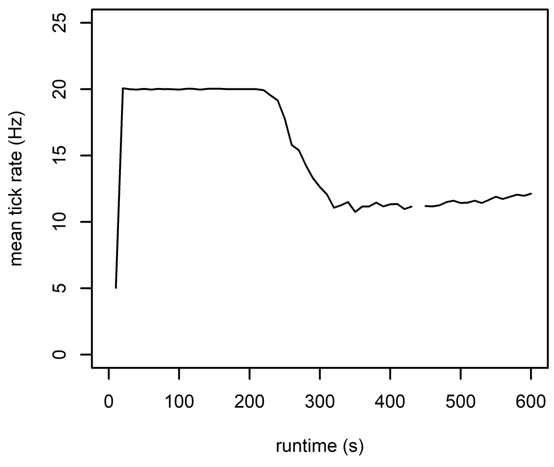


(b)

Figure 3: Mean tick duration of C_0 for workload 1, plotted as (a) four separate runs (b) the mean of four runs.



(a)



(b)

Figure 4: Mean tick rate of C_0 for workload 1, plotted as (a) four separate runs (b) the mean of four runs.

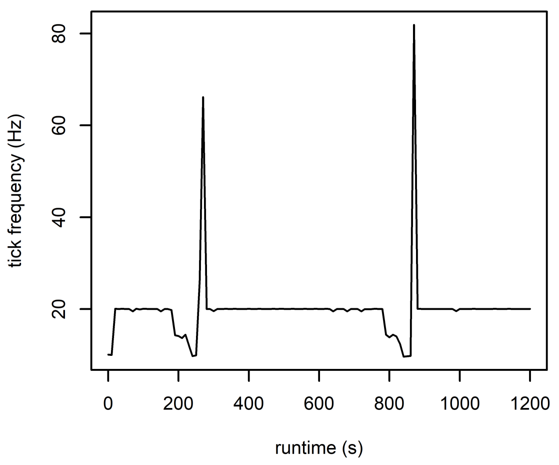


Figure 5: Tick rate of C_0 for workload 3, plotted as the mean of 4 runs.

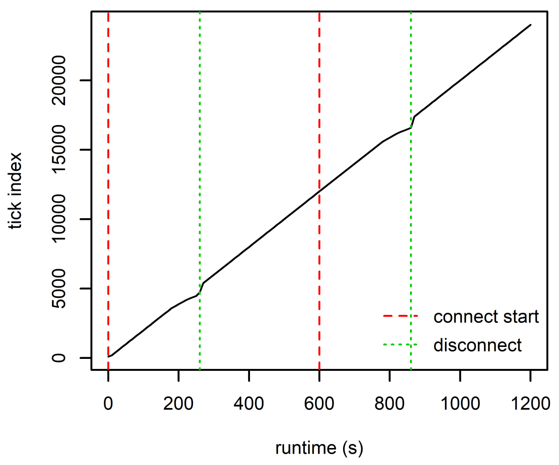


Figure 6: Tick ID of C_0, plotted as the mean of 4 runs. The plot shows the instants when clients begin the joining process and when all clients disconnect as vertical lines.

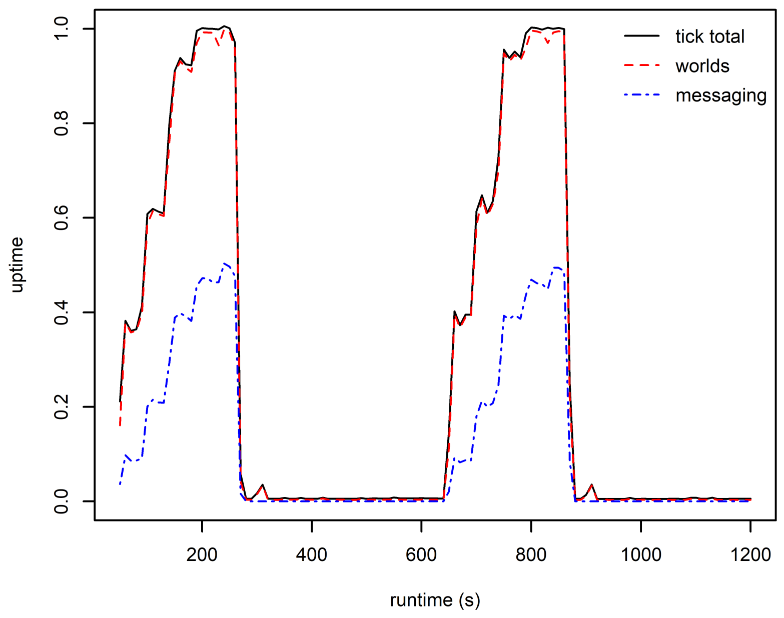
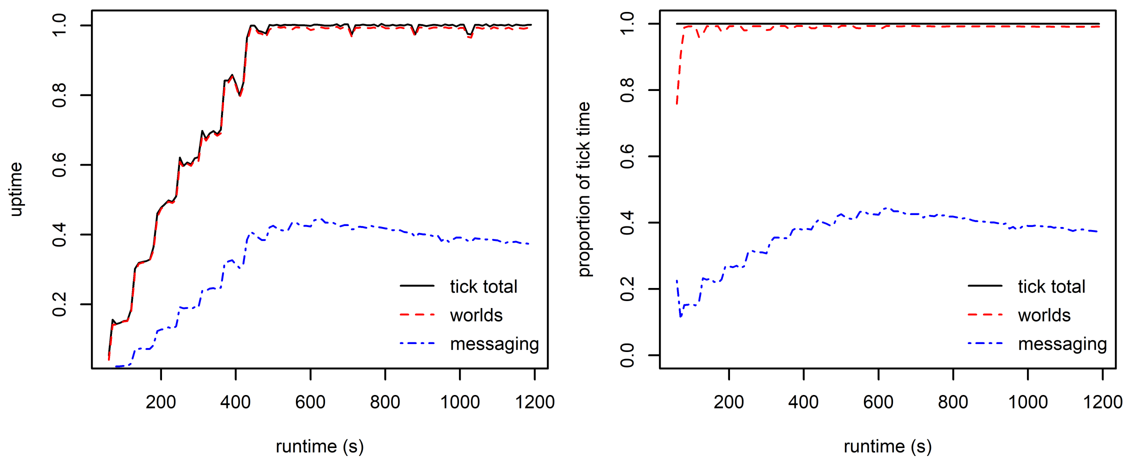


Figure 7: Uptime on the main thread of C_0 for workload 3, plotted as the mean of 4 runs. ‘messaging’ is a sub-task, of ‘worlds’, in turn a sub-task of ‘tick total’.



