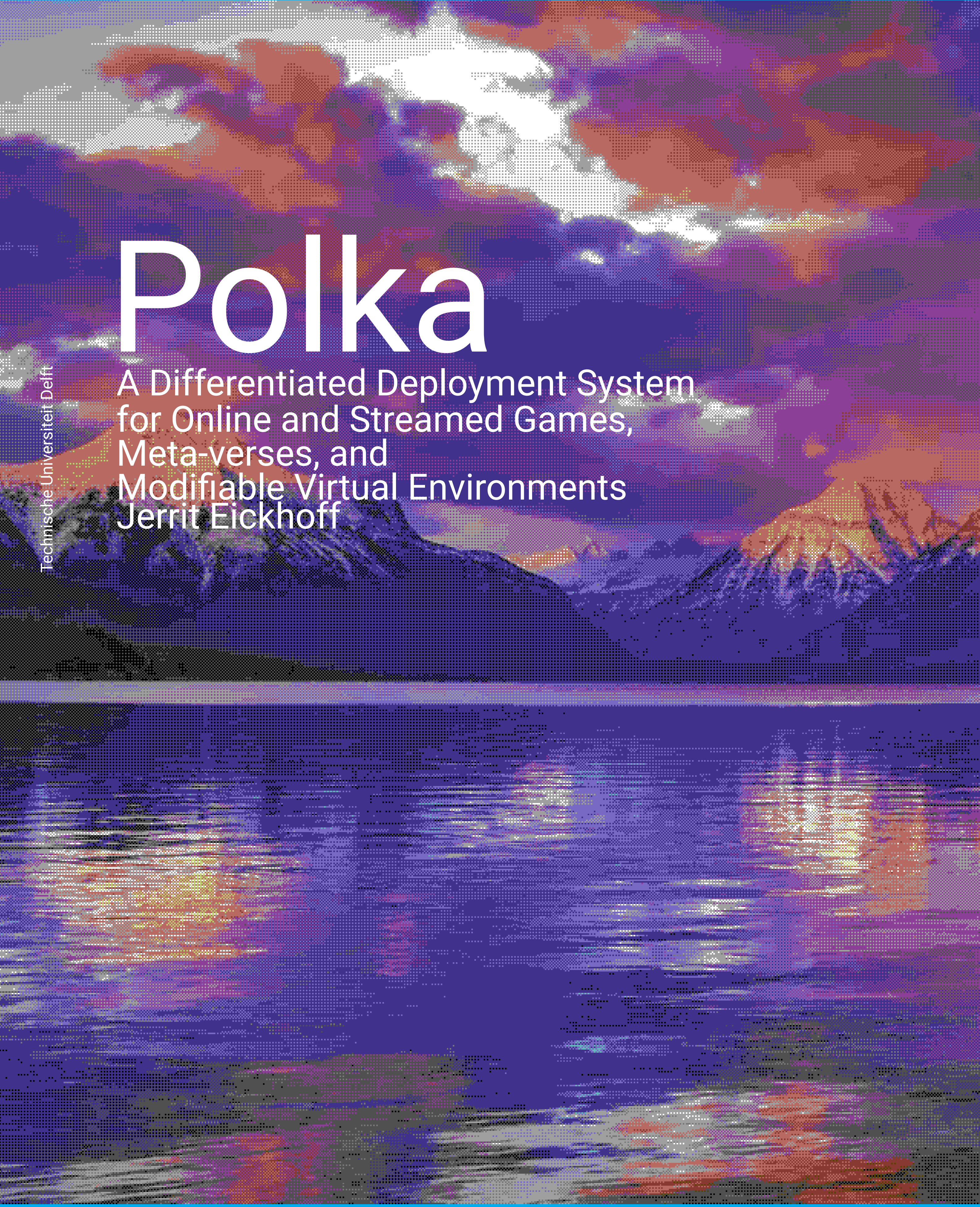


Polka

A Differentiated Deployment System for Online and Streamed Games, Meta-verses, and Modifiable Virtual Environments

Jerrit Eickhoff

Technische Universiteit Delft

Polka

A Differentiated Deployment System for Online and Streamed Games, Meta-verses, and Modifiable Virtual Environments

by

Jerrit Eickhoff

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on March 15, 2023, at 10:00 AM.

Student number:	5487676	
Project duration:	November 1st, 2022 – March 15, 2024	
Thesis committee:	Prof. dr. ir. F.A. Kuipers,	TU Delft, chair
	Dr. A. Katsifodimos,	TU Delft, committee member
	Prof. dr. ir. A. Iosup,	VU Amsterdam and TU Delft, thesis advisor
	Ir. J. Donkervliet,	VU Amsterdam, daily supervisor

TU Delft

Abstract

Online gaming is the world's largest entertainment industry by revenue, and supports over 3 billion consumers worldwide. Many of the world's most popular online games must manage millions of concurrent players through a single unified service. Achieving performant and scalable online games is challenging. Online games are subject to stringent quality of service requirements, notably extremely low response times, with at most 50ms being considered acceptable. Unlike many other types of applications, the performance of online games depends to a large degree on the resources available on end-user devices. These devices are typically heterogeneous, limited in compute and network resources, and subject to unpredictable changes in resource availability.

Addressing this challenge, we propose in this work the concept of *differentiated deployment*, which allows online games to selectively manage and scale online-game systems with fine granularity in response to changes in available resources. We design Polka, a framework for online games which supports differentiated deployment. We then implement PolkaDOTS, an open-source proof of concept of the Polka framework built in an industry standard game development ecosystem.

We evaluate our approach using Dither, a custom-built experiment runner for large scale distributed experiments on online games. We use Dither to perform real-world experiments on a representative Minecraft-like Game (MLG), Opencraft 2, built on the PolkaDOTS stack, and analyze the impact of various differentiated deployment scenarios. From these experiments, we find that differentiated deployment can decrease performance variability of online-game servers, and decrease the response time experienced by players by up to 32%. Most importantly, we show that differentiated deployment enables novel deployment techniques, including switching from local rendering to cloud-based rendering (i.e., cloud gaming) at runtime.

Preface

Online games are uniquely fascinating. They have a simple yet undeniable appeal: they allow people to share experiences regardless of physical distance. In practice, they form emergent and unpredictable microcosms of society and culture, and from a technical perspective they are some of the most complex distributed systems. In my opinion, much of the potential of online games has yet to be explored.

Over the past five years, I have had the privilege of working on the Opencraft project within the AtLarge Research group. The scope and scale of the Opencraft project has expanded since I joined the team. What originally focused on massivizing a genre of online games has grown into an exploration of the very boundaries of the meta-verse. The head of the Opencraft project, Jesse Donkervliet, has been my mentor since the first day, and has given me countless hours of guidance, feedback, and perhaps most important, perspective. Of course, I cannot forget to acknowledge Prof. Alexandru Iosup, for getting me into the group in the first place, and for a great deal of academic and moral support during the multi-year process of submitting my work for publication.

This thesis marks the culmination of my time with the Opencraft project, and incorporates all I have learned both through my own work and through collaboration with my extremely talented peers in the group. I created this report and the implementations of PolkaDOTS, Opencraft 2, and Dither with them in mind. I hope that these tools will foster the next round of research within the Opencraft team, and allow us to continue pushing the boundaries of what is possible with online gaming.

The finalization of this thesis was delayed considerably by administrative error which required a momentous amount of flexibility and support from a variety of people to overcome. Therefore, I would like to thank the members of my thesis committee, Prof. dr. ir. Fernando Kuipers, Dr. Asterios Katsifodimos, Prof. dr. ir. Alexandru Iosup, and Ir. Jesse Donkervliet for their patience and adaptability. Additionally, I wish to thank Prof. dr. ir. Dick Epema for his role in coordination despite being unable to join the thesis committee, as well as Mr. Michel Rodrigues for much-needed guidance throughout.

Finally, I must give a special mention to Ana, who got me through the bureaucratic hurdles and all the five years prior, and without which this thesis would have no end date at all.

Jerrit Eickhoff
Delft, March, 2024

Contents

1 Introduction

Video gaming has become a dominant form of entertainment on a worldwide scale. The video game industry generated more than 184 billion USD in 2023, from an estimated 3.38 billion video game consumers worldwide [57]. 49% of that revenue comes from the global mobile gaming market. A key factor driving this growth is the social aspect delivered by online, multiplayer games. Of top 50 most played PC games, 88% are exclusively multiplayer or feature a multiplayer mode [44]. Online games are not only an entertainment format, they are increasingly important to other aspects of society, such as socialization [29, 50], education [42, 63], and activism [6, 7].

Building a functional online game is an engineering challenge. Video games run complex simulations to generate ever-changing *game states*, and must ensure that players receive this state subject to strict quality of service requirements, even through unreliable network connections. The time and resource constraints involved in game development, paired with the complexity of designing a video game from the ground up, has popularized the use of preexisting game frameworks, engines, and development ecosystems. These allow game developers to reduce development time and cost by bootstrapping the process of game creation, and facilitating reusing and sharing assets between games. It also allows game asset creators and artists to collaborate with programmers using shared platforms with standardized formats and protocols.

Since a majority of modern online games are built using game frameworks, they tend to utilize the networking tools provided by the framework. These networking tools are built to be general-purpose and support any game that can be developed on the framework. To be general purpose, they are designed around a simple but idiomatic networking paradigm: a single, centralized server that computes and distributes the authoritative state to all clients. Deployment of the online game is then synonymous with deploying the server, and scaling the online game can only be done by either optimizing the server or deploying more servers. This results in an inflexible deployment scheme where servers are created piece-wise and do not share state, greatly limiting both the maximum number of players in a single server and the supported complexity of online games.

1.1. Performance in Online Games

The scope and scale of online games has continually increased, leading to higher hardware requirements both for players and online-game providers. This has lead to the popularization of *cloud gaming*, and the development of a variety of techniques to reduce the computational load of hardware resources by limiting what state updates are computed and distributed, broadly termed *consistency management*. As games have sought out enhanced visual fidelity, graphics hardware requirements for users has grown. While specialist hardware is now available to support modern games, it is typically prohibitively expensive for many consumers [16]. An alternative is cloud gaming, where end users interact with a thin client that displays a real-time video stream of a game client deployed on cloud hardware. Beyond the graphics requirements for end users, the amount of concurrent players and the complexity of game simulation in typical online games has increased as well. To support more players but still meet quality of service requirements, a common modern technique is to intentionally introduce inconsistency. If players have different versions of the game state at the same time, their views of the game are said to be *inconsistent*. While inconsistency is generally undesirable, the designers of online video games may use inconsistency to ensure that the state updates a player receive with the lowest latency are the most important state updates for that player. For example, preferring state changes to the area nearest them in game [12, 41].

