# Dyconits: Scaling Minecraft-like Services through Dynamically Managed Inconsistency

Jesse Donkervliet
*Department of Computer Science, VU Amsterdam, the Netherlands*
J.J.R.Donkervliet@vu.nl

Jim Cuijpers
*Department of Computer Science, VU Amsterdam, the Netherlands*
J.J.C.J.Cuijpers@vu.nl

Alexandru Iosup
*Department of Computer Science, VU Amsterdam, the Netherlands*
A.Iosup@vu.nl

***Abstract*—Gaming is one of the most popular and lucrative entertainment industries. Minecraft alone exceeds 130 million active monthly players and sells millions of licenses annually; it is also provided as a (paid) service. Minecraft, and thousands of others, provide each a Modifiable Virtual Environment (MVE). However, Minecraft-like games only scale using isolated instances that support at most a few hundred players in the same *virtual world*, thus preventing their large player-base from actually gaming together. When operating as a service, even fewer players can game together. Existing techniques for managing data in distributed systems do not scale for such games: they either do not work for high-density areas (e.g., village centers or other places where the MVE is often modified), or can introduce an unbounded amount of inconsistency that can lower the quality of experience. In this work, we propose Dyconits, a middleware that allows games to scale, by bounding inconsistency in MVEs, optimistically and dynamically. Dyconits allow game developers to partition offline the game-world and its objects into units, each with its own bounds. The Dyconits system controls, dynamically and policy-based, the creation of dyconits and the management of their bounds. Importantly, the Dyconits system is thin, and reuses the existing game codebase and in particular the network stack. To demonstrate and evaluate Dyconits in practice, we modify an existing, open-source, Minecraft-like game, and evaluate its effectiveness through real-world experiments. Our approach supports up to 40% more concurrent players and reduces network bandwidth by up to 85%, with only minor modifications to the game and without increasing game latency.**

***Index Terms*—dyconit, consistency, scalability, online gaming.**

## I. Introduction

Games provide entertainment to billions of players worldwide, forming an industry with annual revenues of over $175 billion [1]. We focus in this work on an emerging type of game, exemplified by Minecraft, which offers players a *modifiable virtual environment (MVE)*. With more than 200 million copies sold, Minecraft is the best-selling game of all time, and still has more than 130 million active monthly players [2] (more than the global number of MacOS users [3]). Microsoft has acquired Minecraft and started to operate it also as a subscription-based service [4]. Albeit less popular, tens of thousands of Minecraft-like games and services also exist [5], [6]. However popular, current MVEs can scale only by relying on small, isolated instances. Limited by bandwidth usage, these instances can individually scale only to a few hundreds of players even under favorable conditions [7], and the Microsoft Minecraft-service limits scaling to only 10 players [4].

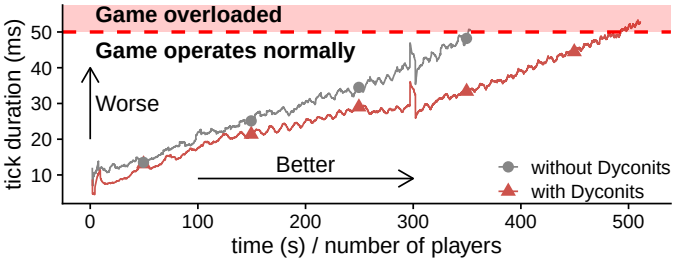



Fig. 1: Computation in an MVE for an increasing number of players over time. Without Dyconits, the game becomes overloaded after connecting 350 players. Using Dyconits, the game scales better, to more than 500 players.

Addressing the scalability challenge of Minecraft-like services, in this work we design and evaluate in real-world experiments Dyconits, an MVE middleware that reduces bandwidth usage through optimistically bounded inconsistency. With Dyconits, as Figure 1 depicts, Minecraft-like games and services can scale significantly better than with the current technology.

Minecraft is a prominent example of MVEs that operate real-time, online, and multi-user. Among virtual environments and online gaming worlds, the distinguishing feature of MVEs is that the player can modify all *world objects* (e.g., apparel, tools, systems) and the *virtual environment* (e.g., parts of the landscape, trees and their branches). This allows users to change radically the game world, to create complex new content from even many game-parts, and even to create dynamic systems by programming the game world. We detailed these aspects in a vision-article [8] and summarize in this work only the relevant technical aspects, in Section II-A.

Because the unique features provided by MVEs are beneficial for many kinds of applications, *tens of thousands of applications leverage the Minecraft-like pattern*. By allowing users to construct and deconstruct the world in complex ways, MVEs enable creative user-behavior that is currently not possible in other games. MVEs are also useful for other important societal tasks. Microsoft's Minecraft: Education Edition contains lessons on a diverse set of topics, for example, computer science lessons in which students construct their own digital computers and history lessons in which they explore UNESCO world-heritage sites. Minecraft has also been used

for social activism, for example, in protecting Europe's last primeval forest from illegal logging [9]. The Java edition of Minecraft is the hotbed of a large modding community, with tens of thousands of mods and tens of highly played modpacks that effectively create different games. The success of Minecraft has also triggered tens of thousands of other game developers, and Minecraft-like titles such as Dragon Quest Builders 2 and Lego Worlds are also massively popular.

Although Minecraft already entertains millions of players, it poses an important *scalability challenge*. Minecraft's current scalability is only achieved by replicating game instances that do not exchange state. Thus, players join a single instance, and do not interact in-game with users on other instances. Moreover, both recent academic findings and industry practice indicate the scalability of a single game's instance is limited. Minecraft instances, and several community-created games using the same protocol, scale to only 300-400 players even under favorable conditions [7]; Section V presents real-world experiments consistent with this. For example, the real-world experiment summarized in Figure 1 shows how the game latency ("tick duration" in the figure) increases with the number of players. For the regular game, for 350 players the game latency exceeds 50 ms, a known limit in first-person interactive gaming beyond which the player experiences the game-updates as slow (lag) and possibly loses a sense of immersion [10]. In contrast, Dyconits enables the game to scale further, up to over 500 players, before the latency limit.

*The scalability challenge is general*, going beyond only the Minecraft game. Minecraft Realms, the cloud-operated Minecraft service, limits to 10 the number of concurrent players in each world-instance [11]. The large body of games and services similar to Minecraft likely[1] suffer from similar scalability challenges.

*How to scale Minecraft-like games and services?* is the main research question addressed in this work. A tempting (but incorrect) answer is to "scale by credit card," that is, to buy more and better hardware. Such hardware would have to cover the computer systems and networks forming, end-to-end, the environment including each player's computer, the game servers operated by the company providing the gaming service, and the compute and network resources in-between. Because new technological waves change the minimal requirements of online games every few years, the environment would need to replace a majority of its hardware periodically, with good coordination across many stakeholders. Although shared infrastructure could lower costs for every game operator, the gaming industry does not use such approaches. Thus, however tempting, for the scale that Minecraft-like games have already achieved, this approach seems both unsustainable and, even without a detailed cost analysis, much more expensive than efficiency solutions such as the work presented here.

Existing and practical approaches typically follow a client/server architecture, where thousands of essentially non-communicating servers support up to millions of concurrently active players [12]. *Server bandwidth*, and sometimes even the server CPU, act as bottlenecks that limit the scalability of any individual server. To address this problem, state-of-the-art games use *interest management* techniques, such as area of interest (AoI) [13] and interest sets (ISs) [14]. Interest management trades off consistency for performance, by determining for each player which updates are likely to be important and/or interesting, and reducing the frequency of all other updates [15], [16]. However, interest management techniques cannot answer the question for two reasons. First, by lowering update frequency, these techniques risk creating large inconsistencies which reduce players' Quality of Experience (QoE), leading to a loss in customers and revenue. Second, these approaches are "always on," creating inconsistency even when this is not required to support the current workload.