(a) (b)

Figure 8: Mean of 4 runs of C_0 for workload 2, shown as the (a) absolute uptime (b) uptime relative to 'tick total'.

performed by Glowstone in each tick is neither well-documented, nor immediately apparent from inspecting the source. Along with Figure 8, we observe a partition of tick time that evidently holds across heterogeneous workloads: the majority of work time is spent traversing and mutating the game state's world-data structures (this sub-task is called *world-tick*). Each world tick recursively invokes *pulse* on the objects that compose it, visiting child objects such that they can mutate their states to advance simulated time, fire collision events and so on. *Messaging* is seen in the figures to represent a sizable sub-task of world ticks, occurring once per tick, per pulse on an entity representing a player avatar. Owing to the all-encompassing hierarchical walk of world ticks, work not associated with any world takes negligible time by comparison; such as handling incoming network traffic and scheduling asynchronous jobs with an executor.

Next we explore what the executor is doing. By number, movement messages within the messaging task of each tick dominate the network. However, Minecraft's transformable world necessitates the transmission of large *chunks* of the data that comprises the state of the in-game simulated space. The tick-time dedicated to the task of streaming this chunk-data is exemplified in Figure 9. From Subfigure 9a, predominant, irregular spikes can be distinguished in the first 300 seconds of the run. These large streaming tasks can be associated with with the connection of new clients, necessitating the transmission all of the chunks in the player's vicinity at once. Also visible in the subfigure is a more steady background of smaller streaming tasks that increase in size with the duration of the experiment before plateauing at approximately 400 seconds. These smaller tasks represent the transmission of chunks in the path of clients already connected as they move to parts of the world they have not recently explored. At first glance, these work times appear capable of contributing greatly to the server's bottleneck. However, examination of the source code reveals that while this work is *scheduled* from within the world-task, the work is actually performed by a job executor service. The time taken to collect and delegate

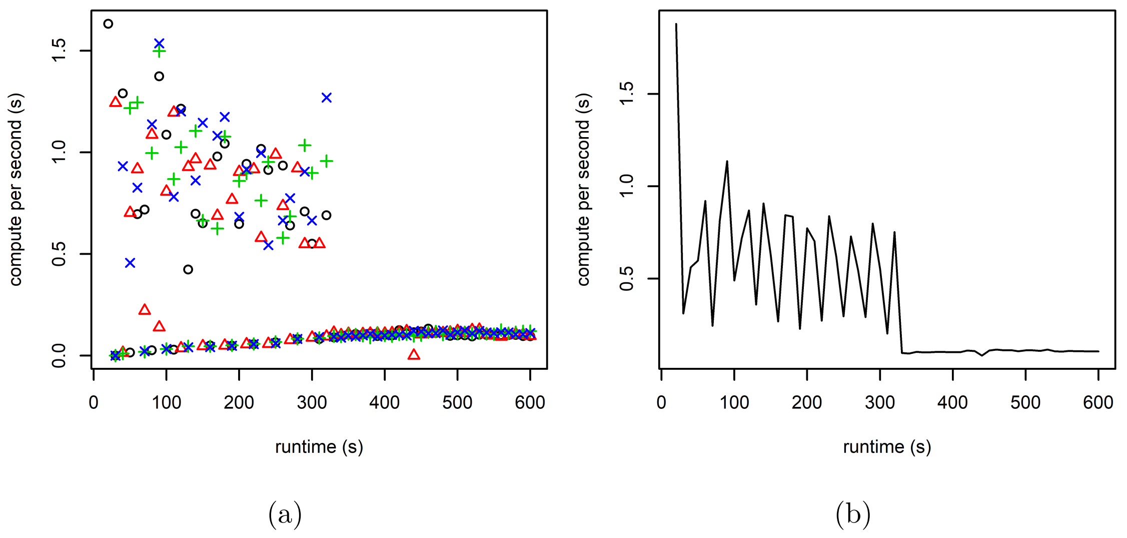


Figure 9: Total compute-time per second on streaming chunk data per second for C_0 for workload 1, shown as (a) a scatter plot of samples from 4 runs (b) the mean of 4 runs.

these scheduled tasks can be seen, for example, in Figure 8 as part of the tick work *not* in tick worlds, clearly negligible in comparison. This parallel execution of jobs explains how Figure 9 represents measurements of work-per-second exceeding 1. As block streaming does not apparently contribute to Glowstone's computational bottleneck, it is not explored further in this work.

Corroborating the findings of van der Sar et al., Figure 10a shows the vast number of movement messages the original Glowstone server is responsible for creating and sending to clients. Figure 10b exemplifies the expected quadratic relationship between clientward messages and the number of clients.

5.2.2 Glowstone with Conits

Figure 11 demonstrates the consistency mechanism in action. Here, C_{med} *captures* messages before they are written to the network, conditionally triggering synchronization. Each such event sends a set of messages to the client which carry the necessary information to repair the inconsistency incurred by the omission of messages since the last synchronization. These figures show that even with the moderate setting, an excess of 97% of potential messages never make it to the network. In the implementation, synchronization events incur the transmission of two network messages. Even so, this example shows a twenty-fold reduction in overall movement-messaging traffic to clients, visualized in Figures 12 and 13.

The *proportion* of messages that trigger synchronization can be seen in Figure 14 to change over the duration of the experiment. The resulting *rate* of outgoing messages is shown in Figure 15. Here, we introduce two new configurations to be compared with C_{med}, each of which represents a 3× scaling in the *bound* before inconsistency is triggered, with C_{low} and C_{high} synchronizing more and less frequently respectively. These two new configurations aim to represent the expected reasonable

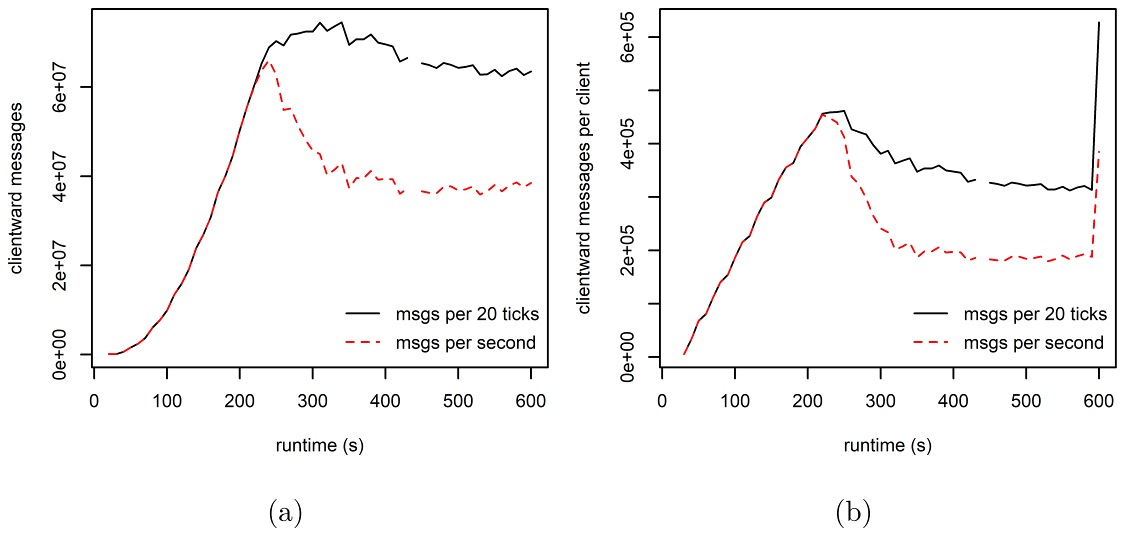


Figure 10: Number of clientward entity-movement type messages for C_0 for workload 1, shown as a function of wall-time as well as the number of ticks, shown as (a) cumulatively (b) per client as the mean of 4 runs.