1.2. Problem Statement

Online games attempt to run complex, distributed simulations on heterogeneous hardware, subject to stringent latency requirements. The modern approach to this problem utilize a static deployment of a monolithic server. To ensure the server remains performant, it is common practice to over-provision resources. Increasing scalability with a monolithic server is viewed as either minimization of server computation, such as through consistency management [12, 41], or through *sharding* [3, 11], deploying multiple independent dedicated servers.

The approach of statically deploying a server on over-provisioned hardware and minimizing server computation popular because it is simple and effective. However, this approach has the drawback of making the online game extremely inflexible. Because deployment of the server is performed in a coarse grained fashion and only once, the online game is incapable of reacting to changes in workload or available resources. In practice, this can lead to inability to maintain service quality, or service failure outright.

Differentiated deployment refers to the ability to selectively deploy individual services. We hypothesize that a differentiated deployment mechanism for online games would allow separation of the monolithic server into a set of independently deployed and scaled services. Additionally, by continuously observing system performance, the differentiated deployment mechanism could allow dynamic migration of these services, letting an online game flexibly adapt to changes in workload and resources.

1.3. Research Questions

We outline **R**esearch **Q**uestions towards exploring the feasibility and advantages of a granular online-game architecture, supporting dynamic and differentiated deployment of online games.

RQ1 How to design an efficient online-game architecture that supports differentiated deployment?
Online games must operate on a highly heterogeneous and dynamic network of end user hardware and commercial computing resources. However, existing online game approaches favor static deployment of a monolithic game server, tightly bundling many game components. An architecture that allows differentiated deployment of game components has many potential benefits. Individual components can be scaled independently, potentially increasing maximum supported players and reducing computational overhead. Components that handle player input directly could be deployed closer to players, reducing latency. Game asset generation can be removed from the game content creation loop, which may reduce performance variability.

Such an architecture is not trivial, it must map the game state dependencies of each component, and provide a mechanism for ensuring these dependencies are available, regardless of component deployment. It must also support a clear and precise method to configure differentiated deployment scenarios.

RQ2 How to implement our design supporting state of the art game development tools?
Only a minority of modern games are built from scratch. Game developers make use of a variety of well-established frameworks, tools, and services. Some examples include game engines, networking libraries, hosting platforms, and storage services. Leveraging existing tools not only greatly expedites the development process, but allows games to share a standardized set of formats and interfaces, facilitating re-use of game assets and ease of integration with external services. A set of game development tools and services sharing format and interfaces form a *game development ecosystem*. Therefore, for our design to be pragmatic, it should integrate with existing game design standards, and be compatible with at least one of the many popular game development ecosystems.

There are two major difficulties in integrating our design into an existing game development ecosystem. First, these ecosystems are not cross-compatible, so we must choose from a large and diverse set an ecosystems to target. Second, use of an ecosystem enforces additional requirements on our design in order to be compatible. For instances, to utilize an existing game engine, we must utilize the game engine's existing programmatic interface in a compatible programming language.

RQ3 How to evaluate the efficacy of differentiated deployment for online games?
Online games are both numerous and extremely diverse. Even within a single genre, each game runs

different simulations under varying latency constraints. To further complicate the issue, online games are deployed on highly heterogeneous hardware. Therefore, it is necessary to evaluate the differentiated deployment approach to discern how much, and under what specific scenarios, online games can benefit from it.

Evaluating differentiated deployment challenging both for conceptual and technical reasons. Conceptually, evaluating differentiated deployment requires identifying suitable and representative workloads, metrics, hosting environments, as well as selecting promising differentiated deployment policies from an unbounded design space. Technical challenges to the evaluation process include the need to design and implement a compatible experiment tool capable of running large-scale experiments in a heterogeneous distributed environment, as well as instruments to collect and analyze the large volume of data collecting in experiments.

1.4. Research Methodology

We answer our research questions utilizing a distributed systems approach combining conceptual, technical, and experimental work, following the AtLarge Design Process [25]. Our work adheres to principles of *open and reproducible science* [2, 34], and our experiments are fully reproducible and self-contained [1, 52].

We answer RQ1 through the iterative design of a novel online-game framework that supports differentiated deployment (Section 3.1). We start the design process by discovering requirements through stakeholder analysis, then provide an overview of a design fitting those requirements, and finally provide detailed design for specific differentiated deployment mechanisms. We implement the resulting design (RQ2) and describe the implementation details in Section 3.2.

We answer RQ3 through real-world experiments on public compute clouds (Chapter 5). To support our experiments, we design and implement a representative online-game built on our framework (Chapter 4). Further, we provide evaluation of the implications of our findings as well as practical actionable insights for game developers and online-game operators.

1.5. Thesis Contributions

Through the exploration of the research questions, our work contributes a variety of significant conceptual and technical advances:

1. **(Conceptual)**

   (a) **Design of Polka**: *the first framework for dynamic differentiated deployment in online games* (Section 3.1). This contribution is comprised of several key conceptual advances: (1) an online-game architecture that facilitates separation of previously tightly-coupled game components, (2) novel deployment categories for online games, allowing more granular control than current static, monolithic, Server and Client deployments, (3) a flexible approach to improve online-game service quality by dynamically adapting to resource availability, and (4) a distributed performance monitoring technique to facilitate performance analysis and allocation policy development.

   (b) **Design of Opencraft 2**: an online game utilizing the Polka framework that demonstrates how online games can utilize Polka's functionality (Section 4.1). We design Opencraft 2 to be representative of the popular genre of Minecraft-like Game (MLG)s. We design Opencraft 2 as a research platform. To this end, Opencraft 2 is fully open-source, is easy to extend, and has configurable and reproducible workload. Opencraft 2 is currently being used for at least two bachelor theses.

   (c) **Evaluation of Polka**: using experimental analysis in real-world deployment of Opencraft 2 we evaluate and analyze the capabilities and performance of Polka (Chapter 5). The dynamic, differentiated deployment scenarios that Polka enables are novel, thus we compare the benefits of these scenarios against static, Server-Client deployment baseline. We then use this comparison to analyze the performance and quality-of-service benefits of the novel features of Polka.

2. **(Technical)** Implementation and Documentation of Polka related artifacts, with all data publicly available following Findable, Accessible, Interoperable, and Reusable (FAIR [58]) principles, and all software published as Free-access Open-Source Software (FOSS) artifacts:

(a) PolkaDOTS, an implementation of our Polka framework in an industry-standard, state-of-the-art game development ecosystem that supports cross platform and large team collaborative development (Section 3.2). Artifact publicly available on GitHub [18].

(b) Opencraft 2, the open-source implementation of our online MLG design built on the PolkaDOTS stack (Section 4.2). Opencraft 2 is intended to act both as an evaluation artifact for Polka as well as a research development platform to support future research. Artifact publicly available on GitHub [17].

(c) Dither, a reproducible experiment system based on the Continuum benchmarking framework [26] (Section 5.1). This system allows running experiments using Opencraft 2 on the commercially available Google Cloud Compute platform and automatic creation of relevant performance metric visualizations. Artifact publicly available on GitHub [37]. Data available on Zenodo [19].

2 Background

In this chapter we describe and explain several concepts relating to online games that are essential to our work. We split our description into three parts. First, we explain the existing approaches for deploying online games. We then describe and explain our system model for online games. Finally, we outline networking techniques common in online games, and how they relate to the deployment approaches.

2.1. Online-Game Deployments

Online games are complex distributed systems. A *deployment strategy* for online games refers to how the components of the online game are divided across multiple devices. There are multiple state-of-the-art strategies to deploy online games. These deployment strategies vary in price, performance, and ease of development, but all have one key drawback: they are static. Once the deployment process has completed, it is difficult or impossible to modify without performing the deployment once again. In this section we outline the various methods of online-game deployment and discuss their properties, scaling methods, and implications. A visual overview of the deployment models we discuss is shown in Figure 2.1.

2.1.1. Dedicated servers

A *dedicated server* is a standalone, monolithic game server artifact that is typically deployed utilizing commercial cloud services. Cloud providers offer special functionalities for dedicated game servers on their services, such as automatic deployment and scaling in response to an increase in players, as well as integration with any Online-Game Services (OGS) they offer. Dedicated server deployment is chosen if an online game requires a high player count, such as when deploying a massively multiplayer online games (MMOG), or when a game studio wants to ensure service availability and quality. Dedicated server deployment is costly for game studios and can require a considerable amount of configuration, and thus many studios will release the dedicated server for end-users to deploy on their own. This has an overlooked side effect of increasing the longevity of the online game. Many online games with studios who have stopped hosting servers themselves have been kept active by community hosts.

Scaling in a dedicated server deployment can be done vertically or horizontally. Vertical scaling entails purchasing more or better hardware for the dedicated server application to run on. Of course, there is a practical limit to how much a dedicated server can be scaled vertically. Horizontal scaling entails deploying more dedicated servers, and has two main variants: *sharding* and *zoning*. Sharding is the simplest form of horizontal scaling, and it involves deploying multiple, separate, dedicated servers. Game state is not shared between dedicated server replicas, so players on different server replicas cannot interact. Zoning is similar conceptually in that it splits workload between multiple dedicated servers, but instead of being independent, each server handles a subset of the game world. Players can then migrate between zone servers by moving their in-game avatar.

2.1.2. Listen servers

A *listen server* is not deployed on the cloud, but is instead hosted by end-users. Players wishing to join an online session can either deploy a listen server or connect to an existing one. Listen servers require no upkeep costs by the game developer, and do not require advanced deployment systems such as autoscaling. They

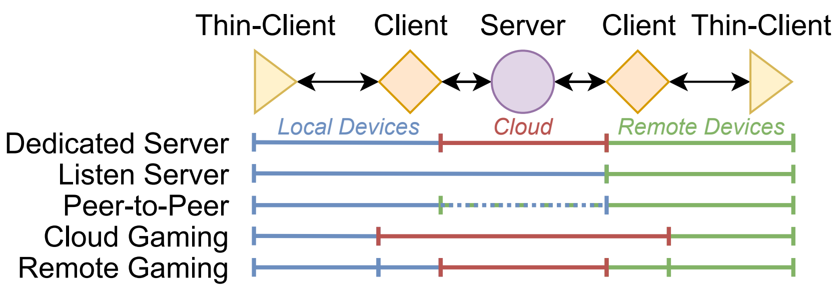


Figure 2.1: Overview of online-game deployment techniques.

also simplify game architectures: a game that has both an multiplayer and singleplayer mode can reuse the listen server for the singleplayer mode. However, listen servers are limited by operating on consumer hardware. End-user devices have limited hardware resources, and since the device hosting the listen server is also running a game client, the server and client must split these resources. Thus, listen servers are suitable only for online games with a low maximum number of players per server.