A variety of traditional consistency models exist [17], but they have not been explored in the context of MVEs; we consider them in more detail in Section II-B. Promising for this work, because games and especially MVEs bound staleness, is the data-centric continuous consistency [18], but this relatively unexplored model has static bounds and thus seems incompatible with the dynamic nature of games. Overcoming this hurdle means addressing important *new* challenges, and a *principled transition from static to dynamic approaches*.

In this work, we design, prototype, and evaluate *Dyconits*, a novel middleware for MVEs that reduces bandwidth usage through optimistically bounded inconsistency. Our main contributions are:

1) The design of the Dyconit middleware (in Section III). The Dyconits system increases the scalability of MVEs by dynamically configuring, and optimistically bounding, the inconsistency between players. This middleware acts between the MVE's existing game-code and networking layers. Our work is the first that uses inconsistency to improve MVE scalability while simultaneously dynamically bounding inconsistency.
2) A Minecraft-like game using Dyconits (Section IV). We realize a full-scale game that replicates all the main features of vanilla Minecraft, based on the Glowstone community project [19]. We show how the Dyconits middleware provides game developers with flexibility in how they partition the game objects and the virtual environment, but also automates their management. We also demonstrate Dyconits are compatible with state-of-the-art scalability techniques such as AoI and ISs, allowing game developers to also leverage the benefits of these interest-management techniques.
3) Real-world experiments evaluating the scalability improvements when using Dyconits (Section V). We implement the Minecraft-like game based on Dyconits and, using the Yardstick MVE benchmark [7], conduct comprehensive real-world experiments. Our experiments cover system scalability, performance variability, and in-game inconsistency, under various settings and ever-increasing numbers of (emulated) players. The results

[1] Anecdotally, the authors of this paper experienced lack of scalability in several Minecraft alternatives. However, no systematic study currently exists.

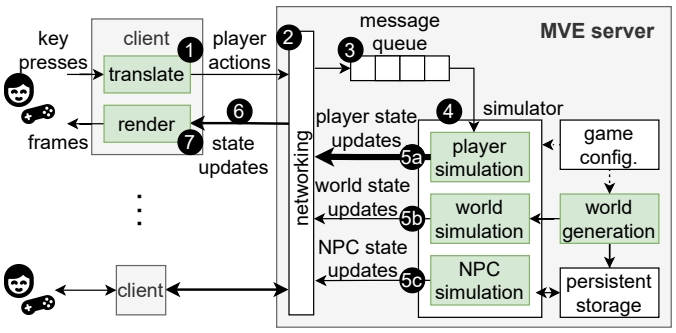



Fig. 2: Operation of a Minecraft-like online game with multiple players. Arrows indicate the main data flow; thicker arrows indicate higher data rates. The data flow runs at 20 Hz or more.

indicate considerable scalability improvements due to Dyconits, with exact values depending on the use of AoI, ISs, and Dyconits-specific policies.

4) A free open-source Dyconit system. We make it available as middleware, usable as a stand-alone Java library [20].

## II. System Model

In this section we present a model of operation and consistency for MVEs. Figure 2 is a visual overview of this model.

### A. Operational Model for Modifiable Virtual Environments

Commercial online games typically use client/server architectures [10], in which each player runs locally a client that connects to a server during play. The game server keeps track of the *global game state*, simulates state changes, and ensures all clients stay consistent with the changing global state (regardless of the high volume of updates).

The client translates user-input, such as key presses, into player actions (component ❶ in Figure 2) and forwards them over the Internet to the game server. The networking stack (❷) receives the actions, deserializes them into one or several game-messages, and enqueues them with other messages to process for the same player (❸).

The server runs the *MVE simulator*. Although both player-actions and state-updates create a continuous stream of messages, the simulator (❹) updates state in *ticks* (steps) with a fixed frequency of typically 20 Hz (*tick duration* up to 50 ms). During each tick, the server simulates the progression of time and all player actions. To avoid exceeding the tick duration, the simulator buffers player actions until the next tick.

An MVE simulator performs three main actions. First, validating and simulating player actions (❺a). Validation prevents players from cheating. Simulating players affects data such as their location and their interactions with the world. Second, simulating the world itself (❺b). For realistic worlds, this includes natural phenomena such as the movement of water and the growth of plants, and artificial phenomena such as the simulation of electrical circuits built by players. Last, simulating non-player characters (NPCs), i.e., the behavior of animals and other characters in the virtual world (❺c).

At the end of each tick, all state updates are sent back to the networking component (❻). Here, messages are serialized and sent to all clients. The client receives the state updates and uses them to render a local view of the virtual world for the player, typically at 60 frames per second (❼). To reduce latency, the client may speculatively apply player actions before it receives a state update from the server confirming its effect.

MVEs generate procedurally an endless world for players to explore. Together with player data, this world is stored in persistent storage. Unlike traditional games, which embed world data into the client statically, MVEs transfer the world data to players dynamically, chunk by chunk, only when the players access that chunk of the world. This is necessary because the world is *mutable*.

### B. Toward a Consistency Model for MVEs

Consistency is essential to good gameplay experience. Players experiencing inconsistencies lose immersion in the game world, and significant inconsistency causes gaming products bad reputation and loss of revenue. Current approaches take drastic measures to prevent this, disconnecting players when their state becomes too inconsistent, for example due to high network latency; but this too can lead to bad experience. How can MVEs manage consistency?

Many definitions and models for consistency already exist [17]. MVEs are highly interactive systems, in which possibly many clients send and receive a continuous stream of update requests to the same part of the world (data item). This makes them ill-suited for traditional client-centric consistency models where concurrent writes are assumed to be rare, because MVEs cannot reduce availability while synchronizing state under bursty writes.

To balance consistency and performance, online games use data-centric consistency models, coupled with updates occurring with a fixed frequency (in lock-step), and scalability techniques that limit the size and number of updates [14], [21], [22]. However, these approaches have limited ability to quantify and bound the inconsistency they create, e.g., they only correct inconsistencies between a few objects (the player avatars) and only manage to bound staleness. This works well for many current games but makes them unsuitable for scaling MVEs, where the objects include not only avatars but also every world component, so the number of possibly inconsistent objects often exceeds a few hundreds [7].

Instead, we use a continuous consistency model, based on a data-centric definition of consistency first proposed in TACT [18]. Under this model, inconsistency between any two replicas of the same data objects (consistency units, or *conits*) can be quantified and the quantities can be bounded. This model could address inconsistency both as a natural phenomenon (e.g., caused by network latency) and artificially imposed (e.g., by state-of-the-art game scalability techniques such as area of interest [21] and interest sets [14]).

However, the conit model has not been applied to gaming and MVEs before. Doing so raises important challenges: (1) it does not address the challenge of nodes (i.e., clients) joining

and leaving the system over time [8, challenge C2]; (2) it does not consider that clients can only change state speculatively; (3) it implicitly does not allow significant imbalance between the characteristics of the nodes running the replicas, because this would limit the performance to that of the weakest participant; and (4) to scale MVEs, as we show in this work, the model must become *dynamic* and be accompanied by policies that set bounds *dynamically*, in response to changes in player behavior and system workload. To address these new challenges, we design and propose in the next section *dynamic conits (Dyconits)*.

## III. Dyconit System Design

In this section we present our design for Dyconits, a middleware system for MVEs. We formulate the requirements of such a system and propose a novel design.

- **R1 Reduce system-wide network usage.** The bandwidth requirements of MVEs can increase quadratically, or even cubically, with the number of users [23]. This forms a scalability bottleneck, even when servers are provisioned from environments with high-performance networking resources, such as datacenters.
- **R2 Quantify and bound optimistically the inconsistency.** In real-time interactive systems, omitting or delaying state-updates causes inconsistency (but can improve performance). Both the type and magnitude of the inconsistency matter for the quality of experience [24], [25].
- **R3 Allow fine-grained control over system inconsistency.** Different state is important to different users. For example, users are most interested in changes in their immediate environment [14], [22] and are likely interested in the actions of their friends [16], but may be less interested in actions taken by users they do not know. Allowing inconsistency where possible, per-user, requires fine-grained control over the inconsistency in the system.
- **R4 Allow consistency bounds to be modified dynamically.** User interest is likely to change, over time or when users switch to a different activity [26, §3.2]. To prevent decreasing gameplay experience, the system must ensure low inconsistency for state that the user interacts with.
- **R5 Keep the system simple yet flexible.** The system must act as middleware. It will be used in MVEs deployed in distributed ecosystems, e.g., in clouds, as services; to this end, the system must have the ability to apply Dyconit-related policies, dynamically.