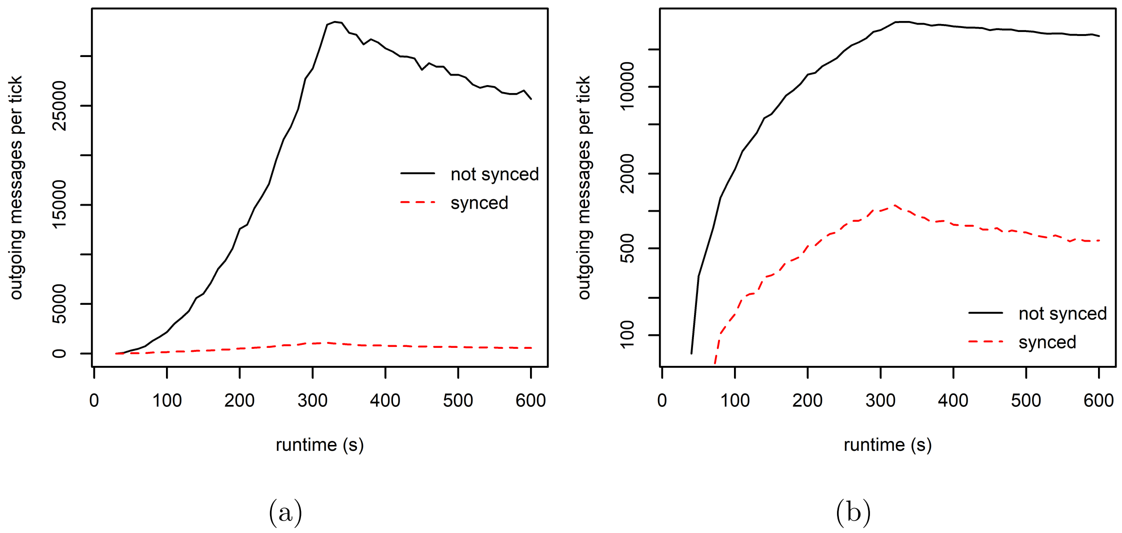


Figure 11: Total count per tick of clientward movement-messages for C_{med} for workload 1, shown as the mean of 4 runs. Messages are classified by whether they triggered conit-synchronization events. Subfigure (b) shows the same data on a logarithmic y-axis.

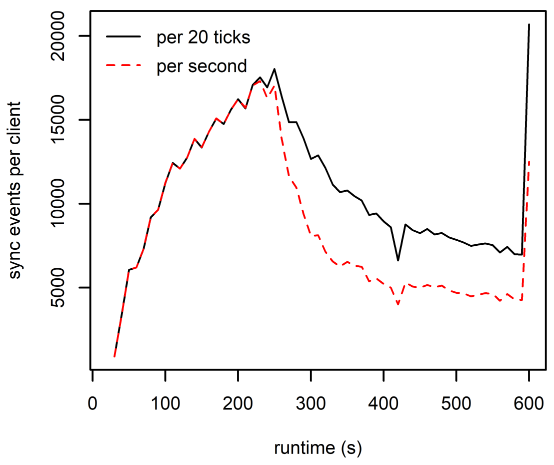


Figure 12: Counts of synchronization events triggered per client for C_{med} for workload 1, measured as the mean of 4 runs.

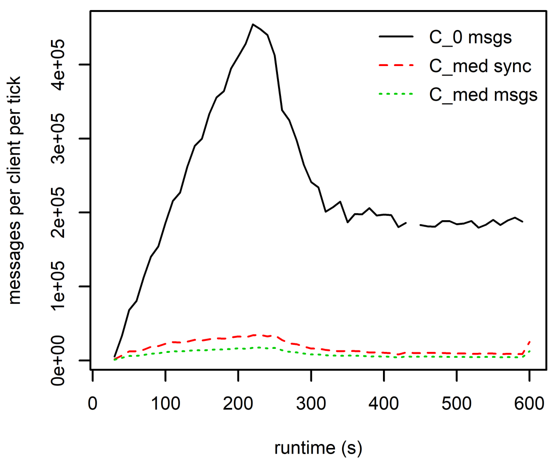


Figure 13: Total clientward movement-messages per tick, compared between aggregates of 4 runs each for C_0 and C_{med} for workload 1. For C_{med}, the corresponding number of synchronization events is also shown.