Availability, consistency and security are difficult to guarantee in a listen-server deployment. End-user devices can lose connectivity without warning. If the end-user hosting the listen server loses connection, then the other players in the online session must either be forcefully disconnect or attempt to elect a new host, start a new listen server on that host, and reconstruct the game state; a process known as *host migration*. Similarly, any persistent state of the server is linked to the host, requiring the same end-user to operate the listen server in order to recover the persistent state. Of course, since the listen server is run on end-user devices, there is no feasible method to prevent malicious tampering with the listen server, allowing the host near unlimited opportunity to perform exploits. Even without malicious intent, the near-zero latency that the hosting player receives compared to all other players makes listen servers unsuitable for competitive games.

Additionally, listen servers are deployed behind a local area network (LAN). Consumer router hardware limits direct connection between the LAN and wider internet. This requires the listen server to perform an unreliable technique known as *hole punching* to bypass router limitations, or to have end-users manually configure their networking equipment to allow external connections. Alternatively, various OGS providers host *relay* services, which forward traffic between the listen server and connected players. Relay services increase round-trip time and therefore player latency, and also must be facilitated by the game studio, for instance through OGS. Modern games utilizing the listen server deployment strategy typically use hole punching with a relay service as a fallback.

2.1.3. Peer-to-Peer (P2P)

Peer to peer (P2P) is the broad term for a networking paradigm where there is no central server. Instead, endpoints, or peers, directly communicate with one another. By not having any one peer hosting the server, the possibility of exploits and the reliance on a single user's hardware are reduced. Theoretically, P2P enables near unbounded scalability since every peer is adding to the total available computational resources. In practice, P2P deployments have many of the same downsides as a listen server, and additional downsides that listen servers do not have. The most obvious downside is that P2P requires all peers to run game server logic, instead of just one. Likewise, all peers must use the network hole-punching technique or a relay service to enable communication through local area network (LAN)s. Since all peers communicate directly, peers that are geographically distant will cause an increase in latency for all other players, instead of just for themselves. Since not all peers have the same hardware, the entire system is limited by the speed of the slowest peer. To ensure no peers are performing exploits, distributed consensus algorithms are necessary to check player action validity. Beyond that, the fully connected nature of P2P models exposes the internet address of every peer, which can pose a security risk.

These challenges have lead to P2P not being widely adopted by the online-game industry. Listen servers are preferred over P2P despite their limited scalability and exploitable nature. There are several notable exceptions, such as Bungie's Destiny series, though their hybrid P2P implementation has been met with considerable controversy by the Destiny player base [22]. Ubisoft's 2017 title For Honor was initially released using P2P netcode, but was later modified to use a dedicated server model after player backlash [5].

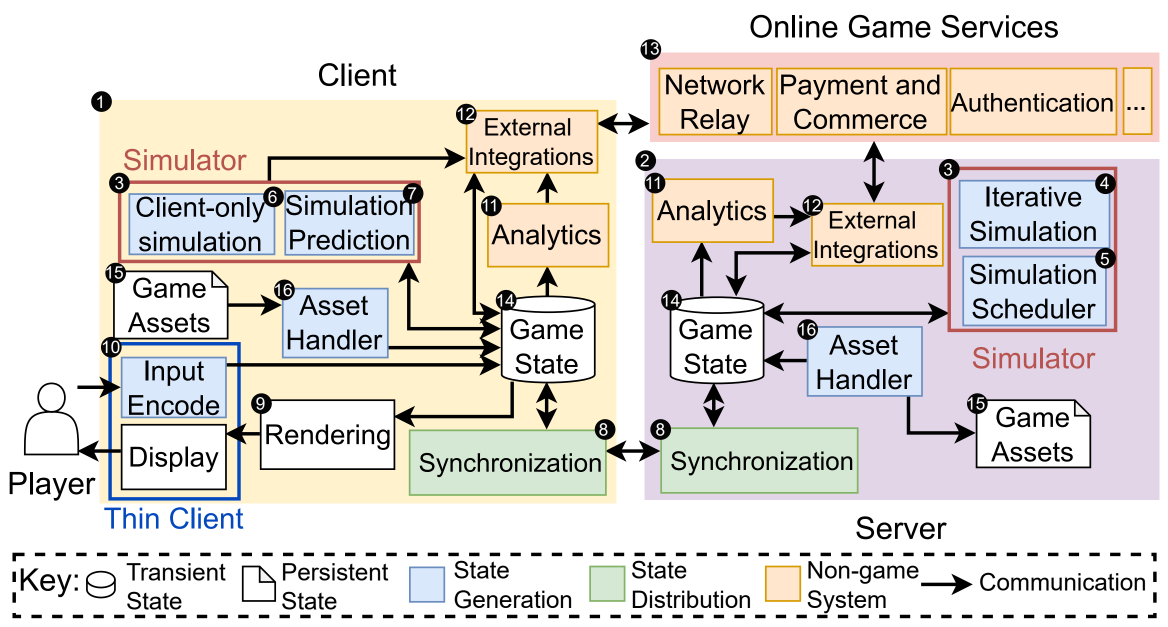


Figure 2.2: High-level model of online games.

2.1.4. Cloud and remote gaming

As opposed to dedicated and listen servers, *cloud gaming* refers to how the game client is deployed. In cloud gaming, a remote game client is deployed in cloud datacenters, rather than the end-user's device. Instead, the end-user runs a minimal, *thin client* that only contains input collection and video stream rendering components. The thin client sends a stream of player inputs and receives a video stream of rendered frames from the cloud client. Cloud gaming is beneficial when end-user devices have insufficient hardware resources to run the game client, but sufficient network resources to receive the video stream at a reasonable quality and latency. Cloud gaming as a deployment paradigm has many promising implications: first, since the client is not run on end-user hardware, the attack surface of exploits is reduced considerably, which could remove entirely the computational overhead of checking non-trusted player actions. Second, the client can be co-located with the game server for optimal latency. Third, as the client is decoupled from the end-user, a new category of game features is possible which would be run on the client when it is not directly controlled by the thin client. Finally, the video stream from cloud clients could easily be multicast, allowing for latency free spectating systems or instantly transferring control of the cloud client between thin clients. A downside of cloud gaming is increasing latency due to communication time between datacenters and end-users. To minimize this latency, prior work has explored deploying cloud gaming services to the *edge*, moving services physically closer to end-users [30, 62].

A variant of cloud gaming is *remote gaming*, which has the remote client hosted on end-user hardware, but not in the cloud. Remote gaming is commonly used to stream games within a LAN, for instance from a stationary personal computer (PC) to a mobile device, or from a mobile device to a multimedia system with a larger display. It is also used to host local-only multiplayer games as online games, for example through Valve's Remote Player Together [53].

2.2. System Model for Online Games

Although online games commonly use a client-server architecture, viewing the online game as only a discrete server and client effectively hides the complexities of the interconnected services used to run an online game. In this section, we introduce an system model for online games. We then use this model to discuss the services most relevant to our work and to demonstrate the drawbacks of how games are commonly deployed today. Figure 2.2 shows our system model.

2.2.1. Overview

Online games can be conceptualized as a distributed, real-time, content creation and distribution system. At a high level, online games are made up of two logical content-producing systems: a *client* (❶) and a *server* (❷). We outline the forms content takes in online games in Section 2.2.2.

Content in both the client and server is created by *simulator* (❸) components. The set of simulator components is not identical between client and server. Generally, a server creates a majority of the online-game content through *iterative simulation* (❹): producing new content based on what content has already been created. Iterative simulation may contain many types of simulation operating at different rates and producing different types of content with varying computational requirements and latency restraints. Operation of the iterative simulation process is performed by a *simulation scheduler* (❺). Not all content needs to be distributed, for instance if it is only relevant to a single player. This is handled by the client simulator using *client-only simulation* (❻).

Even though the server simulator is responsible for a majority of the game content, it is typical to run much of the iterative simulation on the client simulator as well. These shared simulations increase the computational requirements of the game client, but enable *simulation prediction* (❼). Simulation prediction allows clients to hide much of the latency incurred by content distribution. In our model, content distribution is handled by a *synchronization* component (❽). We explain simulation prediction and other content distribution techniques in Section 2.3.

The game client consumes content by converting it to a format that players can engage with. This process is handled by the *rendering* component (❾), which passes the processed content to a *thin client* (❿). A thin client is responsible for directly interacting with the player, typically through collecting keystrokes and other player input, and interfacing with the player's device to display visual and audio output.

Both the game client and server contain components that are not directly related to creating, distributing, and consuming content. Instead, these components relate to the operation of the online game as a complex distributed system. For instance, collecting performance and behaviour *analytics* (⓫), and interacting with existing services through *external integrations* (⓬). There are many existing services specifically for the operation of online games, referred to as *Online-Game Services (OGS)* (⓭). We detail the types and use of OGS in Section 2.2.3.

2.2.2. Game content

The content generated and distributed by online games can be divided into two categories. *Game state* (⓮) is the ephemeral, transient value of game content at any given time. Game state is updated by the simulator components at a fixed but rapid rate. Consequently, the ability to access and modify game state as fast as possible is a crucial engineering challenge for achieving high performance in online games. The synchronization mechanisms between clients and servers operate only on game state, attempting to keep game state on all clients and servers as similar as possible at all times.

By contrast, the other type of game content, *game assets* (⓯), are generally persistent. Examples of game assets include 3D geometry data, texture images, and game saves. Accessing and updating game assets at runtime is performed by an *asset handler* (⓰). While assets do not have as much of an associated latency requirement as game state, handling assets is still a performance concern. Assets can directly impact game state, but may also use a significant amount of data. Asset handlers may dynamically and asynchronously load and unload assets as needed. Asset distribution is either done entirely in advance, often packaged with the game executable, or is done during game operation with a reduced latency requirement. For instance, many online games have some form of *procedural generation* which creates new game assets while the game is running. Games such as Mojang's Minecraft[45] generate persistent and expansive in-game environments in a "just-in-time" manor, creating assets to represent new areas of the game environment as players approach them.

2.2.3. Online-game services (OGS)

Since the popularization of the "as-a-service" model through the 2010's, online systems have become increasingly interconnected, forming complex ecosystems and developing surprising emergent behaviors. Online video games are no exception, and it is common practice for online games to interact with external services. Utilizing existing external services prevents game developers from having to re-implement many standard functionalities, and allows multiple online games to share common communication formats. Many commercial cloud providers offer *Online-Game Services (OGS)* that provide common functionalities for online gaming. Notable OGS providers are AWS for Games [40], Microsoft Azure PlayFab [31], Valve Steamworks [54], Epic Online Services [21], and Unity Gaming Services [47].

These OGS encompass a variety of functionalities. Services relating to handling players and groups of players are common. OGS identity management allows game developers to utilize existing cross-platform authentication, and comes with built-in support for *social gaming* functionalities. Social gaming services allow online-game players to create in-game groups, referred to as *lobbies*. Lobby services may include a *matchmaking* system, which searches all lobbies to find a set that would be a good matches for the online game. Similarly, lobbies may include communication systems for in-game text and voice messaging.