### A. Design Overview, Process, and Alternatives

This section presents the design of a Dyconit system for optimistically bounded inconsistency in MVEs. Focusing on MVEs, which synchronize state frequently and in high volume (see Section II), the Dyconit system limits resource consumption, even when disseminating state updates to large numbers of users, by bounding inconsistency between each client and the global state at a fine-grained level.

Figure 3 depicts our design. In partial fulfillment of requirement **R5**, the Dyconit system (blue area in the figure) acts

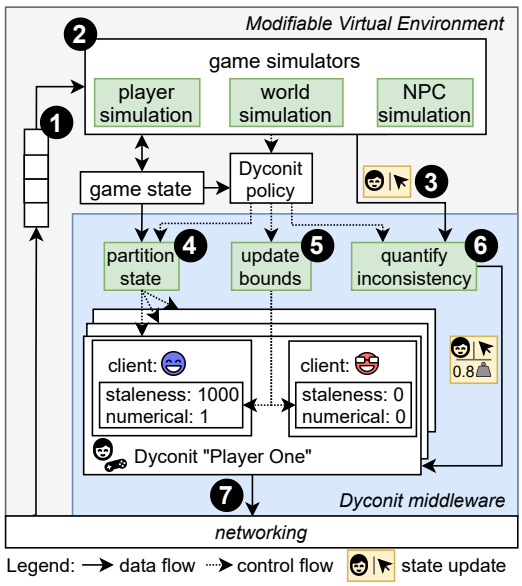



Fig. 3: Dyconit system design.

as middleware between the MVE's existing networking and simulation layers, and requires no client-side modification. The Dyconit middleware interacts dynamically only with the MVE server, which generates and forwards state-update messages. Incoming state update requests are unaffected by the Dyconit system. They are directly queued (❶ in Figure 3) for processing by the game's simulators (❷). However, the resulting state updates (❸) are no longer sent directly to clients, but are handed off to the Dyconit system.

The system quantifies and bounds inconsistency through the use of *dyconits* (first part of **R2**). A dyconit is a consistency unit representing an arbitrary subset of the game state. The subsets (i.e., partitions) are created dynamically (❹) by the Dyconit system using the selected policy (described in Section III-B). Each dyconit can bound the *staleness* of the state, e.g., ensure each game state has been updated at least once in the past 2 seconds, and the *numerical inconsistency*, e.g., positional differences between the locations of the same object across all clients; see Section IV-A for details on quantifying inconsistency. This enables inconsistency bounds, per player avatar, virtual-world object, or arbitrary combinations. The system updates inconsistency bounds dynamically (❺), based on the changing interests of each player. To bound inconsistency, the system quantifies the inconsistency caused by each state update (❻), forwards it to the corresponding dyconit, and evaluates the consistency requirements for each client. If a client's inconsistency bounds are exceeded, the system forwards all state updates to the client (❼).

The inconsistency introduced by queuing state updates *reduces network bandwidth consumption* (**R1**) in two ways. First, the system merges messages that write to the same state, allowing for large reductions in bandwidth usage for state that is modified frequently. Second, queuing state-updates allows them to be sent in batches, reducing system-level overhead

TABLE I: Overview of Dyconit policies.

| Policy name | Short name | Description | Dynamically responds to: | |
|---|---|---|---|---|
| | | | Player State | Workload |
| ZERO | No inconsistency | Creates a single dyconit. All players subscribe to it with an inconsistency bound of zero. Effectively prevents inconsistency by sending new state updates immediately to all players. | ✗ | ✗ |
| AOI | Area of Interest | Creates a dyconit for every chunk. Players set a bound of zero on the 9 closest chunks, and a 1-second bound on all other chunks. | ✓ | ✗ |
| IS | Interest Set | Creates a dyconit for every player. Players set a bound of zero on the 5 players in their interest set, and a 1-second bound on all other players. | ✓ | ✗ |
| ISN | Interest Set - Numerical | Creates a dyconit for every player. Players set a bound of zero on the 5 players in their interest set, and 0.16 units (slightly more than a single step) numerical bound on all other players. | ✓ | ✗ |
| ZERO/IS | Workload-based Interest | Uses the ZERO policy if relative utilization is less or equal than 90%. Uses the IS policy otherwise. | ✓ | ✓ |

such as packet headers.

Because the MVE does not reduce availability while synchronizing state updates, bounding the inconsistency is *optimistic* (second and last part of **R2**). To see why this is the case, consider a client that reaches its staleness bound on one of its dyconits. To prevent the inconsistency from becoming larger, the system must synchronize state to the client. However, synchronizing state is not instantaneous, due to network latency and processing delay. This means the time between sending the message to the Dyconits system and it being processed by the client becomes larger than the configured staleness bound. The system can compensate for this effect by setting the value of the staleness bound to the player's maximum tolerable latency and subtracting the estimated network latency and processing delay.

To enable fine-grained *control* over inconsistency (**R3**), the Dyconits system manages each dyconit, dynamically and automatically. Dyconits are re-configured dynamically (**R4**), to match consistency restrictions that benefit each player; our system maintains this process simple yet flexible through the use of policies (**R5**, as detailed in Section III-B). For example, the positional difference for an important game-object could be configured to be maintained with no inconsistency (numerical inconsistency of 0), for a nearby but less important game-object it can reach small numerical inconsistency, and for objects located relatively far from all players it can reach high numerical inconsistency.

Figure 3 shows an example of a dyconit, "Player One," associated with the game state representing one of the players in the game (). While the red player ( in the figure) is experiencing no inconsistency with regard to player one's state, the blue player ()) does experience some inconsistency. The blue player could tolerate the higher inconsistency, because they are located further away from player one than the red player, and due to the natural effects of perspective and distance can see less the impact of the positional differences. In traditional games, either the blue player would have been disconnected or the game would have slowed down until their client can catch-up (the lock-step approach in Section II-B). In the Dyconit design, the blue player can continue without delay to the game, because the Dyconits system deems the inconsistency acceptable for this situation.

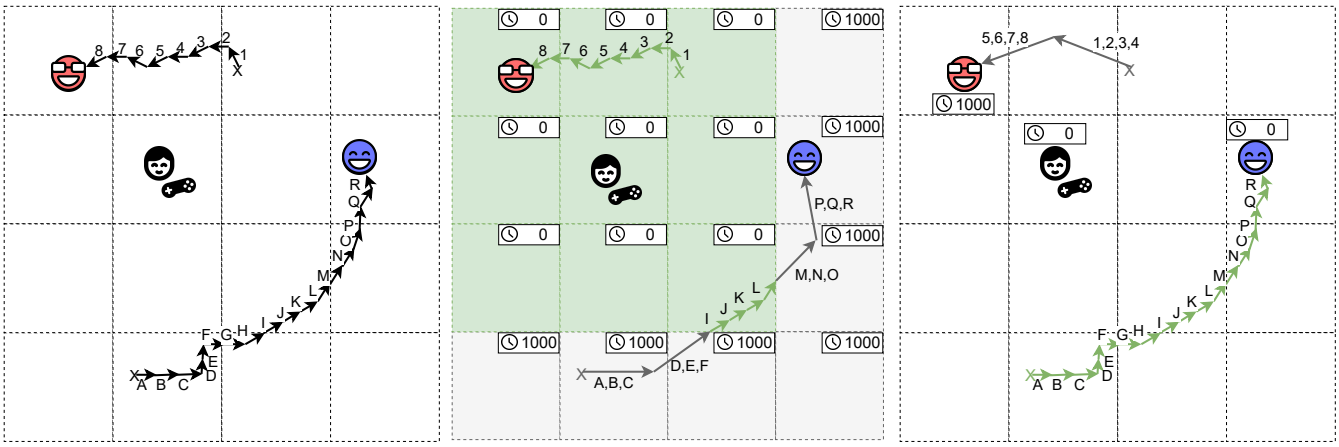
**Design process and alternatives:** We reached this design through a long-term, iterative design process following the principles of the AtLarge design vision [27]. We ideated repeatedly and considered a large pool of designs, which we have analyzed for both their *level of innovation*, that is, that the designs are new solutions for the problem, and *pragmatism*, that is, that they can be implemented in the conditions where they are to be used. This principled approach to design resulted in many design alternatives. We summarize here two of the design alternatives we have considered in more detail.