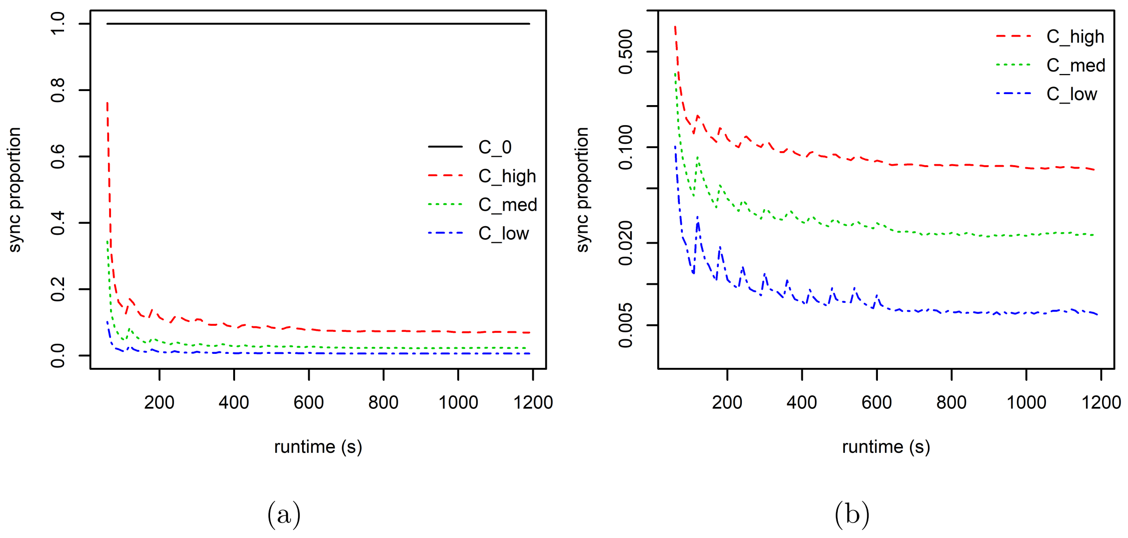
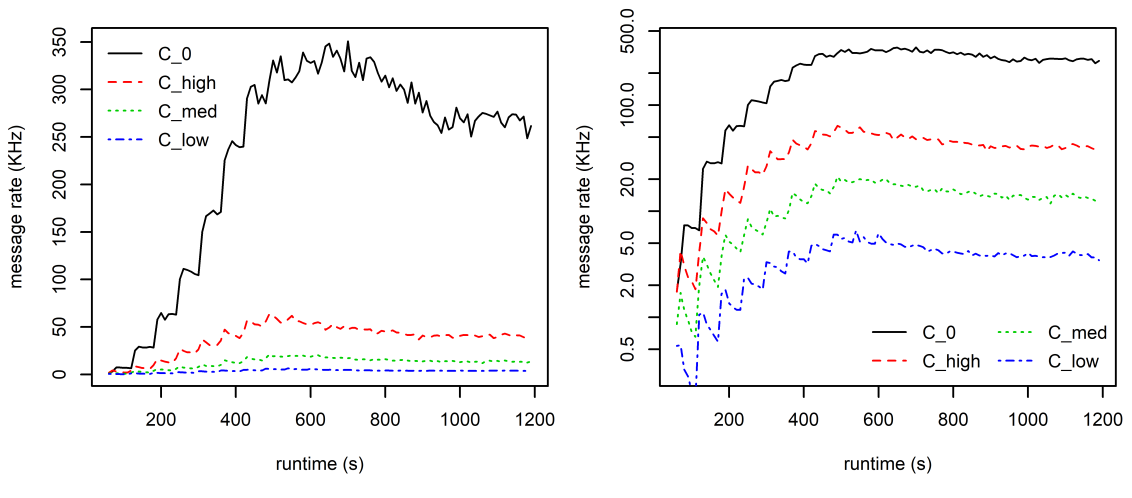


Figure 14: Comparison between Glowstone conit-configurations, showing the proportion of messages generated which arrive at the client, over duration of runs for workload 2. Subfigure (b) shows the same data, but plotted on a logarithmic y-axis.



(a) (b)

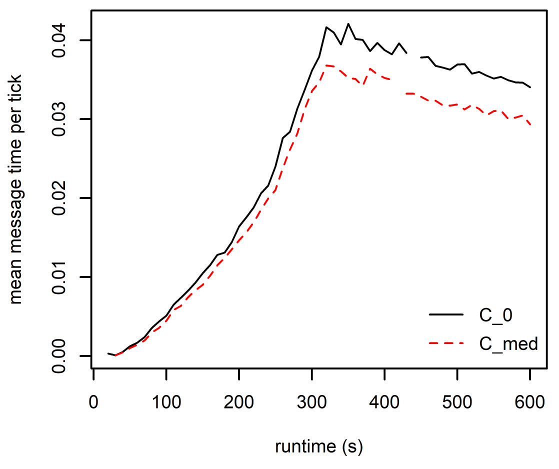
Figure 15: Comparison between Glowstone conit-configurations, showing synchronization rates over duration of runs for workload 2. Subfigure (b) shows the same data, but plotted on a logarithmic y-axis.

extremes that a real system administrator might consider for their Glowstone server. The figures show that their respective differences in resulting *inconsistency bounds* are clearly represented.

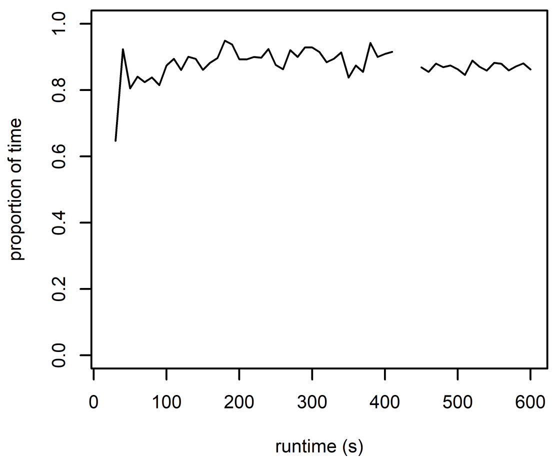
Figure 16a shows that for Workload 1, C_{med} experiences only a small reduction time spent on the messaging sub-task, despite sending far fewer messages than C_0 does. Figure 16b shows C_{meds}'s messaging work time as a proportion to that of C_0; this proportion stays close to 0.83 despite the fluctuations in client load. This suggests that C_{med} and C_0 have some sub-task of *messaging* in common that (a) requires similar work time regardless of whether messages are sent on the network, and (b) is proportionate in size to the total work for messaging.

The same outcome is replicated consistently for Workload 2, shown in Figure 17. In this case, all our conit configurations were seen to exhibit similar behavior, each with slightly different but nevertheless consistently large *proportions* of *messaging* work the conit mechanism fails to avoid. Ultimately, these small savings are even smaller at the scale of mean tick time, seen in Figures 18 and 19. The conit configurations reduced this time by only 3.4%, 6.0%, and 7.8% for C_{low}, C_{med}, and C_{high} respectively. Even for C_{high}, for which Glowstone produces almost no outgoing traffic, the messaging task appears to require significant computation time. Consequently, the overall performance gain is insufficient for the server to support significantly more clients. The differences between these configurations are still visible, manifesting as layering that matches our intuition: configurations that synchronize more frequently have longer tick times. However, these differences are dwarfed by the proportion of work that all configurations (C_0 included) have in common.

The Glowstone server was not designed with continuous consistency in mind. Originally, there was no meaningful distinction between the messages the server *conceptually* sends, and those that *actually* arrive at the client. As such, the messaging task of both versions of the Glowstone server, shown in simplified form in



(a)



(b)

Figure 16: Comparison between Glowstone conit-configurations, showing mean time messaging as the mean of 4 runs for workload 1 (a) of C_{med} and C_0 (b) of C_{med} as a proportion of that of C_0.

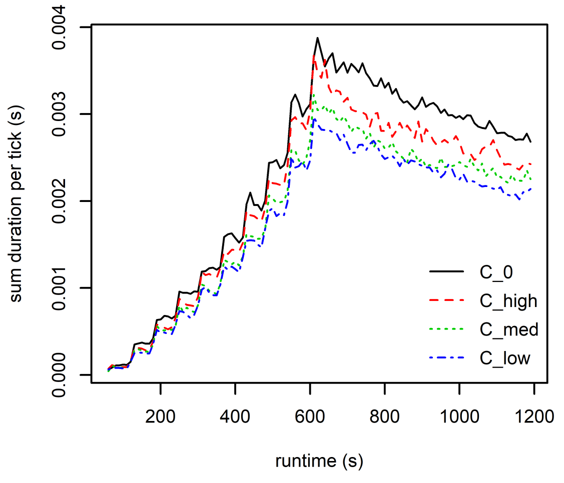


Figure 17: Summed duration of messaging tasks per tick for workload 2, plotted as the mean of 4 runs for each configuration in $\{C_0, C_{high}, C_{med}, C_{low}\}$.