Other OGS relate to the operation of the online game. For example, specialized cloud storage systems for game assets. Assets such as game saves and configuration files can be synchronized to cloud storage for easy retrieval on multiple player devices. Assets that developers may not want to be modifiable on player devices, such as player inventory and digital goods, can be located and accessed only from the cloud. This category of OGS also includes more traditional online services. Online-game monetization is facilitated by payment and commerce services, as well as advertisement. Game developers can use cloud performance monitoring and analytics to diagnose their online game and find bugs.

2.3. Online Game Networking (Netcode)

The architecture, deployment, and operation of online games is heavily influenced by online game networking, referred to as *netcode*. Different netcode types have different sets of simulator components, and can widely vary in terms of computational and bandwidth requirements. Much like the deployment of the online game as whole, the set of components being run on the client and server are typically static. A motivating factor behind exploring differentiated deployment is the possibility and implications of a *dynamic* netcode, where the set of simulator components being run on the client and server are dynamically modified.

2.3.1. The netcode zoo

At it's core, netcode determines what, when, and how game content is distributed between all game servers and clients. All netcode systems can be categorized based on their system of *replication* and *authority*. Replication refers to what game components and content is present at what times on the server and clients, and authority refers to which server or clients hold the "official" version of a specific subset of game content. These two related concepts not only effect the details of content distribution, but also determine the architecture of the game as a whole, by indicating what game components are part of the client or server, and what parts are shared. Netcode with no client-side replication or authority (e.g. *snapshot interpolation*) will have no content generation components in the client, whereas netcode with full replication (e.g. *deterministic lockstep*) will have every content generation system shared between client and server.

Netcode is rarely made from scratch for a given online-game project. Larger game studios tend to have in-house netcode systems that are adapted and reused in many online-game titles. Some prominent examples are Valve's Source Multiplayer Networking, used in Dota 2, Team Fortress 2, Counter-Strike: Global Offensive, and Left 4 Dead 2, among others [55], as well as Nintendo's NEX system, which was the core netcode of every Nintendo game with online functionality released between 2003 to 2021 [14]. Rather than develop and maintain custom netcode, smaller game studios may choose to use commercially available or open source netcode libraries, with some popular examples being Mirror [32], Netcode for GameObjects [51], Photon [35], and Normcore [33].

To ensure reusability and portability, these netcode libraries do not make assumptions specific to a particular game. Instead, they implement one or more general game networking techniques. There are many relevant networking techniques, but we enumerate and explain the following important examples: deterministic lockstep, snapshot interpolation, state synchronization, interest management, lag compensation, and client-side prediction.

Deterministic lockstep requires a deterministic game engine, where a given state and input will yield the same new state with no variance. This a challenging prerequisite, but if satisfied, then all clients can run the same simulation, with the server only forwarding player inputs to all clients. Determinism ensures that all clients stay in sync and provides inherent exploit resilience. This technique is bandwidth optimal, but does not handle poor connection quality well and requires all clients to run the entire simulation which may be infeasible on devices with limited resources.

Snapshot interpolation, by comparison, has the server computing and sending the entire game state at a fixed rate to all clients. Clients do no state computations on their own, but instead interpolate between the last two received game states. This is bandwidth inefficient, but highly scalable in terms of player count, and simple to implement as the server holds authority for the entire game state.

State synchronization seeks to strike a balance: like deterministic lockstep, simulation is run both on server and client, and input is sent from the client to server, but like snapshot interpolation, the server holds the authoritative state and sends it to the clients, and there is no requirement for determinism. Instead, when a client receives a game state from the server, it smoothly interpolates its own, potentially incorrect, state, to the correct state received by the server. This is an example of *inconsistency management,* which we explore further in Section 2.3.2.

Lag compensation is a method to mitigate or hide the effects of network latency. When a server receives a timestamped player action, it computes the result of the action based on what the game state was at that timestamp, rather than the current game state. Lag compensation requires keeping a buffer of previous game states within a set amount of time, and must be carefully implemented as lag compensation favors a player making an action, which may be unfair if the result of that action is applied to a different player.

Client-side prediction, like lag compensation, seeks to reduce the effects of network latency. Client-side prediction requires the client to compute game state, and immediately applies any player actions locally. The server's authoritative response to player actions will arrive back at the client after a delay of at least twice the latency between server and client (i.e. round-trip time). If the client's game state differs from the server's response, the client has mispredicted. It must then perform *reconciliation*, either through smooth interpolation similarly to state synchronization, or through rolling back the game state to the last server authoritative game state and recomputing game state from there.

2.3.2. Consistency in online games

Online games are continuously becoming larger, more immersive, increasingly detailed, and more dynamic. A notable side effect of these trends is that the size of game content has steadily increased, which requires larger and more frequent game state updates. As online games attempt to squeeze more game state updates onto public internet infrastructure, the result is an increase in *performance variability.* Online games are sensitive to performance variability. Simulators and netcode run at set rates, and performance variability is characterized by decreases in these rates. Such variability becomes visible to players in many ways, all undesirable, such as annoying delays to player inputs and actions having visible effect, or players and objects appearing to teleport back and forth at random.

In order to mitigate the performance variability caused by increased workload and network constraints, and important class of netcode optimizations have been developed: those of *consistency management.*

Consistency is a comparative property of game state. Two views of game state are said to be consistent if they encode the same game view at the same logical time step (i.e. the same game tick). Traditional netcode techniques, notably deterministic lockstep, have implicit but strict consistency constraints, with all clients having exactly the same game state during every tick. However, maintaining strict consistency becomes challenging and expensive as virtual environments become more complex. Therefore, modern netcode relaxes these constraints by employing *eventual consistency.* Instead of a strict game state equality, there is now an inconsistency limit, where clients will have game state that is not identical, but is guaranteed to be "close enough" with a bounded amount of time. Within this paradigm, specific subsets of game state can be given a per-client priority, through *interest management.* Interest management takes some priority heuristic, such as distance to a player's in game avatar, or elapsed time since the player interacted with an object. Game state with lower priority according to these heuristics is updated less frequently, greatly reducing the bandwidth and simulation costs on each client. More advanced systems can implement *dynamic consistency,* to continually modify priority heuristics for each partition of the game state or for each player [12].

Interest management policies must be designed carefully, as an interest heuristic that does not approximate a player's actual interest well enough can reduce quality of service significantly. As an example, consider a competitive first-person-shooter game using a proximity based interest management. In this scenario, the interest management policy would favor players using short-ranged weapons and disadvantage those using long-range weapons.

3 Polka: A Framework for Differentiated Deployment in Online-Games

In this chapter, we introduce Polka, a framework for online games that supports *differentiated deployment*. We discuss differentiated deployment in detail in 3.1.3. Online games experience sudden changes in hardware resource availability. For example, on desktop platforms, programs running alongside the online game compete for resources such as CPU and memory. On consoles, downloading games or updates in the background reduce available network bandwidth. Mobile devices receive highly variable network quality while moving, and throttle performance when battery charge is low. On *dockable devices* such as the Nintendo Switch or Steam Deck, available resources are improved drastically when in the docked mode.

Although the amount of available resources changes over time, online games are commonly deployed statically. Static deployment makes the online game susceptible to significant performance variability caused by fluctuation in workload or resource availability [20]. Polka allows online games to adapt to changes in workload and resources by dynamically migrating individual game services.

3.1. Design

In this section, we address the first research question (RQ1) with the design of Polka, an online-game framework supporting differentiated deployment. As outlined in the AtLarge Design Process [25], we design the Polka architecture iteratively. We begin with problem analysis and formulating requirements (stage 1 and 2 of the AtLarge Design Process, Section 3.1.1). Based on the problem analysis we bootstrap the creative process (stage 3) with a key insight: *Polka can facilitate application-specific goals by moving the task of resource management into the application layer.* Using this principle, we provide a high-level design overview of Polka (Section 3.1.2) followed by low-level design of key functionalities (stage 4, Sections 3.1.3 to 3.1.5)

3.1.1. System Requirements

In this section, we formulate requirements that must be met by the design of Polka. This is the first stage of the AtLarge Design Process [25]. We find requirements through use-case analysis. First, we identify relevant stakeholders for online games, and determine several important use-cases of Polka for each stakeholder. We then establish functional and non-functional requirements for Polka to support these use-cases.

Stakeholders

We identify four relevant **S**takeholders of our architecture:

(**S1**) **End-users** or **Players** are the primary end-users of an online game. Unlike consumers of other networked systems, they also act as an important source of game content. After all, a primary feature of online games is the ability to interact in real-time with other players. Players are not typically interested in the details of how an online game is deployed, instead caring that the service quality they experience remains adequate.

(**S2**) **Game developers** are the potential adopters of Polka, responsible for engineering and creating an online game. Game developers must weigh many architectural and implementation design decisions under stringent time and monetary resource constraints. Thus, developers will judge the efficacy of our design by how suitable it is for their game in terms of the trade-off between additional programming difficulty and improved capabilities.

(**S3**) **Game providers** are the service hosts of an online game. The game providers can be the game developers (e.g. *in-house hosting*), but it is increasingly common for the game providers to be a commercial cloud service hired either by the game developers, or in some cases, players. Like other online-service providers, game providers must maintain contractual agreements on service availability and quality while minimizing hosting costs. Therefore, game providers are interested mainly in whether games developed using our design are in some way beneficial to host compared to comparable games without our design. The primary metric game provider use in this comparison would cost, but there are other relevant metrics, such as risk reduction and server schedulability.

(**S4**) **Researchers** study and develop models, behavior, and practices relating to online games. They are responsible for fostering academic collaboration towards the development, assessment, and adoption of new techniques. The use of our design by researchers in the future is of particular importance, as we intend the design presented in this work to be the starting point of an active research project life-cycle, with future researchers improving, analyzing, and extending our design.

Use-cases

Each type of stakeholder of online games uses them in different ways, and therefore have different requirements. From our identified stakeholder we derive 6 important **U**se-**C**ases where Polka is applicable:

(**UC1**) **Improving quality of experience (QoE) for players**
A *player* (S1) attempts to move their in-game avatar on a low-powered mobile device to avoid an in-game hazard. However, their device is currently overloaded due to simulating a complex section of the environment. The player's avatar fails to avoid the hazard, resulting in understandable frustration from the player. With Polka, the overloaded state of the player's device could be avoided by dynamically disabling the complex environment simulation, allowing the player to react in time to save their avatar.

(**UC2**) **Increasing scalability for developers**
A *game developer* (S2) wants to run a digital festival event for 200 players in their online game. This online game regularly handles thousands of concurrent players, so the developer expects the event to be no problem. However, during the festival, the CPU and network bandwidth used by the server increases exponentially, and players are randomly disconnected due to network timeouts. Polka could increase the scalability of the game in this scenario by enabling, disabling, or migrating services to reduce the overall consistency constraints of the online game, allowing more players to exist in a localized area.