Considering innovative designs, we also analyzed enforcing strict, instead of optimistic, consistency bounds. However, this would effectively limit the performance of the system to that of the weakest node. This is undesirable because the game server, typically hosted in a datacenter, can have significantly more resources than a client running on a (mobile, or otherwise constrained) user device.

Considering pragmatic designs, we considered letting each client independently configure their Dyconit bounds, to match the design of TACT [28]. However, because the game server has access to the global state, we decided to simplify our design by managing the dyconits using a centralized policy and avoiding unnecessary coordination between the server and clients.

### *B. Dyconit Policies (Requirements **R4**, **R5**)*

The Dyconit system can reduce resource usage by allowing inconsistency. However, applications are likely to have esoteric requirements in terms of the kind and magnitude of inconsistency that is acceptable to its users, depending on the MVE (static requirements) and on current workload (dynamic requirements). By cleanly separating mechanism from policy, the Dyconits system supports a variety of approaches that tolerate inconsistency. Here we describe how the Dyconit policies work, introduce five policies (see also Table I), and show how they allow the system to use state-of-the-art interest-management approaches that reduce bandwidth consumption.




(a) No Dyconits. (b) Dyconit AOI policy. (c) Dyconit IS policy.

Fig. 4: Visual overview of how Dyconits can effect state synchronization. A player ( ) observes the movement of two avatars ( , ) through a virtual world consisting of contiguous chunks (i.e., areas, ). Every arrow (→) represents a state update received by the player. Every dyconit is shown together with the player's staleness bound (e.g., 1000 ms as 1000). The AOI policy creates a dyconit for every chunk ( ), whereas the IS policy creates a dyconit for every player.

Each Dyconit policy can affect the operation of Dyconits in three areas. First, the policy determines how the global state is partitioned across dyconits. Second, the policy determines the weight of individual state-updates; accumulation of weight beyond the bounds leads to synchronization. Third, the policy can re-configure dyconit bounds for each player, based on player state and the dynamic system workload.

We now explain, through a series of examples, how policies can affect system behavior. Table I (on the previous page) summarizes five Dyconit policies, which we explain in turn. The first three policies are not new, but the latter two (**ISN**, **ZERO/IS**) are, because they respectively create a novel combination of policy and consistency dimension, and consider system load. Overall, these policies are valuable because they can be more flexible and fine-grained than previous policies.

The **ZERO policy** is a null-policy that emulates a system without Dyconits. First, the policy creates a single partition for the entire game-world, which means all state-updates are assigned to this one partition. Second, the policy assigns a weight of one to every state update. Third, sets a bound of zero for both staleness and numerical error for each client. Because every message exceeds the consistency bound of each client, state updates are synchronized immediately to everyone.

The following three policies are based on interest management approaches in virtual worlds. These policies dynamically update consistency bounds based on player state. A visual representation of their behavior is shown is Figure 4.

The **AOI policy** is based on the concept of AoI from virtual worlds. The AoI intuition is that users are typically more interested in changes to nearby objects. Our AOI policy achieves this by first partitioning the system state into areas corresponding to geographical locations in the virtual world. Second, it assigns weights to state updates based on the magnitude of their change; for object-location updates, the magnitude is equal to the displacement. Third, the policy sets bounds for each user depending on the location of their avatar in the virtual world: zero staleness for adjacent areas and a bound of one second for areas further away. The higher bound of one second is based on a value reported in related work [14].

The **IS policy** is based on the concept of *interest sets* [14]. The intuition is that humans can only focus on a limited set of objects, so objects outside immediate focus allow more inconsistency. The IS policy realizes this behavior by creating a partition for each user. Second, it assigns a weight to each state update by using the magnitude of the change, similar to AOI policy. Third, the policy sets bounds of zero staleness for avatars in a user's interest set, and of one second for all others.

The **ISN policy** adapts the IS policy to use a numerical bound, instead of a staleness bound, for avatars outside a user's interest set. This policy sets a numerical bound of 0.16 (see Table I) for avatars outside the player's interest set. Because the weight of a location update message is equal to its displacement, a bound of 0.16 allows a location inconsistency slightly larger than the maximum distance traveled in a single tick, preventing synchronizing small avatar movements.

Finally, the **ZERO/IS policy** updates dynamically the consistency bounds, based on both player state and system load. The policy monitors resource usage and initially behaves like ZERO. If critical resources (e.g., CPU, network bandwidth) become scarce, it trades off consistency for reduced resource usage by behaving like the IS policy, repartitioning the system state across users and setting a staleness bound of 1 s for all avatars outside a player's interest set.

### C. Adaptability, Extensibility, and Generality of Dyconits

In our vision on future distributed (eco)systems [29, challenges C3 and C6], we have challenged the community to build systems that adapt by "changing service level objectives

(SLOs) upon detecting a resource overload or a straggling task". Approaches should consider fine-grained non-functional requirements and change SLOs both spatially (e.g., for small game object) and temporally (e.g., in short time-period). We have shown in Section III-B how, through an extensible approach to policies, Dyconits adapt to real-world conditions and thus can support this vision conceptually. Section V shows evidence this support can also exist in practice.

The system is made more extensible and practical by using only server-side modifications. This also gives sufficient benefits, performance-wise. Dyconits focus on limiting traffic. As shown by a state-of-the-art evaluation of Minecraft-like games [7, §4.4], the server-bound network traffic in Minecraft-like games is orders of magnitude smaller than the client-bound network traffic.

We also conjecture that *the Dyconits mechanism and system are general across many domains*, but the policies proposed here are tied to the gaming domain. Part of our conjecture, Dyconits should generalize across games such as First-Person Shooters and Massively Multiplayer Online Role Playing Games, and even non-gaming applications where different users have highly heterogeneous consistency requirements, such as virtual concerts and interactive classrooms. We leave proving this conjecture to future work.

## IV. Realization of a Dyconit System in MVEs

In this section we describe how we realize the Dyconit system in a real-world MVE. We consider as technological base the open-source, Java-based, Minecraft-like MVE Glowstone [19], and analyze how to add to it the Dyconits system. Realization precedes the actual implementation of a prototype; conceptually, realization addresses a level lower than the design in the previous section, but higher than software engineering decisions.

### A. Quantifying Inconsistency in Practice

The Dyconit system quantifies inconsistency along two dimensions, *staleness* and *numerical error*. The (combined) usage of these inconsistency dimensions was first introduced in TACT [18], but still needs to be adapted for MVEs. Staleness is defined simply as the amount of time elapsed since messages where last synchronized.

Numerical error is more complex, but allows more semantically meaningful ways of bounding inconsistency in MVEs. To quantify numerical error, all messages are first assigned a global weight. This weight objectively indicates the size of the state update. For example, in an online game we set the weight of an avatar's position update to equal the displacement. Importantly, the weight of the message does not indicate the importance of the update for individual clients. Using the weight of each update, we define the numerical error as the sum of the weights of all updates that have not yet been forwarded to the client. When to forward these updates is determined by the Dyconit policy.

### B. Dyconit Policies in Practice

Each Dyconit policy partitions the game state, weighs state updates, and configures dyconit bounds for all clients. This section shows how these tasks can be completed efficiently.