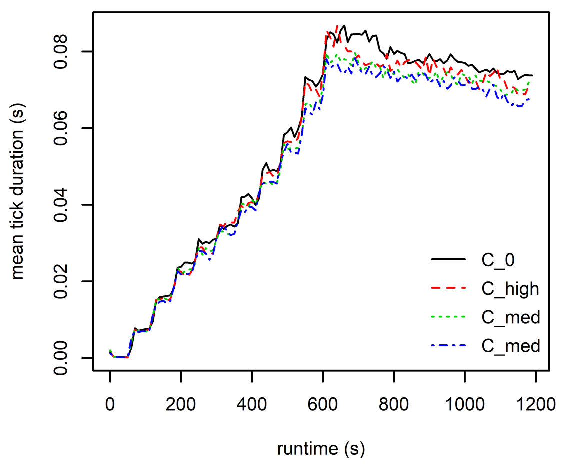


Figure 18: Mean tick duration for workload 2, plotted as the mean of 4 runs for each configuration in $\{C_0, C_{high}, C_{med}, C_{low}\}$.

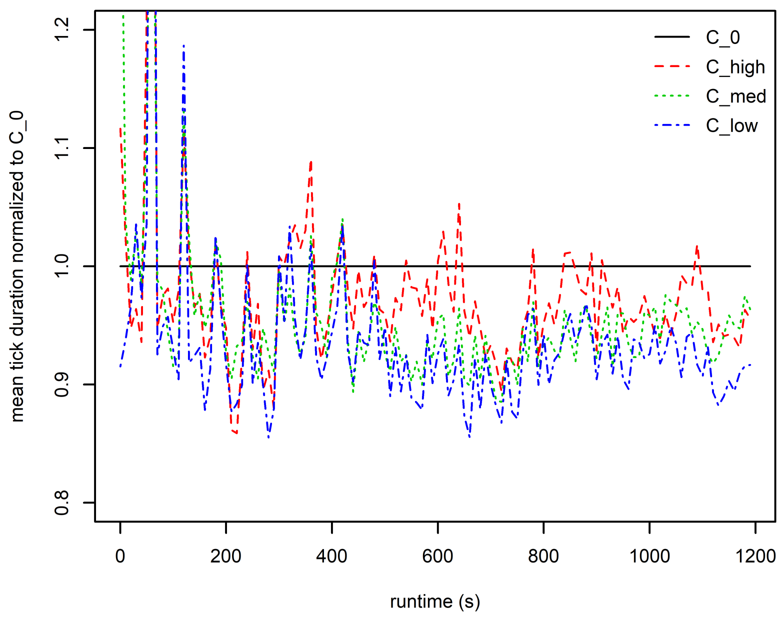


Figure 19: Mean tick duration for workload 2, plotted as the mean of 4 runs for each configuration in $\{C_0, C_{high}, C_{med}, C_{low}\}$. These measurements are drawn normalized to that of C_0 for comparison.

```java
for (Entity e : entities) {
    for (Message msg : e.createUpdateMessage(session)) {
        Float weight = conit.messageWeight(msg, e);
        if (weight == null) {
            // Fall-through case. Original message sent unaltered.
            session.send(msg);
        } else {
            // Conit captures the message. `msg' is discarded.
            boolean synced = conit.feedMessageWeight(weight, entity);
            // `synced`==true if conit just sent synchronization messages.
}    }   }
conit.maintain(); // Update caches, etc.
```

Listing 1: Simplified Java source code from Glowstone's `GlowPlayer` class, inside the `pulse` method, executed once per tick. The server traverses the world and fetches updates for the client of this `GlowPlayer`.

Listing 1, works by traversing the output of a black-box message iterator. From this level, conits digest messages that are somehow retrieved with each iteration step. The brevity of this iterator pattern belies the work it represents; for each message yielded, the caller thread performs the significant task of searching the contents of the game state and gathering information that must be relayed to the client as network messages. Per call, numerous disparate data structures are traversed, and significant bookkeeping and checking are performed. In its current form, Glowstone must spend time generating every message to completion before it is yielded by the conit mechanism, whose time savings are thus overshadowed.

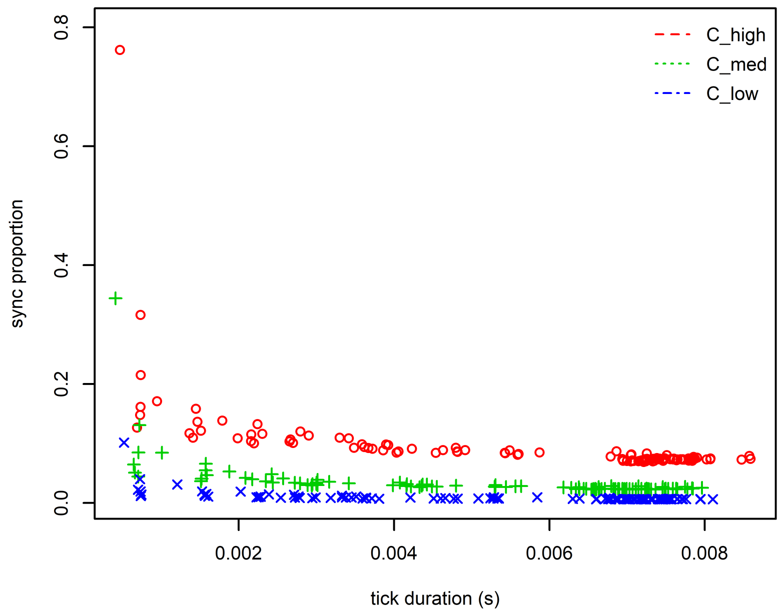


Figure 20: Proportion of messages that triggered synchronization plotted against the duration of the tick in which the message was sent. Values are sampled from runs of workload 2, for configurations C_{high}, C_{med}, and C_{low}. This plot represents the trade-off between consistency on one hand, and performance cost on the other.

Figure 20 shows the consequence of this unavoidable overhead: while the configurations can be distinguished when comparing them in terms of *sync proportion*, the differences between them are not significant for overall tick duration. In the figure, we see that the variations *within* a run are much more significant than variation *between* the runs of different configurations. In other words, Glowstone's conit mechanism fails to provide the interesting consistency-for-performance trade-off that characterizes continuous consistency. Presently, its ability to increase performance is seldom worth the reduction in tick synchronicity.