(**UC3**) **Reducing cost for game providers**
Game providers (S3) must spend hosting costs to achieve the advertised player count for an online game. To account for unpredictable surges in player-count, they over-provision resources even during relatively low activity periods. Polka could reduce costs incurred through fine-grained scaling. Specific services, such as those relating to handling new player connections, could be scaled independently, allowing the entire online game to elastically respond to surges in player count without needing to incur cost by scaling the entire game server.

(**UC4**) **Reducing risk for game providers**
A *game provider* (S3) hosts an online game with a digital economy system, where players can trade digital items with real-world cost. The online game is deployed as a single static server, which experiences a temporary power failure. Even though a backup is quickly restored, players find they have lost some digital items, corresponding to the loss of thousands in real-world currency. The game's players subsequently lose trust in the digital economy, negatively impacting the game provider. Using Polka, services relating to the digital economy could be differentiated and deployed separately, and continued to operate even when the main server fails.

(**UC5**) **Expanding game design possibilities for game developers**
A *game developer* (S2) wants to develop an online game that is very bandwidth intensive, so much that a consumer LAN hardware can only reliably support a single game connection. This effectively prevents players on the same LAN from playing together, and results in many players who live together and wanted to play the game together getting a refund. Polka could allow game developers to make the

game work for players on the same LAN by allowing client-to-client connection. Multiple players could share the same server to client connection, rather than each needing their own.

(**UC6**) **Facilitating research and development for researchers**
A *researcher* (S4) specializing in online games wants to perform experiments measuring the impact of different netcode techniques on game clients. However, all the existing frameworks that support performance evaluation experiments either do not include the game client, or do not support modifying netcode. Polka allows researchers to do this experiment, among others, by acting as a platform for online-game research.

Functional Requirements

We derive 4 **F**unctional **R**equirements for Polka from these use-cases. In Table 3.1, we specify what requirements of Polka enable which use-cases.

Table 3.1: Relationship between use-cases and system requirements.

		QoE	Scalability	Cost	Risk	Game design	Research
		UC1	UC2	UC3	UC4	UC5	UC6
Differentiated deployment	FR1	●	●	●	●	◐	●
Deployment scenarios	FR2	●	●	●	●	◐	●
Game ecosystem	FR3	○	○	○	◐	●	●
Performance monitoring	FR4	●	◐	●	●	○	●
Scalability	NFR1	○	●	●	◐	●	◐
Quick migration	NFR2	●	◐	●	●	○	○
Extendable	NFR3	○	○	○	○	●	●

(**FR1**) **Support differentiated deployment of game components**
Polka should support differentiated deployment of game components to heterogeneous networked resources. This deployment mechanism must be capable of ensuring that the game state dependencies and latency requirements of separately deployment game components are met. Without FR1, Polka will behave identically to existing, monolithic, online games.

(**FR2**) **Allow specification of differentiated deployment scenarios**
Polka must support a variety of differentiated deployment scenarios, which map game components to heterogeneous networked resources within the scope defined in Section 2.2. The specification of these scenarios must be specific enough to function in heterogeneous networks, but general enough to allow experimental replication. Without FR2, Polka would be difficult to utilize and experimentally assess, making its adoption unlikely.

(**FR3**) **Integrate into existing game development ecosystem.**
The game development industry has well-established tools, methodologies, and practices. Polka should not be developed standalone, but rather integrated into this ecosystem, operating with existing tool and supporting established methodologies and practices. Without FR3, Polka would have a high barrier of entry and low portability, making its adoption unlikely.

(**FR4**) **Facilitate performance monitoring and analytics**
Polka must report detailed and relevant performance and operational metrics. These metrics must be gathered at a sufficient rate and within a reasonable real-time latency, to allow both for differentiated deployment policy mechanisms to function, and let game providers make operational decisions based on the real-time data. Without FR4, the deployment mechanism is unable to adapt to changes in resource availability, and it is impossible to assess the potential benefits of a given differentiated deployment scenario.

Non-Functional Requirements
Beyond the functional requirements, there are 3 **N**on-**F**unctional **R**equirements crucial to Polka:

(**NFR1**) **Maintain scalability.**
For differentiated deployment to be considered worth the additional system complexity, they must improve system flexibility without decreasing scalability. Thus, Polka must maintain scalability compared to existing deployment schemes. Without maintained or improved scalability, system complexity and overhead caused by scheduling components may prevent game developers from considering adopting Polka.

(**NFR2**) **Low migration latency.**
A crucial aspect of differentiated deployment is the ability to respond to changes in resource availability. Without this feature, games built on Polka may be strongly affected by changes in resource availability and exhibit high performance variability. Performance variability has been shown to directly affect quality of experience [20], so it is important for Polka to stabilize performance through flexible reaction to resource availability. Migration must be quick enough to adapt to changes in network and hardware availability, but not too quick to cause undue performance variability: adapting to resource changes that are short-lived. Acceptable migration latency depends on the type of resource and the impact on end-users. We propose a bound of 3 seconds for actions that interrupt end-user gameplay and a bound of 20 seconds for actions that are transparent to end-users.

(**NFR3**) **Extendability and modifiability.**
Polka, as an online-game framework, should be simple to integrate when building games on top of it. Additionally, third parties should be capable of adapting Polka functionality to their specific use-case requirements or to development new functionality within Polka. To this end, Polka's programmatic interface should be limited, requiring a developer to write only a single behavior script for basic functionality. Similarly, Polka should be open-source, well documented, and provide any associated build and testing tools. Without NFR3, Polka would be static and unable to adapt to novel online-game developments, limiting its lifetime and scope.

3.1.2. Design Overview of Polka

Figure 3.1 shows the overview of Polka's architecture. Each of the large colored boxes represents an individually deployable component of Polka.

The primary users of Polka are game operators that configure the deployment of an online game. This includes both game developers during a game design, implementation, and testing cycle, as well as game providers that operate an online game as a service to players. Users interact with Polka through a game *deployment component* (❶). By setting the *deployment configuration* (❷), users can control both deployment and game-specific parameters, addressing FR2. The *service deployment component* (❸) deploys game services according to the user-provided configuration, alongside information in the *Service Graph* (❻). The service graph is a precomputed directed graph of game systems with their game data dependencies (discussed further in Section 3.1.3). The service deployment component deploys a set of *servers* (❼), addressing FR1, and automatically performs per-server configuration.

Each server runs a subset of *game services* (❽) which read and modify a local subset of the total *game state* (❾). The order in which services are run and the game state each service operates on is stored in the service graph and configured by the service deployment component. Each server contains a *networking component* (❿) which synchronizes game state with other servers and with clients. The game state that is synchronized between servers is determined by the game state dependencies between the services that each server runs. By default, all game state is synchronized to all game clients, though it is possible to override this behavior using custom game services to implement per-player area-of-interest and dynamic consistency policies, partially addressing NFR3.

The *game client* (⓫) contains its own view of the game state (❾). This game state is used by a *rendering* component (⓬) to create frames that are displayed to game players. By default, client game state is a synchronized copy of all server game state, with smoothing performed using state interpolation. However, additional *client-side game services* (⓭) can change this game state update behavior, further addressing NFR3. There are two types of client-side game services: replicated services and client-only services. Replicated services are copies of game services that run in game servers. By replicating latency-critical game services in clients,

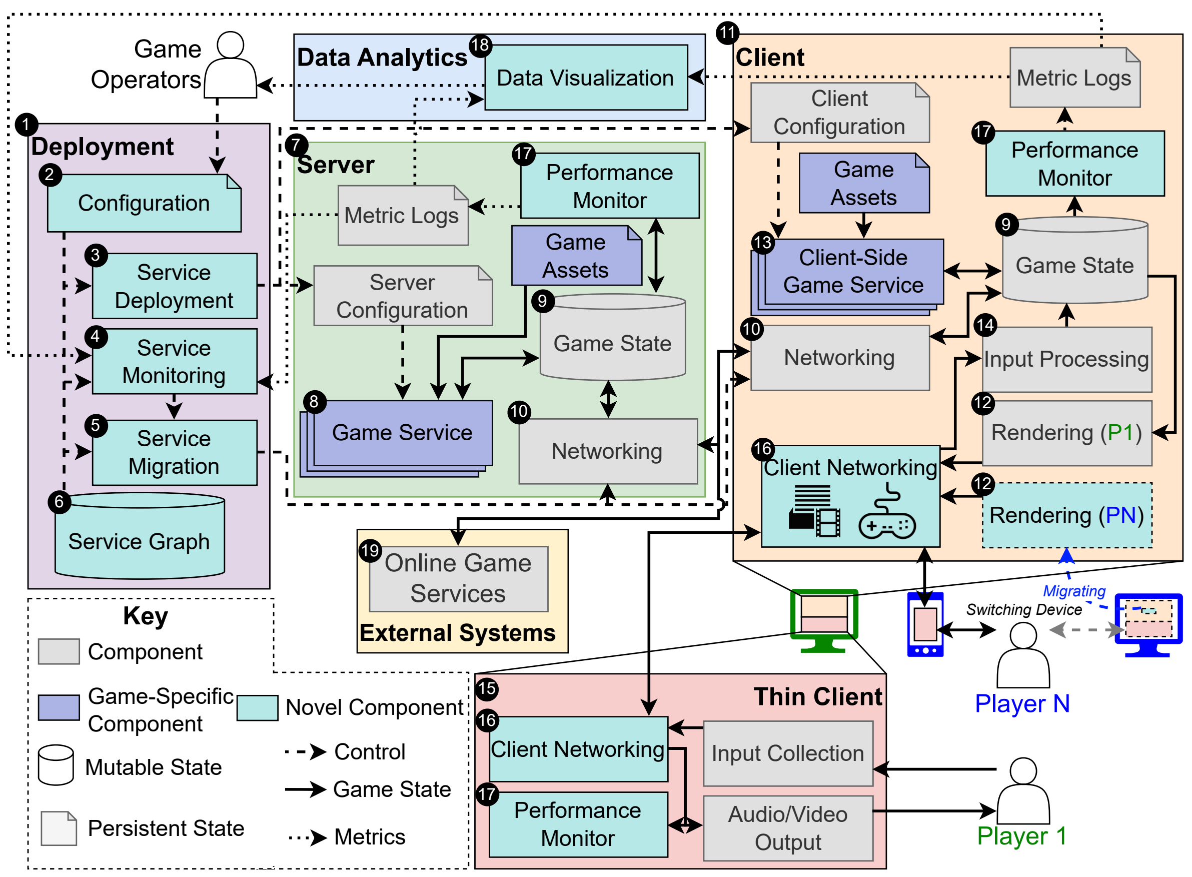


Figure 3.1: An overview of the high-level architecture of Polka

latency and network jitter can be hidden through using client-side prediction and rollback. Client-only services do not run on servers, and operate on game state that is not network synchronized. This is primarily used for front-end features, such as processing game state for use by the rendering component.

In Polka, there is a distinction between the game client and the game frontend. Each game client operates *input processing* (14) and rendering (12) for one *or more* players. Each player interacts directly with a *thin client* (15) which contains no game state and does not operate any game services. Instead, the thin client sends player inputs and receives rendered frame streams through a *client networking* component (16).