The MVE developer controls the behavior of the Dyconit system through the Dyconit policy. These policies allow the Dyconit system to perform interest management (e.g., the AOI and IS policies), to make trade-offs between consistency and performance (e.g., the ZERO/IS policy), etc. To give policies access to the MVE's internal state, we define an interface for Dyconit policies. This allows the MVE developer to implement their own policy, and has two main additional benefits. First, it facilitates access for the policy to implementation-specific game state, enabling policies such as AOI and IS, which need this data to operate. Second, it offers control and thus more reasons for MVE developers to adopt Dyconits.

The Dyconit policies need to partition the game state; however, this may have undesirable performance drawbacks for the simulator, which benefits from data locality. To let policies partition the game-state without affecting data-locality, we create *logical* partitions that do not affect the physical layout of the state's underlying data. Additionally, policies create partitions *lazily*. To do so, the policy inspects outgoing state-updates to determine which state they modify, and to which partition this state belongs. Only if the update affects a partition for which no dyconit currently exists, one is created. Similarly, state updates that remove objects from the game (e.g., an area of the world is unloaded because it is unused) can trigger the removal of a dyconit.

Because virtual worlds in MVEs are large and unbounded in size, players are likely to have no interest in the majority of the game state partitions (i.e., dyconits). This makes explicitly setting bounds for each player-dyconit pair inefficient. We address this issue through dyconit *membership*: a client becomes a member of a dyconit when their bounds are explicitly set by the active policy. For all non-members, their bounds are assumed to be infinite (i.e., no consistency). The Dyconit policy can assign membership to clients, per dyconit, either periodically or based on specific events. Because player behavior is simulated every tick, player state is likely to be inspected every game tick. Changing membership behavior occurs in this step at relatively little cost, although the final complexity of the operation is determined by the complexity of the Dyconit policy. Similar to assigning state-updates to dyconits, membership changes can also lazily initialize dyconits: if a player should be a member of a non-existing dyconit, this dyconit is created dynamically by the system.

### C. Interest Management in Practice

This section discusses how to realize, in a real-world MVE, the interest management techniques used by the Dyconit policies presented in Section III-B.

The AOI policy partitions the game-state based on geographical areas. In MVEs, players explore a procedurally generated, voxel-based virtual world. This world is typically endless and generated on-demand in fixed-sized *chunks*. This

TABLE II: Overview of experiments.

| Policy | | Workload | | | Parameters | |
|---|---|---|---|---|---|---|
| Section | Focus | Type | Num. Players | World | Policy (§III-B) | Duration |
| V-B | Network Resource Usage | Fixed | 200 | flat | ZERO, IS, ISN | 5m |
| V-C | System Scalability | Increasing | 1/s | flat | ZERO, AOI, IS, ISN | <1h |
| V-D | Game Inconsistency | Fixed | 200 | flat | ZERO, IS, ISN | 5m |
| V-E | Dynamic Consistency-Performance Trade-off | Spike | 300-400 | flat | ZERO, IS | 6m |

removes the need for the AOI policy to run its own world-partitioning algorithms—it creates a dyconit for every chunk. To associate updates with the correct dyconit, the system inspects update messages. If the update modifies terrain, the policy assigns it to the dyconit representing the chunk that has been modified. If the update modifies an object, the policy accesses the game state to look up the object's current location; it then assigns the update to the chunk's dyconit.

The IS and ISN policies partition the game-state per player. Every state-update concerning a player action (e.g., avatar movement) is associated with the dyconit matching the player. However, not all state-updates are directly associated with a player, e.g., the movement of an Non-Playable Character (NPC). For all such updates, the policy creates one additional, *catch all* partition. To make sure clients do not miss such updates, all clients are a member of this partition using an inconsistency bound of zero.

The ZERO/IS policy has two modes of operation, whose change is triggered by the system load. Initially, this policy behaves like the ZERO policy: it creates a single dyconit that includes all clients as members, with an inconsistency bound set to zero. If the tick duration gets close to 50 ms (excluding outliers), the system is close to overload so the policy re-partitions the game state and re-assigns membership according to the IS policy. When the workload drops, the reverse occurs.

## V. Real-World Experiments

We evaluate in this section the Dyconits system, through *real-world experiments*. We build a *prototype* starting from the codebase of Glowstone, a popular open-source and full-scale implementation of Minecraft. We also implement the five policies introduced in Section III-B; policy ZERO allows us to compare the game with and without the use of Dyconits. We release the complete project, as *Opencraft*, on GitHub [20].

We conduct comprehensive and realistic experiments, using the Yardstick MVE benchmark [7] and the settings detailed in Section V-A, to assess resource usage, system scalability, in-game inconsistency, and the dynamic consistency-performance trade-off. Table II summarizes our experiments, from which we derive the following **M**ain **F**indings:

**MF1** Dyconits can reduce the game mean bandwidth usage by 49% when allowing low inconsistency, and by 85% when allowing large, but tolerable, inconsistency.

**MF2** Dyconits significantly improve the scalability of MVEs.

**MF3** Dyconits improve MVE scalability while bounding inconsistency.

**MF4** Dyconits improve the robustness of MVEs by temporarily allowing large inconsistency during workload bursts.

### A. Experiment Setup

Here we describe the experiment setup. A summary of the *workloads* is available in Table II. For an MVE, the workload consists of two parts: the world and the players. The workloads all use a *flat* world, i.e., a world consisting of an infinite flat plane, but vary the number of players. The *increasing* workload connects 5 new players every 5 seconds. The workload is synthetic, but useful for scalability experiments, as it enables observing the maximum number of supported players. The *fixed* workload connects a predetermined number of players: 25 new players every 5 seconds until the set number of players is reached. The *spike* first connects a base number of players. Then, after 180 seconds, it increases the number of players to create a peak. The extra players remain connected for one minute before disconnecting. The players are connected in the same way as in the fixed workload. The players move in-world using Yardstick's behavior model.

We run our experiments on DAS-5, a distributed multi-cluster for academic and educational use [30]. For all experiments, we use one machine to run the Opencraft server, and Yardstick uses one machine per 50 emulated players. Each machine is equipped with a dual 8-core 2.4 GHz CPU, 64 GiB of memory, and an InfiniBand network with a maximum throughput of 48 Gbps. This setup is comparable to the high-bandwidth environments available in data centers, which are used in industry to host MVEs as a service [11].

### B. Reduction in Network Resource Usage (Leads to *MF1*)

The first experiment reports resource usage; Figure 5 on the next page summarizes the results. The results show that allowing low inconsistency can reduce mean bandwidth usage by 49%, while allowing large but still acceptable inconsistency can reduce the (*arithmetic*) mean of bandwidth usage by 85%.

The top-most plot in Figure 5 shows the number of messages sent by the MVE server per second. With more than 800,000 messages per second, the ZERO policy sends the most messages, by far. Because the policy behaves like a traditional game without Dyconits, it allows no inconsistency and immediately synchronizes all state updates with every player. This static behavior also explains its low variance. For a tick rate of 20 Hz and 200 players, we expect exactly 800,000 messages per second. The mean being slightly above this value is likely caused by additional state updates, albeit less frequent, such as those describing the layout of the terrain.

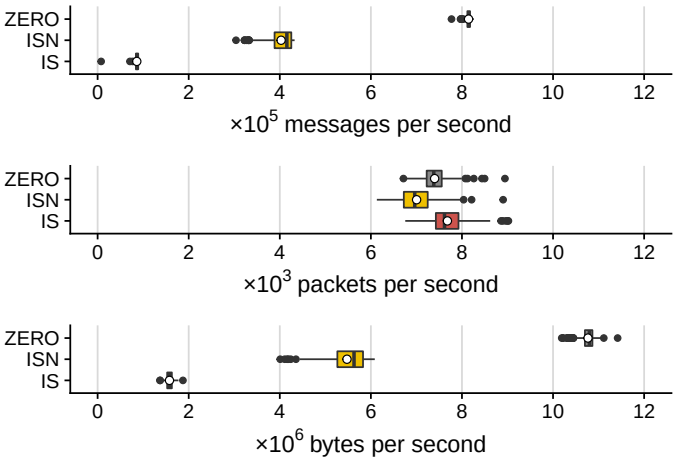



Fig. 5: Network usage of the same game, for different Dyconit policies. Lower values (more to the left, horizontally) are better. White dots show the arithmetic mean.