5.2.3 Glowstone with Dynamic Conits

The distinction between *static* and *dynamic* conits is dependent on the definition of what other properties of the system can be considered to be static. C_{med} could be considered dynamic, as it will synchronize players that are closer together in the Minecraft world more frequently than if they are not, where *distance* is a relationship between entities that changes at runtime. Regardless, we are interested in providing the server with the ability to change its bounds *arbitrarily*. As Glowstone already comes with an interactive console with commands and output, the system easily affords the ability for the *inconsistency bound* to be re-defined at runtime. This approach was not ergonomic for our testing toolchain. Instead, a trivial modification *scheduled* a number of changes to the bounds function to demonstrate this functionality as simply as possible.

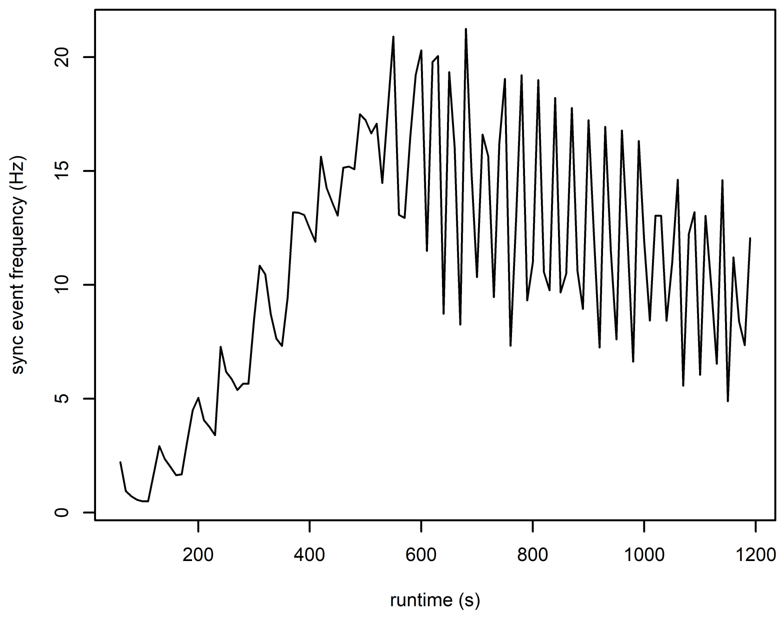


Figure 21: Mean synchronization event frequency for one run for workload 2 for Glowstone when configured to toggle between the behaviour of C_{low} and C_{high} every 90 ticks (4.5 seconds at 20-Hz tick rate).

Figure 21 shows the synchronization rate of the server when instructed to toggled between C_{low} and C_{high} regularly. Shown in Figure 22 are measurements from the test when repeated with a larger toggle-period between the two most extreme bounds-values possible: always synchronize and never synchronize. These extremes manifest as two clearly identifiable clusters in Figure 23.

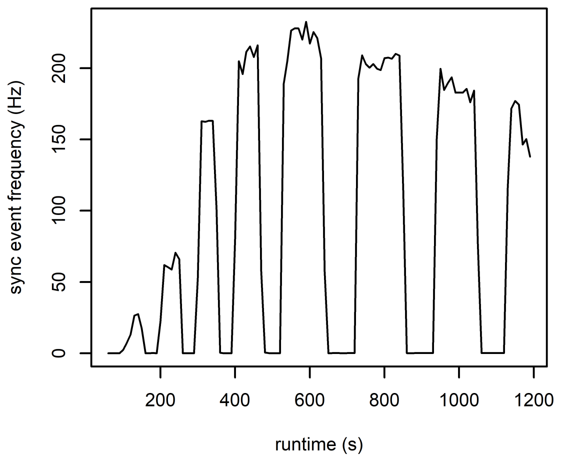


Figure 22: Mean synchronization event frequency for one run for workload 2 for Glowstone when configured to toggle between trivial infinite- and zero-bound conit-configurations once per 1000 ticks (50 seconds at 20-Hz tick rate).

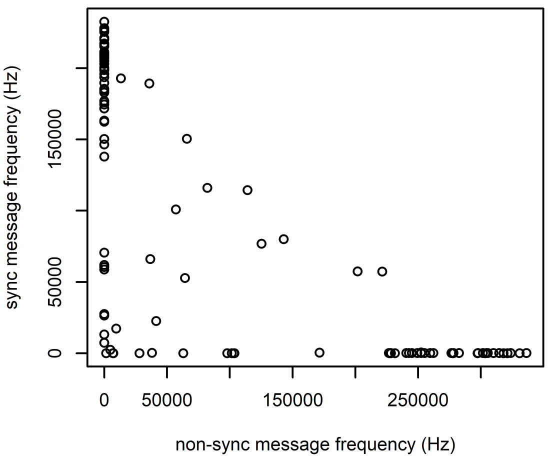


Figure 23: Samples of synchronizing vs. non-synchronizing message frequencies, sampled per time of a single run of workload 2 for Glowstone when configured to toggle between trivial infinite- and zero-bound conit-configurations every 1000 ticks (50 seconds at 20-Hz tick rate).

6 Discussion

Ultimately, the experimental findings in Section 5.2 suggest that the current implementation of the Glowstone server is unable to meaningfully increase the number of clients that can be supported by the server. While the mechanism is working as intended, only a portion of the cost per message is saved. This section explores changes that would have to be made to further improve the conit mechanism's ability to reduce overall work time. We also discuss which of these changes are specific to Glowstone, and which are likely to be relevant for the exploitation of continuous consistency in general.

6.1 Reducing Message Generation Cost

Experimental results suggest that, currently, generating the average movement-update message is more expensive than sending it over the network. Even for a game where entities are doing nothing but running around, we observe immense time spent traversing the world's data. Indeed, handling entity-movement messaging in the conit necessitates expensive traversal if the *weigh* and *component bound* functions are expressed in terms of variables distributed all over the world state.

A future work might attempt a more constrained (less expressive) conit implementation for movement messages, such that computation of these functions requires access to minimal state. Consider, for example, a conit defintion which is expressed only in terms of *staleness* and entity-pair distances. This would require hardly any state at all. It would be feasible to create a compact, sorted structure that can be used to quickly compute the relevant *weigh* outputs the moment a client's request message arrived. The cost of generating each message would be greatly reduced.

6.2 Leveraging Conits to Increase Parallelism

It is well-established that large matrix and vector operations are well-suited to parallelism, as the addition and multiplication of data items becomes trivial if threads have their own space in which to write, and the same operations to apply. If Glowstone's work was more like the matrix arithmetic, parallelism would allow for better throughput.