Both server and client modules interact with external *online-game services* (17) for integration into the modern online-game ecosystem, addressing FR3. These external services includes platform-specific authentication, user profiles, social and communication features, live content delivery, as well as operations services such as remote configuration.

To facilitate benchmarking and configuration optimization, all game modules generate performance and gameplay data, which are collected to metric log files by *performance monitoring components* (17), addressing FR4. These files can be read by a *data visualization module* (18) which aggregates and displays the data for the game operator in a visual format.

During operation of the online game, metric logs are synchronized to the *Service Monitoring component* (4). This component applies service performance policies, which can be configured by the user for a variety of goals. An important goal is flexibility: reacting to changes in resource availability. Thus, the service monitoring component can use the metric logs to detect performances changes and respond through service migration, handled by the *Service Migration component* (5). These components address NFR2. The ability to specify custom service migration policies to migrate or scale individual services allows the online game to react to and mitigate performance degradation. Therefore, these policies allow the online game to handle more workload without decreasing performance, increasing scalability (NFR1).

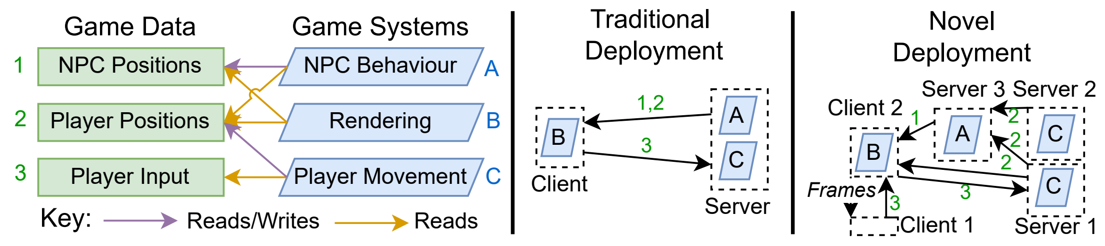


Figure 3.2: Comparison of traditional deployment and novel deployment opportunities enabled by Polka's service graph for an example online game.

3.1.3. Fine-Grained Component Deployment

In this section, we expand upon Polka's fine-grained differentiated deployment mechanism. Differentiated deployment is the process of making deployment decisions on the scale of individual game services based on the dynamically changing resources and Quality of Service (QoS) requirements of the application.

Constructing the Service Graph

We introduce the concept of a *service graph* (6) to represent co-deployed game services and the game-state they exchange. The service graph allows Polka's deployment mechanism to reason about what game services can be migrated and what game-state they require, supporting FR1. Without the up-to-date service-level view the service graph provides, migrating specific services is challenging as it is not possible to discern what other services depend on or are depended on by that service, and what game-state they require.

The service graph is a directed graph of *services*, which are each a set of *game systems*. Edges between services indicate *game-data* update dependencies. The service graph acts as an important heuristic for initial deployment by the service deployment component: game services with few update dependencies are likely to be good candidates for differentiated deployment.

To construct the service graph, we first need well-structured information on the online game. A minimal version of this information is a bipartite graph of *game systems* and *game state*. In this graph there are two types of edges corresponding to a game system's access to game data, *read* and *write*. From this bipartite graph, the service graph can be constructed by merging game systems nodes according to shared read and write dependencies.

Importantly, *the service graph imposes constraints on games built using Polka*. First, differentiable game behaviors in online game must be expressed as a set of discrete game systems, and all types of data the game uses be organized into a set of unique data-types. Second, every game system must be associated with a list of data-types that it reads and writes. Third, if multiple game systems write to the same data-type, there must be a *strict partial ordering* to the updates of these systems. By settings these prerequisites, online-game developers can ensure functionality when utilizing Polka without needing complex interfaces, addressing NFR3.

Our service graph deployment mechanism both enables new ways of deploying online games and is backwards compatible with existing deployments. For example, a service graph for a traditionally deployed game will consist of a single service node with no edges. We visualize an example, and novel deployment strategies enabled through the service graph concept, in Figure 3.2. This example depicts an online game where players move and interact with non-player character (NPC)s. In a traditional deployment, rendering is performed on a client, and player movement and NPC behavior is run on a server. Using the service graph, several beneficial new deployment options are made possible. The rendering service may now be run on a separate device from a client (e.g. cloud or streamed gaming). The player movement service run on multiple servers for horizontal scaling, and the NPC behavior service can be separated to run on its own server, without affecting the operation of the player movement services.

Service Deployment

Using both the operator-provided configuration and the service graph constructed from the game system and data relations, the *service deployment component* (3) performs the initial differentiated deployment of services (FR1). This initial deployment is a multi-phase process, consisting of *validation*, *deployment*, and *bootstrapping*.

The validation phase of deployment minimizes risk from misconfiguration. In this phase, the service deployment reads the user-provided service deployment configuration from the configuration component (2).

It then performs several sanity checks, including querying remote network resources to ensure authentication credentials are sufficient and comparing data in the service graph to user provided service operation policies.

During the deployment phase, the service deployment component uses the validated configuration to determine a candidate initial deployment based on service operation policy. Then, the online-game artifacts and any prerequisites are installed on networked resources used in the initial deployment.

Finally, the bootstrapping phase starts the online game on each node in a bootstrapping mode. In this mode, the online game performs local validation of pre-requisites before connecting to and requesting configuration from the service deployment component. If all nodes successfully connect back to the service deployment component and request configuration, the bootstrapping completes by sending tailored configuration to each node.

Once the bootstrapping phase has confirmed all nodes are operational, the service deployment component annotates the service graph with the network location they are deployed to. This information is used by both the service monitoring and service migration components to keep track of the operation of the online game.

To facilitate the deployment and bootstrap, Polka imposes an additional constraint: *the game artifact must be a singular, polymorphic application.* In other words, the executable run on every networked device must be logically identical. The specific behavior of the game application is configurable and toggleable both locally, through configuration files and arguments, and remotely, by the deployment module. This is the last requirement for online games to integrate Polka, addressing NFR3.

Configuring Differentiated Deployment

Traditional online-game deployment is a simple process. Game operators making deployment decisions need only decide on what hardware to host the monolithic online-game server. Polka allows game operators to specify game operation policies to control differentiated deployment and dynamic service migration, addressing FR1. However, by giving game operators more fine-grained control, differentiated deployment increases system performance and flexibility, but comes at the cost of complexity. Game operators must configure not only networked hardware resources, but choose or develop a suitable differentiated deployment policy tailored for the game they deploy.

Complex distributed systems such as online games have serious risk of system failures caused by misconfiguration [59]. To reduce this risk and and the complexity that differentiated deployment introduces, Polka includes a multi-tiered configuration component (❷). The first tier is a *basic* configuration interface, which exposes only a limited set of configuration parameters, and requires only necessary information on network resources and game components. In the basic configuration mode, game operators can select from a set of simple service operation policies, which support only general-purpose deployment, monitoring, and migration of game services.

For more advanced use-cases that require game-specific information or complex service operation policies, the *advanced* configuration interface can be used. This interface exposes all parameters of Polka, and allowing operators to run custom service operation policies taking into account game-specific performance metrics or hardware events. The advanced configuration interface increases the possibility of misconfiguration, custom policies could introduce unforeseen behavior, and even result in system failures.

3.1.4. Automated Service Migration and Policies

After the service deployment component has successfully performed the initial deployment of game services, the *service monitoring component* (❹) is responsible for monitoring the online-game system and taking actions according to a *service operation policy.*

Distributed Service Monitoring

Monitoring the performance of the online game during runtime requires performing distributed performance analysis. Existing performance monitoring tools for online games report system-wide aggregate performance values over a long period, which is unsuitable for making service migration decisions. To monitor performance at a sufficient rate in an unreliable, heterogeneous networked environment (addressing FR4), Polka's service monitoring component relies on a two-tiered reporting structure.

Local *performance monitoring components* (⓱) deployed on each networked node to collect a detailed view of application-specific metrics, such as the duration of game loop iterations and response times of specific players. The performance monitors of each nodes can take actions locally to reduce strain when perfor-

Action	Description
Service Control	Start or stop a service
Scale Node	Vertically scale a node up or down
Scale Service	Horizontally scale a node up or down
Migration	Migrate a service to a different node

Table 3.2: Service operation policy actions.

mance is affected. For instance, if the affected node is running only a thin client, it may reduce the bit rate of the video stream. This allows Polka to react quickly, partially addressing NFR2.

At a higher tier for monitoring performance across the whole system, local performance monitors send periodic performance logs to the Service Monitor, which uses them to make migration decisions.

Service Operation Policies

The service monitoring component performs actions to improve scalability and flexibility of an online game. The specific actions taken, and what situations trigger them, are specified through *service operation policies*.

A basic service operation policy is specified in the form of trigger to action pairs. The trigger of an action is a predicate statement based on the metric reports sent to the service monitoring component by the performance monitors. Some example, simple triggers are “response time above 50 milliseconds for over 5 seconds", or “frame time above 16.66 milliseconds with less than 20 players connected".

Table 3.2 shows the set of primitive actions that can be performed by the service monitoring component. The most basic action is service control, which simply starts or stops a service. This is useful for toggling replicated services, such as client-side services for performing client-side prediction.

Scaling nodes vertically is an action only available when Polka is given the ability to control the networked hardware manager, such as when Polka is deployed on a cloud computing provider. While this is less conceptually interesting from a differentiated deployment perspective, it would allow Polka to act as a holistic resource manager, rather than requiring game operators to configure both Polka and a separate hardware resource manager.

Horizontal scaling of services is an advanced functionality of Polka’s design that would requires specific game systems to be marked as horizontally scalable and given a specific horizontal scaling handler method. For instance, if a game system that handles player movement is marked as horizontally scalable by in-game environment regions, then an associated handler for horizontal scaling would need to instruct the deployment system to transfer player movement handling to the new player movement services based on where players currently are in the game environment.

Finally, service migration is the core mechanism for achieving dynamic and flexible online-game operation. It allows specific game services to be migrated between network nodes while limiting disruption to player experience. We discuss the service migration mechanism in detail in the following section.

3.1.5. Dynamic Migration of Game Services

Migrating active services while the online game is running is a complex problem. In this section, we outline our design approach to Polka’s service migration through the *service migration component* (❺).

Identifying Migration Candidates

The first major challenge for dynamic migration of game services is identifying both what game services would benefit from migration, and where these services can be migrated to. In the simplest case, game operators could directly instruct Polka what services can and should be migrated through service operation policy actions. In this scenario, Polka must still identify what networked resource to migrate these services to. Automatically determining deployment configurations that result in better performance addresses NFR1.

Finding promising candidate deployment targets requires both the current performance status of every network node and up-to-date information from the service graph. The service monitoring component continually aggregates performance data it receives from performance monitoring components, and adds these performance values to the service graph.