The ISN policy also behaves as expected. Allowing most players to be inconsistent by a single step results in synchronizing every-other step, resulting in a reduction of messaging, by half. Although players keep an inconsistency bound of zero with other players in their interest set, the number of players in the a player's interest set is sufficiently small (five) to not impact significantly the number of messages sent per second.

Similarly to the ISN policy, the IS policy sets a bound of zero on avatars in the player's interest set. However, it allows more inconsistency (staleness bound at 1 s) for avatars outside the interest set. In our experiments, this results in an 89% reduction in the mean number of messages sent per second.

All policies send slightly more messages than predicted. Our prediction only includes estimates for the number of entity-movement messages, whereas the measurements include all types of messages. This result confirms an earlier result in the community, which shows that the majority of messages sent in MVEs are entity-position updates [7].

Only minor differences appear in the statistical properties (i.e., mean, median, narrow IQR) of the number of packets sent per second, for the three policies. This seems to be caused by the underlying network stack, which concatenates multiple messages into a single packet because the majority of messages is sent in bursts (once per game tick).

The strong reduction in number of messages per second, combined with the relatively stable number of packets per second, results in a strong reduction of bandwidth usage (bytes per second). The number of bytes per second shows the same trend as the number of messages per second, but less strongly. This is caused by the game server sending chunk-related (world-data) messages, which are much less frequent than avatar positions, but much larger in size, and are not affected by our Dyconit policies.

### C. Increasing Maximum Number of Players (*MF2*)

Dyconits can significantly improve the maximum number of players supported by an MVE. Figure 6 shows the results of the *scalability experiment*. The runtime differs for each policy

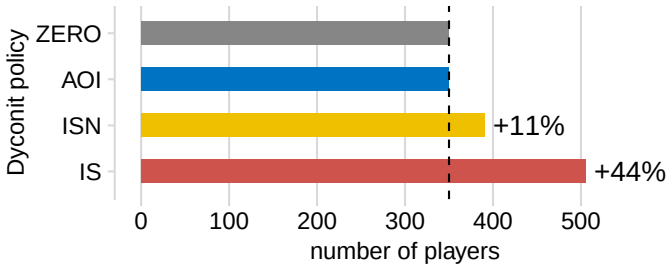



Fig. 6: Maximum number of players for a variety of Dyconit policies. Policy ZERO represents the baseline, i.e., without the use of Dyconits, allowing zero inconsistency.

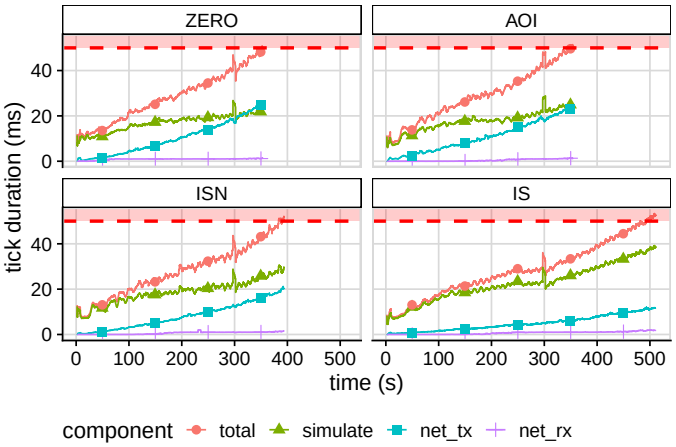



Fig. 7: Breakdown of tick duration for an increasing number of players and a variety of Dyconit policies. Policy ZERO represents the game without the use of Dyconits.

because the experiment runs an increasingly heavy workload until the game becomes overloaded. Without Dyconits, the game scales to 350 players. Using Dyconits can increase the scalability of the game, depending on the active policy.

The AOI policy fails to increase scalability. In our analysis, we found that this is likely due to the close proximity of the emulated players. The AOI policy sets increasingly high inconsistency bounds for areas further away from the player. Because all players are close together, their inconsistency bounds are effectively zero. The ISN policy increases scalability from 350 to 390 players (+11%), which means the large reduction in bandwidth usage (see **MF1**) does not translate to large scalability improvement. By allowing larger inconsistency, the IS policy can support up to 505 players (+44%).

To better understand how the decreased consistency created by the Dyconit policies affects the game's scalability, we analyze the time taken out of the tick duration by each component, including networking. Figure 7 breaks down the components of the tick duration, over time, as the number of players increases. The top (orange) curve shows the total time spent in the game tick. The other curves show how much time is spent on each component. `Simulate` combines the time needed to simulate the players and the world; `net_rx` and `net_tx` show the time spent on receiving and sending state updates, respectively. The majority of the time is spent on simulating the virtual world and on sending state updates.

In Figure 7, all curves follow the constantly increasing number of players. Despite the tick duration increasing slowly over time (i.e., sub-exponentially), the scalability of the game is limited because the game becomes unplayable when the tick duration exceeds 50 ms (dashed-red, horizontal line). When this happens, the game is no longer able to update its state at 20 Hz, slowing down time in the simulation and causing undeterministic behavior. The experiment connects five players every five seconds until the game becomes unplayable in this way, at which point the game is stopped.

When not using Dyconits (plot ZERO), the time spent on sending state-updates exceeds the time spent on simulation for large numbers of players, and the game becomes overloaded. Both ISN and IS allow inconsistency between players, reducing time spent on synchronizing state (`net_tx`), which increases the maximum number of players.

All plots in Figure 7 show a spike in workload after 300 seconds. We found the cause of this behavior to be the game's automatic save function that triggers every five minutes. Saving removes unused area of the world from memory, and writes the entire game state to disk. These operations take additional time to complete, creating a spike in the tick duration.

Finally, when using Dyconits (AOI, IS, and ISN), the `simulate` part of the tick starts increasing more rapidly after the world has saved (300 seconds), increasing the total tick duration and limiting scalability. The reason for this behavior is unknown, but negatively effects the scalability improvements gained by using Dyconits.

### D. Improve Performance, Bound Inconsistency (*MF3*)

Dyconits can successfully improve game performance while simultaneously bounding inconsistency. Figure 8 shows the MVE inconsistency and resource consumption for several Dyconit policies. The figure reports the *system-wide* inconsistency, calculated as the sum of the inconsistencies between each player and the global state. The top two figures respectively show the numerical error and staleness for each policy.

The top plot in Figure 8 shows that the ISN policy successfully limits inconsistency. The policy permits a numerical error of 0.16 units (see Table I) on all avatars outside a player's interest set. With 200 players and an interest set of 5, the upper bound on the system-wide inconsistency is $0.16 \times 200 \times 195 = 6,240$. We observe that the ISN policy stays well below this value. Because the policy synchronizes every other step, and not all players start moving simultaneously, it is unlikely that a player is inconsistent with all other players. Using the IS policy, which bounds staleness, results in a global numerical error ranging from roughly 22,000 to over 60,000. Without dead-reckoning, a global numerical error of 60,000 for 200 players means that, on average, each player sees every other player's avatar 1.5 meters (units) from where they actually are.