The conit mechanism effectively allows Glowstone to *simulate* sending update messages to clients by instead incrementing the associated *weight* value in a hash map. This operation is nothing more than simple arithmetic. The repetitive structure of these operations could be put to good use if they were translated into large matrix operations. If done well, a threadpool or a GPU would make short work of every *weigh*, *bound component*, and *check* process currently performed on the main thread. In the blink of an eye, an output matrix would, for example, give a dense bitfield telling Glowstone precisely *which* entities must synchronize their states with *which* clients (which could also be performed in parallel).

6.3 Reducing World Traversal Cost

Traversing and updating the world's state each tick was found to be a major contributor to work for the server, even in the absence of any mutation of the world aside from the player avatar's position and rotation. We conclude that Glowstone's current implementation is inefficient at performing such work, particularly since it appears to be work that scales with the number of clients (this rules out contributors such as grass growing and water flowing).

Investing the time to learn and benchmark every detail of the current implementation is out of the scope of this project. Rather, we simply conclude that Glowstone must be optimized to handle work that results from entity movement messages. For example, Glowstone is implemented in a very object-oriented fashion, storing references to game data in structures that correspond to *logical hierarchies*. While intuitive, this approach gets in the way of multi-threading, as object hierarchies tend to be *narrow and deep*, discouraging one of the most fundamental approaches to performance optimization: batching like-tasks. A future implementation might opt instead for something like an entity-component-system, which intentionally flattens and homogenizes the in-memory representation of logical hierarchies of game data. Flatter hierarchies encourage batched tasks that perform the same tasks faster as reducing pointer indirection reduces cache misses.

It should also be noted that games such as Minecraft are 'the perfect storm' for unavoidable work: (1) clients cannot know the layout of the world, so servers must spend resources informing them. (2) the contents of the world can be arranged in many configurations, reducing the amount of information that can be omitted from in-memory data due to being implicit[17]. Other kinds of distributed system may therefore be in a better position to benefit from continuous consistency, as the mechanism would be able to influence a larger proportion of task-time.

6.4 Augmenting the Communication Protocol

Choosing which part of the distributed state to manage with the conit requires a delicate balance; one must *maximize* the state such that more work can be avoided, but also *minimize* the state the conit is responsible for repairing.

For this implementation, this balance was struck by delegating to conits the task of managing the inconsistency of *entity movement*. This was suitable, as movement-update messages represent the majority of messages. Unfortunately, the Minecraft communication protocol lacks a way to update all of the position, rotation and head-orientation fields at once. Thus, a synchronization event necessitates the transmission of two distinct messages to the client. This is not a showstopper, as the experiments showed that message transmission is not a computational bottleneck. However, enriching the protocol to support a single `Sync` message would halve work that scales with the *number* of messages, such as the cost of invoking network-message handlers.

6.5 Modifiable clients

Owing to its proprietary nature, we do not wish to change the Minecraft client. However, removing this constraint opens new avenues for optimization. In our case, experimental findings suggest that reducing serverward traffic is unlikely to make a significant difference. This is not the case in general, as there are several reasons that client-side conits can be worthwhile.

For Glowstone, the ability to modify the client allows the system to become more performant in another way: client-side procedural generation. It is unclear to what extent chunk data streaming really affects the performance of the Glowstone server. Although it was found to be considerably time consuming, it does not *directly* contribute to the server's computational bottleneck, as this work is performed off the main thread. Thorough investigation of the mechanism of chunk-data streaming was not performed, but it is not hard to imagine that such a task might necessitate some *coordination* with the main thread. Assuming a cost is involved, it makes sense to reduce the work of chunk streaming.

The current Minecraft implementation does not exploit the predictability of the Minecraft world. Of course, a given terrain *could* contain any arrangement of blocks at all. This promises that regardless of what one does, if all clients are to agree on the world in which they supposedly co-exist, sending information about the world

[17]For example, checking for collision with water is easier and cheaper in a game like World of Warcraft than in Minecraft. In both cases, this check is necessary to determine whether the player is in water. However, the Mineraft world facilitates arbitrary placement of flowing water blocks. As such, deciding the presence or absence of water is an involved and expensive process. In World of Warcraft, by contrast, aside from some exceptions, the presence of water is a function of only the player's altitude.

between peers cannot be entirely avoided. However, in the vast majority of realistic[18] Minecraft games, the contents of a *chunk* of the world is very similar to its contents when it was first generated. In Minecraft and games like it, the vast expanses of unique game world can be generated from nothing but some *seed number* and the knowledge about the *coordinates* of the chunk in question. As the server is the computational bottleneck, we can lighten the load by moving the generation work to clients (conditionally, if needs be). If chunk-streaming tasks become faster, fewer server resources are used, and data-dependencies on the chunks being read are resolved sooner.

To prevent desynchronization, server and client can assure their deterministic generation algorithms match by exchanging hashes (or similar). Alternatively, the server could conceivably *transmit* the procedural generation algorithm to clients as data when they join. However, this case would require that care be taken to avoid introducing security vulnerabilities.

7 Conclusion

Conit-based continuous consistency offers a rich field of possibility for gaining performance while only relaxing requirements as much as the situation allows. These mechanisms can be considered yet another tool in one's belt, affording fine control over an important trade-off that exists in distributed systems we may otherwise overlook.

This work set out to demonstrate the potential of continuous consistency by enriching Glowstone implementation with a configurable module that allowed the server's administrator to dynamically trade-off consistency with performance, facilitating the support of more concurrent clients without sacrificing the server's ability to deliver its service. Unfortunately, the bulk of the computational cost associated with supporting a large number of clients proved to come unexpectedly from places beyond the influence of the mechanism. As a result, the impact of the configuration was not significant enough to increase the number of supportable clients meaningfully and reliably. Now equipped with a more detailed breakdown of Glowstone's performance characteristics, we provided a discussion of ways to improve weaknesses in the implementation. These improvements promise to be fruitful in their own right, but better still, they are designed to complement the continuous consistency mechanism. Implementing these improvements is beyond the scope of this project, but can serve as ripe grounds for future work.

Experimental findings suggested that the continuous consistency mechanism was capable of making drastic changes that could have great potential when applied in a different context. For example, the 'moderate' configuration setting was able to make a twenty-fold reduction in the number of clientward messages sent on the network without losing important user-experience properties; all the while the client is

[18]Even if players are intentionally working to maximize chunk-data entropy, even the most omnipotent of players cannot get around the fact that mutating the contents of the game world takes *time*. It stands to reason that no matter what, there will be game ticks the server must process in which a substantial portion of the world is largely unaltered.

oblivious to the optimization at work, requiring no modification whatsoever. Applications that facilitate a meaningful trade-off between consistency and performance stand to gain greatly from the deliberate use of continuous consistency.

References

[1] Jerom van der Sar, Jesse Donkervliet, and Alexandru Iosup. Yardstick: A benchmark for minecraft-like services. In *Proceedings of the International Conference on Performance Engineering, Mumbai, India, April, 2019*, 2019.

[2] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation-Volume 4*, page 21. USENIX Association, 2000.