Candidate migration targets then must be ranked according to their current performance, and on the amount of communication cost incurred. The performance rank is determined as a system health metric: has the node had stable and high performance for a sufficient duration. Communication cost is a function of service graph operations, if a service is migrated to this node, what new write edges are added to (or removed from) the service graph?

When the service migration component has completed a ranking of candidate targets, there are three potential results. In the best case, there is a healthy node with sufficient capacity and low communication cost. Such a target is a an easy choice as a migration target. In a worse case, there are no healthy nodes available, but Polka has been configured with the ability to start new hardware nodes, which can act as migration deployment targets. These new hardware nodes may also act as deployment targets for future service migrations, though care must be taken to limit when Polka would deploy new nodes to avoid continually deploying them. In the worst case, Polka cannot deploy new nodes and there are no healthy migration targets. In this scenario, Polka should either decline migration to avoid worsening performance further, or attempt a migration to an unhealthy node that has a low, or negative, communication cost.

Automatically determining services to migrate is an advanced functionality. Per-system performance data is not sent to the service monitoring component by performance monitors. Instead, the performance monitor must self-report *migration requests* to the service monitor. These requests should be sent when the local performance monitor finds systems that may benefit from migration. Migration feasibility is judged by the percent of processing time spent on individual systems. However, performance monitors should delay sending a migration request to ensure that poor performance of the system is not temporary.

When the service monitoring component receives a migration request, it will perform the candidate deployment migration target ranking as described above. If a suitable candidate is found, the migration process is performed. If one is not, the service monitor responds with a negative to the performance monitor which sent the request. This negative response can act to temporarily suppress migration requests for that system.

Synchronizing Data Dependencies

Migrating a service from a source to a target network node requires synchronizing the data that the service operates on. This process requires identifying what data to synchronize, where to get it from, and how to handle maintaining data consistency during the migration process.

Determining what data to synchronize for a given service is possible due to the service graph. The service graph includes information on the data dependencies of all services and keeps track of what services currently modify that data. In the general case, when moving a service to a target node, the data needed for that service is already present on the source node. Thus, synchronizing data is a simple process involving directly sending data from the source to target.

More complex cases occur when a service is either split into or combined from multiple services. Data must be assembled piece-wise from different sources, with duplicate or out-of-date data discarded. Again, this process relies heavily on the service graph informing the service migration component of data locations.

The sequence of service migration may take a considerable amount of time due to synchronizing data. During this time, that data may be changed or updated. To ensure consistency during and after the migration process, the first step of the synchronization process is to mark the target node as a subscriber for updates to those game state types. The target node keeps these received updates in an ordered buffer, and applies them to the synchronized data after it has been completely received. If the migration process fails, the target should discard this buffer and be removed as a subscriber.

As a consequence of the synchronization process, it is a good engineering practice to *minimize the data dependencies of latency critical services.* When such services must be migrated, the duration of the migration process is reduced from needing to synchronize a smaller amount of data.

Service Migration Sequence

In Figure 3.3 we show the messages exchanged between various Polka components to trigger and execute a service migration between networked nodes. In the pictured scenario, the game operator has configured Polka with a service operation policy to migrate service X based on a performance trigger on Node 1.

When the service monitoring component detects this trigger, it begins the migration process by querying the up to date service graph. It then performs candidate target ranking as described in section 3.1.5.1. In this case, Node 2 is selected as a target. The service monitor then informs the service graph of this decision by sending a migration started notice. This subscribes Node 2 to updates to the data dependencies of service X. Finally, the service monitoring component instantiates a service migration handler, and hands off control of the migration process to it.

The service migration component begins the service migration sequence by informing Node 2 of the migration and starting an instance of service X on Node 2. Service X is started in an awaiting synchronization mode, and is storing updates to data dependencies of service X in a buffer. Once the data from Node 1

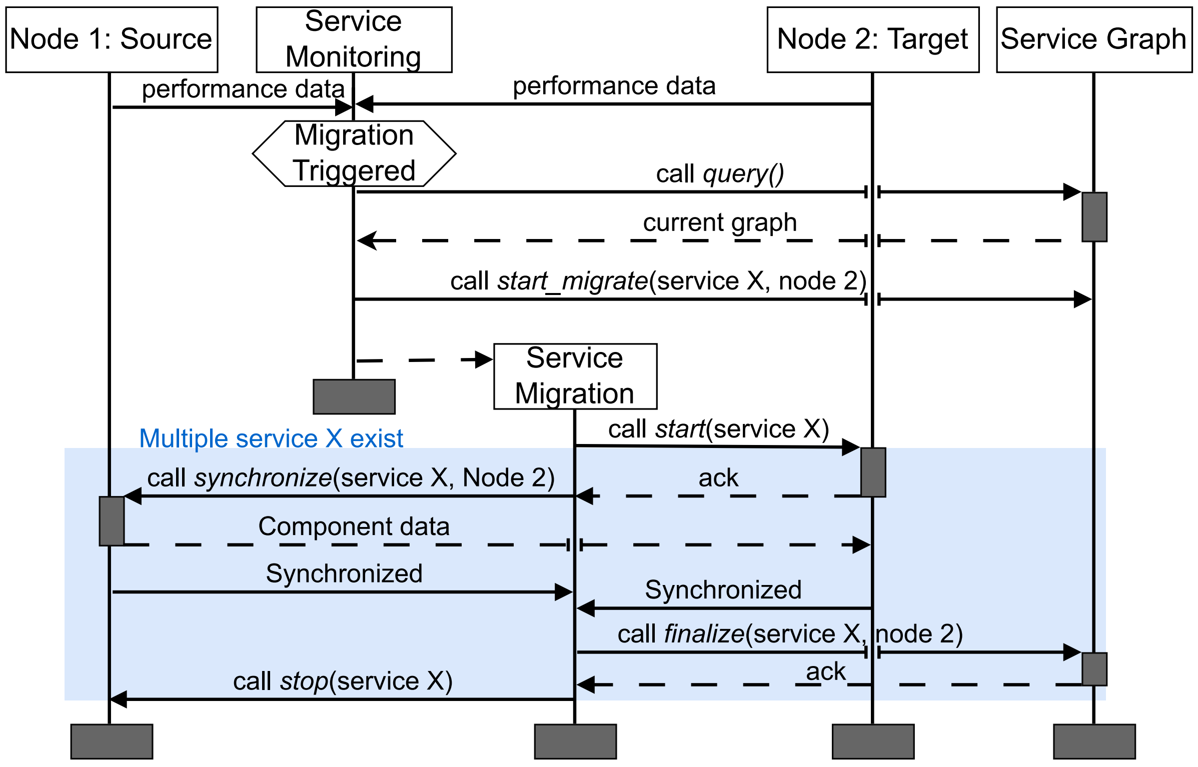


Figure 3.3: Polka service migration message sequence chart.

has been synchronized with Node 2 and both nodes have informed the service migration component of this event, it sends a finalized migration notice to the service graph.

Upon receiving this notice, the service graph removes service X from Node 1 in the graph and informs nodes of the change in location. Once the service migration manager has received acknowledgment from the service graph, it stops service X on Node 1, and the migration sequence is complete.

3.2. PolkaDOTS: an Implementation of Polka in the Unity Ecosystem

In this section, we address RQ2 with a detailed description of how we implement a proof of concept of the Polka design using tools from *Unity*, a state-of-the-art game development ecosystem. We call this implementation PolkaDOTS. We then further describe the steps necessary to utilize and integrate PolkaDOTS when designing and implementing an online game.

3.2.1. The Unity Ecosystem

There are a large variety of publicly available game development frameworks. These frameworks, at their core, offer a fully-featured game engine and a packaging system to deploy games built on that engine. Typically, they also feature a user-friendly graphical editor. Several of these frameworks have achieved a high adoption rate, with many game developers choosing to build their products on them. Some key examples include Godot, Unreal Engine, and GameMaker Studio, but the one we target for this work is Unity.

There are several reasons why we pick Unity. First, Unity is an industry-standard framework. Unity has been used for massively popular online games, including Pokemon Go, Rust, and Hearthstone [15]. In fact over 70% of the most downloaded mobile games are built on Unity [49]. Second, Unity has a low barrier of entry: Unity's features are well documented both in official and online-community sources, and the framework's package manager allows a simple, easy to learn core feature set to be extended with more advanced functionality. Finally, Unity provides more than only an engine and packaging framework. Unity is a modern online-game ecosystem, and offers a cohesive suite of online-game services in the form of Unity Gaming Services (UGS). Under the UGS umbrella there are services for both development and operation of online games. During development, developers can make use of an asset store to purchase or sell game assets, utilize an integrated version control system, and create a build-automation and continuous integration pipeline. Online

Table 3.3: DOTS key technologies.

Technology	**Description**
ECS for Unity	Implementation of an entity-component system (ECS).
Burst Compiler	Vector optimized LLVM compiler for .NET/C#.
C# Job System	Entity-component system (ECS) compatible multi-threaded task system.
Netcode for Entities (NFE)	ECS compatible modern Netcode.

games published with Unity can utilize online services for user authentication, matchmaking, voice chat, asset delivery, and secure in-game economies. This makes Unity a good candidate as a target framework and addresses FR3.

3.2.2. Data-oriented technology stack (DOTS)

The DOTS is a feature of Unity that allows game developers to build games using data-oriented programming (DOP). In Table 3.3 we describe the key technologies that comprise the DOTS.

The DOTS operates alongside Unity's more traditional object-oriented programming (OOP) system called *GameObjects*, thus allowing for a hybrid OOP and DOP programming model. This hybrid approach allows programmers to trade-off between complexity and performance for specific game features. Separation of OOP and DOP is maintained through the use of *entity worlds*.

An entity world is an isolated set of systems and entities. Each system in an entity world operates only on entities in the same world. Entities can be added to entity worlds either through systems in that world, or through a *baking system*, which convert regular OOP GameObjects into an entity representation.

PolkaDOTS makes extensive use of all the DOTS key technologies, but Netcode for Entities (NFE) is of particular interest as we use and extend it for communication between PolkaDOTS components. NFE is a netcode library for Unity ECS. NFE is a server-authoritative state-interpolation netcode, but it supports out-of-the-box a variety of advanced netcode techniques, including client-side prediction and server-side lag-compensation (see Section 2.3.1). It includes configurable inconsistency and interest-management policies which allow developers to set snapshot size limits and dynamically specify player interest per-entity. NFE uses entity worlds for many of its features, such as to separate client and server worlds when running in a listen-server deployment, for asset streaming, and for integrating with Unity's physics system.

3.2.3. Other Unity packages

Aside from the DOTS, there are several important Unity packages used in PolkaDOTS. The first is the *Unity Input System* [48] package. This package is the modernized replacement for the existing built-in Unity Input Manager. Aside from supporting more input devices and better integration with mobile devices, the Input System provides an undocumented feature of the ability to record and playback inputs [46]. This feature is beneficial for performance evaluation using PolkaDOTS as it allows player emulation on thin-clients.