The middle plot in Figure 8 shows that the IS policy successfully limits staleness. The policy permits staleness of 1 second for avatars outside a player's interest set. With 200

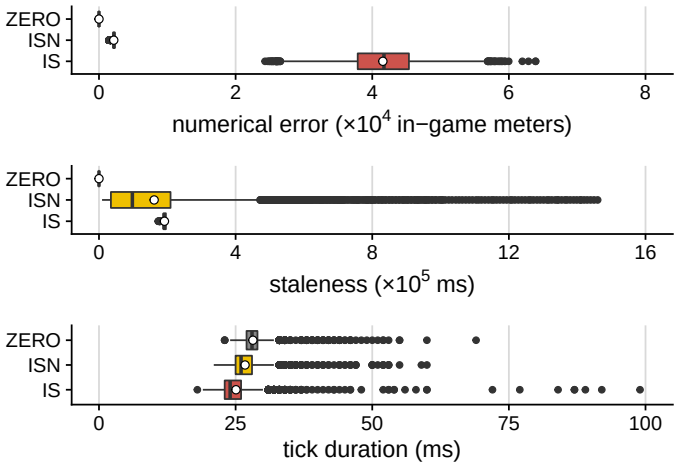



Fig. 8: Trade-off between consistency and performance for varying Dyconit policies under the Fixed-200 workload. Lower values (more to the left, horizontally) are better. White dots show the arithmetic mean.

players, this creates a global consistency bound of 1,000 × 200 = 200,000 ms. The policy reaches, but does not exceed, that bound. The ISN policy, which uses a numerical error bound, far exceeds the consistency bound of the IS policy. Its outliers reach an inconsistency of ≈1,500,000 ms, which means, on average, here players have not received some state updates for 1,500,000 / 200 = 7,500 ms. Further analysis reveals that these outliers arise from players outside the interest set who move very little during an extended period of time; their accumulated change does not exceed the numerical bound required to trigger an update, but staleness increases.

The bottom plot in Figure 8 shows the Dyconit policies reduce the tick duration, but only by a small amount. Concerning the resource overhead of running the policies, the CPU usage for these policies (not shown) follows the same trend. Because time spent on sending state updates is relatively small when using a workload of only 200 players, Dyconits do not reduce much the resource usage. The bottom plot in Figure 8 also shows large numbers of outliers for the tick duration. During further analysis we found these outliers are present throughout the duration of the experiment, and occur at a roughly fixed interval. Due to this observation, combined with the fact that the outliers are present across all Dyconit policies, we conjecture they are not caused by the Dyconit system, but by periodic system-level events such as JVM garbage collection.

### E. Dynamic Consistency/Performance Trade-off (*MF4*)

Although the previous experiments show that Dyconits can improve performance by allowing bounded inconsistency, the performance under lighter workloads may be sufficiently good without introducing inconsistency. The previous experiment is an example of such a case: under a workload of 200 players, the MVE can sustain the required 20 Hz tick frequency because the tick duration is less than 50 ms. In this case,

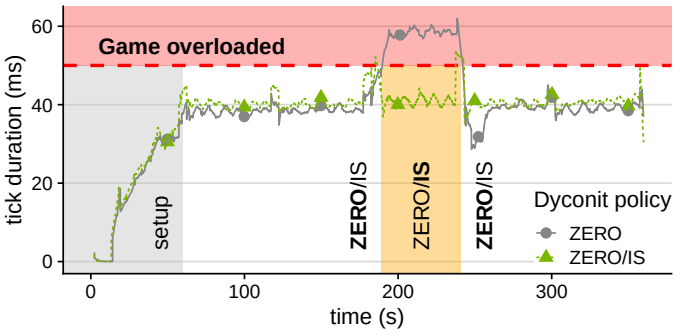



Fig. 9: Tick duration over time during the Spike workload. Policy ZERO represents the game without the use of Dyconits.

introducing inconsistency can only marginally improve the players experience, and may even decrease it.

In this experiment we evaluate the ZERO/IS Dyconit policy. This policy monitors the tick duration over time and initially avoids inconsistency by synchronizing state updates with clients immediately. However, when the tick duration approaches 50 ms, to improve scalability it starts behaving similar to the interest set-based IS policy. When the number of players drop below the number that triggered using interest sets, the policy will try to stop using Dyconits.

Figure 9 shows the results of the experiment. The first 60 seconds of the experiment are the setup phase in which players join the game. Every 60 seconds, the game performs an automatic save, creating a peak in the tick duration. After 180 seconds, the number of players increases from 300 to 400. When using the ZERO policy, this causes the tick duration to exceed 50 ms, which in turn causes the game to drop below the required 20 Hz update frequency. In contrast, the ZERO/IS policy briefly lets the tick duration exceed 50 ms, then enables interest sets and brings the tick duration back below 50 ms.

After 240 seconds, 100 players leave the game and the game performs an automatic save. Despite the additional workload caused by the save, the ZERO/IS policy disables interest sets and maintains a tick duration below 50 ms.

## VI. Main Threats to Validity

Applying Dyconits to a professionally engineered game can result in higher levels of performance improvement. In games, significant performance and scalability improvements can be obtained through advanced engineering. Because we built Dyconits into Opencraft, an open-source game, we conjecture that Dyconits can obtain a larger performance improvement when applied to a professionally engineered system.

Our experiments use the Yardstick MVE benchmark to emulate players [7]. Unfortunately, the benchmark uses a player behavior model based on players in the game Second Life, which is an MVE but not Minecraft-like. Thus, the mobility model we use may not capture all elements of Minecraft-like player behavior. However, there currently exists no model in the community for mobility in Minecraft-like games.

Similarly, as the Yardstick experiments indicate [7, Table 1], the workload of Minecraft-like games can depend on various game-world features, including complexity and view distance. Exploring these features can be done with Yardstick, given enough time and material resources, and remains outside the scope of this work.

## VII. Related Work

We survey in this section work related to Dyconits. In contrast with all the existing approaches, ours is the first to focus on MVEs—we provide new mechanisms and dynamic policies that bound inconsistency and trade it off for scalability.

We base Dyconits on the *continuous consistency model* introduced in TACT [18]. TACT is designed for distributed databases. Our dyconit is similar to the TACT conit, but, to make it useful for the highly dynamic and write-intensive MVEs, our model allows modifying the consistency bounds frequently at runtime, and prevent reducing availability while synchronizing state by bounding inconsistency optimistically.

*Predictive treaties* [31] are a mechanism to improve performance in distributed systems by reducing synchronization frequency. Predictive treaties use the observation that, when state changes are predictable, the system can formulate predicates about the system state that remain valid for a certain amount of time and can be used instead of retrieving an exact value. Whereas predictive treaties are designed for database systems and improve performance by avoiding network latency, Dyconits are designed for online games in which players are often unpredictable and game worlds non-linear, and scalability is limited primarily by bandwidth.

Multiple approaches exist for increasing game scalability by allowing inconsistency. *Interest management approaches* [16] omit—or reduces the frequency of—state updates that are further removed from the player's avatar. We compare here with two such approaches, Colyseus and Donnybrook. Colyseus [21] is a distributed middleware for multiplayer games that replicates game objects (e.g., avatars, items) across nodes, distributing the computation needed to update these objects every tick. Colyseus optimistically bounds inconsistency to a staleness of 100 ms, using AoI. In contrast, Dyconits further use numerical error to prevent large inconsistencies, and supports other interest management techniques such as interest sets. Finally, Colyseus does not limit bandwidth usage.

Donnybrook [14] introduces interest sets per player and limits update-frequency beyond sets. Donnybrook uses döppelgangers, AI-based movement extrapolation (effectively, predictive treaties), to reduce inconsistency. In contrast, Dyconits also bound the numerical inconsistency.

The Mobihoc middleware [22] introduces the Vector-Field consistency model for games in ad-hoc networks. Mobihoc is also based on TACT, and allows developers to define time, sequence, and value error-bounds on location-based *pivots*. In contrast, Dyconits supports dynamically changing consistency bounds, and players joining or leaving ongoing games.

DynFilter [32] is a publish/subscribe middleware for online games. DynFilter partitions the virtual world and creates a high-frequency and a low-frequency topic for each partition. Players subscribed to the high-frequency topic receive all updates from that partition immediately, while subscribers to

the low-frequency topic may not. Like Dyconits, DynFilter determines dynamically how many updates to omit, based on the workload. In contrast, Dyconits bounds the resulting inconsistency between players, and is compatible with non-distance-based approaches such as interest sets.