[3] Wan Fokkink. *Distributed algorithms: an intuitive approach.* MIT Press, 2013.

Appendices

.1 Full Reference Conit-configuration 'C_{med}'

```
# defines which kinds of entities will use conit-logic for player update messages
enable:
  players: true # other players
  villagers: true # villager NPCs
  animals: true # cows, sheep, wolves, ...
  monsters: true # creepers, pigmen, skeletons, ...
  vehicles: true # minecarts
ticks-per-conit-clear: 200
# scales the (distance increment = 'weight')of a message according to the type of
 ↪  message
message-type-scale:
  # defines the weight of movement-type messages as c + d*x
  movement:
    constant: 2.0  # defines c
    # defines d, the sum of absolute values for distances in all 3 dimensions
     ↪  where x is the distance
    distance: 0.2
  head-rotation: 1.3
  rotation: 2.0
# defines the tick-period for staleness increments. Increase to update more
 ↪  rarely
staleness-period-millis: 3000
# when entities are out of sight of the player, message weights are multiplied by
 ↪  this value (suggest 2.0)
out-of-sight-weight-multiplier: 0.2
# defines d, where current distance is x and next distance is x+d
# d = mx + c
staleness-func:
  constant: 0.0 # defines c
  multiply: 1.0 # defines m
# defines the bounds computation between two entities
# C*(xd + y(d)^2 + c)
bounds:
  ticks-per-recompute: 30 # how frequently bounds are recomputed
  constant: 30.0 # defines C. Increase to update less frequently.
  # ratio of contributions for each component to bounds computation ()
  # bound is computed as xd + y(d)^2 + c, where d is the geometric distance
   ↪  between the two entities
  numeric-components:
    constant: 10.0 # defines c
    distance: 1.0 # defines x
    distance-sqr: 0.0 # defines y
```

Listing 2: Configuration file C_{med} for the 'moderate' conit-enabled Glowstone server, used as an example for experiments in Section 5.2.

.2 Experimental Data Pipeline

Tool	Role
Yardstick (MC)	This Java CLI tool takes as arguments some parameters for a predefined experimental pattern (such as walk around, join every 30 seconds in groups of 5 etc.).
YSCollector	While not used directly, this small modification to Glowstone by van der Sar et al. inspired the Glowstone-Prometheus interface.
DAStools	This set of Python scripts, part of Yardstick [1], provided a high-level API for synchronizing nodes on the DAS5 on one task, in the correct order. It orchestrated the yardstick, Glowstone and Prometheus processes on their own nodes.
Prometheus	Monitored server and client nodes on the DAS5. The Prometheus server also acts as a database that can be queried for the recorded metrics. After experiments were concluded, the server was started to access the database.
Promtool	Supplement to Prometheus, this CLI tool submits queries to the prometheus server and responds with an ascii dump of retrieved values. In our case, as each database contained only the data of precisely one run, each metric was queried with semantics that boiled down to ”all values in the last 999999 years”.
db2r.py	This script performed the heavy-lifting of parsing the output of Promtool into a usable form. It took care of generating R vectors of x and y pairs in a format that aligned time values and already derived some useful composed data (eg: mean).
help.R	Some custom R functions performed aggregation of vector pairs, omitting values they don't have in common etc. such that composite plots can be created which contains values from many sources at once.

Table 4: Tools and dependencies that formed the pipeline all the way to the plots seen in this report, included to assist in reproducibility.

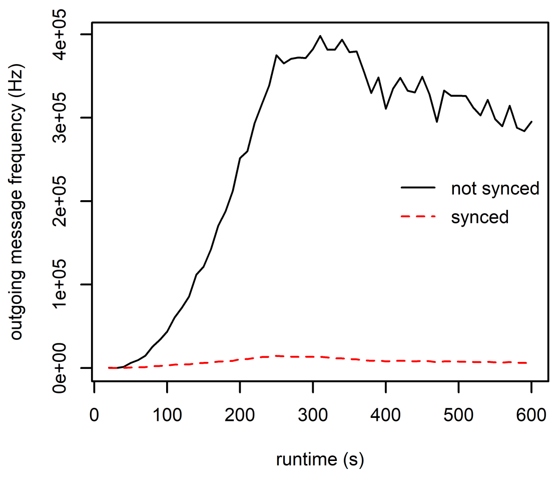
.3 Prometheus Metric Types

Measurements arrive at the *Prometheus* benchmarker by being ‘reported’ by the monitored node itself, then collected and aggregated by the monitor, whereupon the monitor-local database is updated. For various reasons, the acts of reporting and collecting a value occur at independent rates; to avoid data loss, both sides must agree on semantics for aggregating values. Prometheus defines two orthogonal systems for this purpose: *Keys* and *Metrics*. Reports require a Key to identify the group to which a reported value belongs, dictating with which other values it will be aggregated, thus allowing the exchange of an arbitrary number of independent measurements. The choice of Metric determines both the API exposed on the monitored-side, as well as the semantics for aggregation of values with the same Key. Prometheus offers several such Metrics, each with their own semantics and typical use cases.

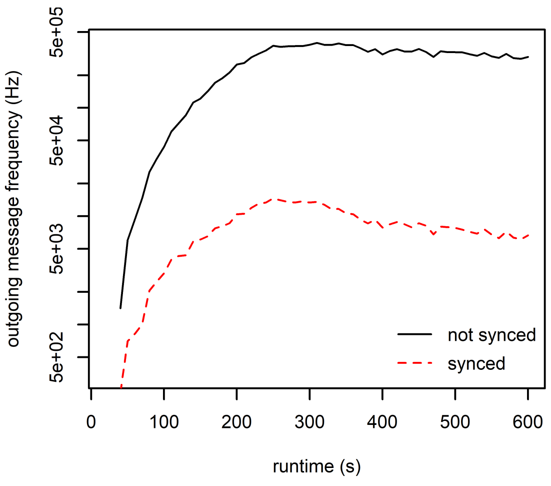
For our experiments, the vast majority metrics were classified as ‘Summaries’, which, expose a function to report a single floating-point value, v. Each invocation of this function increments a *counter* by one, and a *sum* by v. Ultimately, the database contains a sequence of sampled values, effectively associating points in time with a *tuple pair*, containing the *counter* and *sum* values as they were observable at that moment in time. While summaries may not be ergonomic at first glance, their counter-and-sum composition allow for easy composition and derivation of compound metrics. Summaries also are resistant to outliers, as their semantics effectively *average* values counted between sample points.

.4 Extra Result figures

This section contains figures that were not necessary for conveying the primary findings. For completeness, and to offer addition insight, they are nevertheless included here.



(a)



(b)

Figure 24: Frequency of clientward movement-messages for C_{med} for workload 1, shown as the mean of 4 runs. Messages are classified by whether they triggered conit-synchronization events. Subfigure (b) shows the same data on a logarithmic y-axis.