## VIII. Conclusion and Future Work

Minecraft-like games, and more generally MVEs, currently scale to only a few hundred players in self-hosted games and to as low as only ten players in cloud-hosted services. This contrasts sharply with their popularity, e.g., Minecraft alone has over 130 million active users. To address this problem, we introduce the Dyconits system, the first MVE middleware that keeps inconsistency bounded, dynamically.

We design our Dyconits system around a data-centric, continuous consistency model adapted to MVEs, which it uses to quantify and optimistically bound the inconsistency experienced by players. Through the creation and dynamic management of consistency units (the *dyconits*), our design allows dynamically-changing bounds on arbitrarily small parts of the game state. The system separates consistency mechanism from policy; we propose in this work four policies that leverage dyconits in conjunction with various other state-of-the-art interest-management mechanisms. We realize the Dyconit system and implement it in Glowstone, a popular open-source Minecraft-like MVE, demonstrating Dyconits can be applied in practice. We conduct comprehensive real-world experiments: compared with the current technology, Dyconits can reduce network bandwidth consumption by up to 85%, improve scalability by up to 40%, and trades off efficiently and dynamically consistency for performance.

In future work, we aim to evaluate the efficacy of Dyconits for MVEs and beyond. For MVEs, managing the inconsistency caused by non-player entities is a relatively open area of research. We conjectured in Section III-C the Dyconit mechanism is general, but requires domain-specific policies; proving this is left for future work.

## Acknowledgments

This work is supported by NWO grants MagnaData and OffSense, and by structural funds from VU Amsterdam.

## References

[1] B. Gilbert, "Video-game industry revenues exceed sports, film combined in 2020 - Business Insider," Dec 2020. [Online]. Available: http://bit.ly/GamesExceedFilm

[2] "'Minecraft' Tops 131 Million Monthly Active Users," Jan 2021, [Online; accessed 8. Jan. 2021]. [Online]. Available: http://bit.ly/Minecraft131Million

[3] M. Panzarino, "Apple pushes the reset button on the Mac Pro," apr 2017. [Online]. Available: https://bit.ly/MacOSUsers

[4] "Minecraft Realms for Java," Nov 2020, [Online; accessed 12. Jan. 2021]. [Online]. Available: http://bit.ly/MinecraftService

[5] "Steam Search," Jan 2021, [Online; accessed 12. Jan. 2021]. [Online]. Available: http://bit.ly/steam-mves

[6] "Mods - Minecraft - CurseForge," Jan 2021, [Online; accessed 12. Jan. 2021]. [Online]. Available: http://bit.ly/ModsForMinecraft

[7] J. van der Sar, J. Donkervliet, and A. Iosup, "Yardstick: A Benchmark for Minecraft-like Services," in *ICPE*, 2019, pp. 243–253.

[8] J. Donkervliet, A. Trivedi, and A. Iosup, "Towards Supporting Millions of Users in Modifiable Virtual Environments by Redesigning Minecraft-Like Games as Serverless Systems," in *HotCloud*, 2020.

[9] A. Natividad, "How Greenpeace Used Minecraft to Stop Illegal Logging in Europe's Last Lowland Primeval Forest," jan 2018. [Online]. Available: https://bit.ly/MinecraftGreenpeace

[10] M. Claypool and K. T. Claypool, "Latency and player actions in online games," *Commun. ACM*, vol. 49, no. 11, pp. 40–45, 2006.

[11] Mojang, "Minecraft Realms." [Online]. Available: https://www.minecraft.net/en-us/realms-plus

[12] R. A. Bartle, *Designing virtual worlds*. New Riders, 2004.

[13] M. R. Macedonia, M. Zyda, D. R. Pratt, P. T. Barham, and S. Zeswitz, "NPSNET: A Network Software Architecture For Large Scale Virtual Environments," *Presence Teleoperators Virtual Environ.*, vol. 3, no. 4, pp. 265–287, 1994.

[14] A. R. Bharambe, J. R. Douceur, J. R. Lorch, T. Moscibroda, J. Pang, S. Seshan, and X. Zhuang, "Donnybrook: enabling large-scale, high-speed, peer-to-peer games," in *SIGCOMM*, 2008, pp. 389–400.

[15] J.-S. Boulanger, J. Kienzle, and C. Verbrugge, "Comparing interest management algorithms for massively multiplayer games," in *NetGames*, 2006, p. 6.

[16] E. S. Liu and G. K. Theodoropoulos, "Interest management for distributed virtual environments: A survey," *ACM Comput. Surv.*, vol. 46, no. 4, pp. 51:1—-51:42, 2014.

[17] P. Viotti and M. Vukolic, "Consistency in Non-Transactional Distributed Storage Systems," *ACM Comput. Surv.*, vol. 49, no. 1, pp. 19:1—-19:34, 2016.

[18] H. Yu and A. Vahdat, "Design and evaluation of a conit-based continuous consistency model for replicated services," *ACM Trans. Comput. Syst.*, vol. 20, no. 3, pp. 239–282, 2002.

[19] "Glowstone Project," Jan 2021, [Online; accessed 12. Jan. 2021]. [Online]. Available: https://glowstone.net

[20] atlarge research, "opencraft-dyconits," Jan 2021, [Online; accessed 8. Jan. 2021]. [Online]. Available: https://github.com/atlarge-research/opencraft-dyconits

[21] A. R. Bharambe, J. Pang, and S. Seshan, "Colyseus: A Distributed Architecture for Online Multiplayer Games," in *NSDI*, 2006.

[22] N. Santos, L. Veiga, and P. Ferreira, "Vector-Field Consistency for Ad-Hoc Gaming," in *Middleware*, vol. 4834, 2007, pp. 80–100.

[23] V. Nae, A. Iosup, and R. Prodan, "Dynamic Resource Provisioning in Massively Multiplayer Online Games," *TPDS*, vol. 22, no. 3, pp. 380–395, 2011.

[24] K.-T. Chen, C.-C. Tu, and W.-C. Xiao, "OneClick: A Framework for Measuring Network Quality of Experience," in *INFOCOM*, 2009, pp. 702–710.

[25] P. Chen and M. E. Zarki, "Perceptual view inconsistency: An objective evaluation framework for online game quality of experience (QoE)," in *NetGames*, 2011, pp. 1–6.

[26] S. Shen, S.-Y. Hu, A. Iosup, and D. H. J. Epema, "Area of Simulation: Mechanism and Architecture for Multi-Avatar Virtual Environments," *TOMM*, vol. 12, no. 1, pp. 8:1—-8:24, 2015.

[27] A. Iosup, L. Versluis, A. Trivedi, E. V. Eyk, L. Toader, V. van Beek, G. Frascaria, A. Musaafir, and S. Talluri, "The AtLarge Vision on the Design of Distributed Systems and Ecosystems," in *ICDCS*, 2019, pp. 1765–1776.

[28] H. Y. Vahdat and Amin, "Combining Generality and Practicality in a Conit-Based Continuous Consistency Model for Wide-Area Replication," *ICDCS'01*, pp. 429–438, 2001.

[29] A. Iosup, A. Uta, L. Versluis, G. Andreadis, E. V. Eyk, T. Hegeman, S. Talluri, V. van Beek, and L. Toader, "Massivizing Computer Systems: A Vision to Understand, Design, and Engineer Computer Ecosystems Through and Beyond Modern Distributed Systems," in *ICDCS*, 2018, pp. 1224–1237.

[30] H. E. Bal, D. H. J. Epema, C. de Laat, R. van Nieuwpoort, J. W. Romein, F. J. Seinstra, C. Snoek, and H. A. G. Wijshoff, "A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term," *IEEE Computer*, vol. 49, no. 5, pp. 54–63, 2016.

[31] T. Magrino, J. Liu, N. Foster, J. Gehrke, and A. C. Myers, "Efficient, Consistent Distributed Computation with Predictive Treaties," in *EuroSys*, 2019, pp. 36:1—-36:16.

[32] J. Gascon-Samson, J. Kienzle, and B. Kemme, "DynFilter: Limiting bandwidth of online games using adaptive pub/sub message filtering," in *NetGames*, 2015, pp. 1–6.