The second is the *Unity Render Streaming* package, which uses a Unity implementation of WebRTC to allow Unity projects to send or receive streams of video, audio, and inputs. By using the Render Streaming package, PolkaDOTS's client networking component (⓮ in Figure 3.1) is compatible with any application supporting WebRTC, meaning, for example, that most modern browsers can act as an PolkaDOTS thin-client. The Render Streaming package supports multiple concurrent stream connections, allowing PolkaDOTS to support the Polka feature of de-coupling players and rendering components: a single PolkaDOTS client can independently stream rendered frames to multiple thin clients.

3.2.4. System modifications and integration

To integrate into the Unity ecosystem, PolkaDOTS makes use of many existing Unity features and packages. Many of these preexisting systems were not designed to interface with each other. The two packages outside of the DOTS, Unity Render Streaming and the Unity Input System are particularly challenging to reconcile. These two systems are designed to operate on OOP GameObjects, and do not support ECS out of the box.

In Figure 3.4 we show the Unity components we use in PolkaDOTS and how they are connected. Unity components we extend are marked with a green plus, and components we create are marked in blue. Components that bridge the operation of OOP and DOP systems are marked as interfaces. Both the Input System

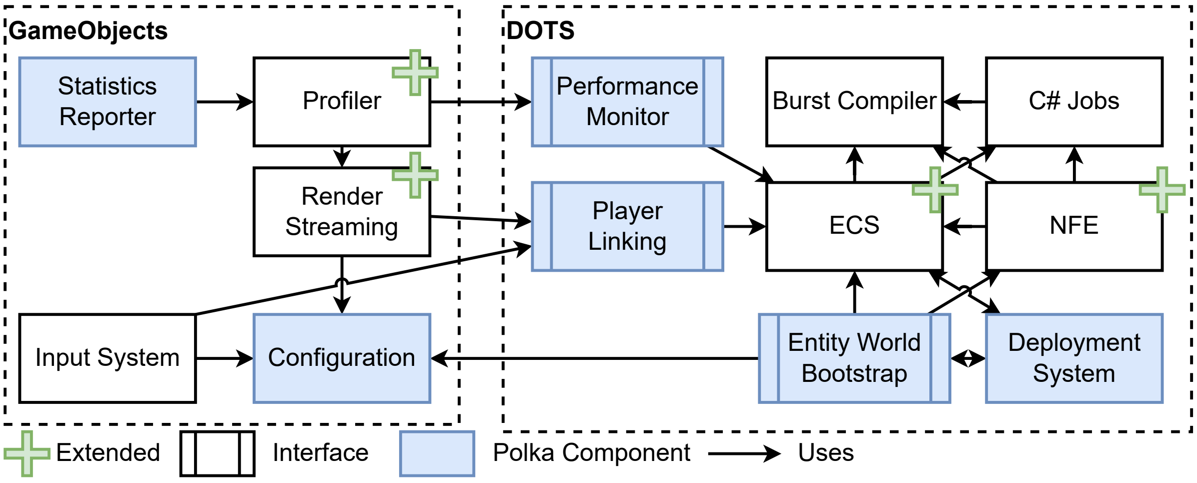


Figure 3.4: Unity package integration with Polka components.

and Render Streaming packages use the concept of a *player object*: a single object that represents the in-game position and avatar of the local player. These packages expect that the player object has an associated behavior to collect and handle player inputs, and has as an associated camera object responsible for rendering a view of the game from the players position. In ECS and NFE, following their DOP paradigm, there are no player objects. Instead, each player in the game is represented as an entity and identified through association with specific components. Handling player input is then performed in two steps, gathering input from all players, and then applying the corresponding results to each player. Polka provides several premade Player GameObject templates and handles instantiating them, but developers must handle linking them to new player entities (see section 3.2.6).

The PolkaDOTS Entity World Bootstrap and the Deployment System are additions to the DOTS that enable Polka functionalities. The *Entity World Bootstrap* is responsible for creating and managing entity worlds. When the application is started, the Bootstrap immediately creates an entity world responsible for running the *Deployment System*. Depending on configuration, the deployment system either acts as a deployment host, or attempts to connect and request deployment configuration from a remote deployment host. In both cases, the deployment system then instructs the bootstrap to create and start running game entity worlds depending on the received configuration. The deployment system remains active during the game's runtime, and can issue instructions to the bootstrap to create new entity worlds or modify the state of existing ones. It is through a sequence of instructions issued from the deployment system that PolkaDOTS performs service migrations.

To enable the operation of the bootstrap and the deployment system, several changes are made to ECS and NFE. Where possible, changes to existing Unity packages are implemented as Extension Packages, which are added to PolkaDOTS as dependencies. This decouples PolkaDOTS from the DOTS packages, which are likely to be continuously updated. We use an extension package to override the default ECS entity world bootstrapping logic, giving the PolkaDOTS bootstrap fine-grained control of what game systems are run in each entity world. Similarly, we extend the default NFE entity world handling to add new networked entity world types, such as the world for running the deployment system. Additionally, we add several mechanisms to allow the PolkaDOTS bootstrap to monitor world state, and to instruct specific entity worlds to attempt connection or listen to specific network endpoints.

Performance is monitored in PolkaDOTS using extensions to the Unity Profiler for PolkaDOTS and game statistics (see section 3.2.5) as well as a game service to aggregate PolkaDOTS statistics and report them to the Unity Profiler. Such a service is necessary to capture several metrics in existing packages that are not directly reported by the Unity Profiler, including NFE network statistics, Render Streaming network statistics, and any game-specific metrics.

3.2.5. Performance monitoring

In PolkaDOTS, the Polka performance monitor component is implemented as an extension to Unity's Profiler module. The profiler collects a wide range of system and application metrics. We list these metrics in Table 3.4. Metrics we add to the Unity profiler are marked in bold. Online games built on PolkaDOTS can

Table 3.4: Profiler Metrics collected by Opencraft 2. The metric type is Application level, System level, or Network. Extensions we add to the Profiler are marked in bold.

Type	Metric	Description	CSV Writable
A	Game FPS	Rendered frames per second	Yes
A	**Stream FPS**	**Rendered stream frames per second**	**Yes**
A	Triangles	Rendered triangles per frame	No
A	Entities	Entity count and modification events	No
S	CPU Usage	Frame CPU time, split by category	Yes
S	Stack Trace	Frame function and thread stack	No
S	Memory	Memory usage, split by category	Yes
S	Threads	Thread total	No
S	File I/O	Files read/written	Yes
N	**Stream round-trip time (RTT)**	**Round trip time of render streaming**	**Yes**
N	**Stream BitRate**	**BitRate of render stream**	**Yes**
N	**Stream Packet Loss**	**Dropped packets of each render stream**	**Yes**
N	**Game RTT**	**Round trip time of game connection**	**Yes**
N	**Game Jitter**	**Jitter of game connection**	**Yes**
N	**Network Tick**	**The current network tick number**	**Yes**

further extend the Unity profiler with game-specific metrics through *profiler modules*.

PolkaDOTS performance metrics can be monitored in several ways, both online and offline. Using the Unity Editor, remote instances of PolkaDOTS can be monitored online directly using the Unity Profiler's remote connection feature. PolkaDOTS can be configured to serialize performance data to a binary file, which can be loaded and inspected using the Unity Editor. Use of this binary file alone is not recommended: these files can be quite large, and can only be analyzed within the Unity Editor in ranges of 2000 frames at a time. To allow analysis with other tools, including our own visualization tools, we package with PolkaDOTS an Unity Editor extension that converts multiple performance log files to a comma-separated value (CSV) format. The conversion process has downsides: it is incompatible with any game-specific profiler modules, it requires access to a version of the Unity Editor equivalent to the version used to compile the PolkaDOTS application, the conversion process is quite slow, and relying on conversion does not reduce the storage, computational, and network requirements for handling performance logs. A more practical method of getting performance metrics in a CSV format is having PolkaDOTS record a subset of performance metrics directly to CSV during runtime. The metrics for which this is possible are indicated in the rightmost column of Table 3.4.

3.2.6. Using PolkaDOTS

PolkaDOTS is a framework for online games. Making a game on the PolkaDOTS framework requires implementing certain required components and satisfying a set of restraints. In this subsection, we outline the programmatic interface of PolkaDOTS and enumerate what requirements and assumptions the framework imposes.

PolkaDOTS feature support requires that game systems be compatible with the Unity DOTS interface. In short, game systems must be implemented as a Unity ECS *ISystem* or *SystemBase*, and all game data must be organized into *IComponentData* or *IBufferElementData* structures. However, PolkaDOTS does not require full interoperability with the Burst Compiler or C# Jobs, though support of these features will improve online-game performance.

PolkaDOTS does not restrict DOTS hybrid functionality, GameObjects and other OOPbehaviors can be used. However, developers must be aware that these *GameObject behaviors will run on all PolkaDOTS services*, and program such behaviors accordingly. Ensuring correct behavior may require adding additional interface services to PolkaDOTS's DOTS extensions.

An important restriction that PolkaDOTS introduces is that behaviors can no longer assume there is only a single local player. Because clients can host rendering components of multiple thin-clients, multiple players may be handled by a single client. To facilitate this, PolkaDOTS includes two template player GameObjects, or *prefabs*. The *LocalPlayer* prefab reads input from connected hardware, a input recording file, or a remote guest player connected through WebRTC. It then renders the player's point of view either to the locally connected screen or streamed to the remote guest player's device. The *GuestPlayer* prefab is then responsible for sending player inputs to a hosting client, and receiving and displaying the resulting video stream. PolkaDOTS handles instantiation and connection of these two symmetrical types of Player objects. However, it does not define or create Player entities. Therefore, the developer is responsible for including systems to instantiate Player entities and subsequently read input for them from the corresponding Player object.

As mentioned in section 3.2.5, any game-specific metrics that a game developer wants to expose to the performance monitoring and logging system must be implemented as an extension to the Unity Profiler. This requires creating a *Profiler module*, which records per-frame the value of a set of performance metric (see Section 4.2.4 for an example). Collecting and registering these metrics each frame must be performed by systems within the online game.

A final restriction for games built on PolkaDOTS is the requirement of polymorphism. As outlined in section 3.1.3.2, the executable artifact for the online game must a singular, and be adaptable to different components through local and remote configuration. By default, PolkaDOTS handles polymorphism through starting the application in a bootstrap mode and starting necessary entity worlds according to local or remote configuration. The entity world types supported out-of-the-box are:

1. **Deployment Server**: Responsible for creating and managing the service graph, and responding to deployment requests.
2. **Deployment Client**: Requests deployment configuration from a deployment server.
3. **Game Server**: Performs online-game simulation, accepts connections from game clients and synchronizes game state with them.
4. **Game Client**: Performs replicated game simulation, connects to a game server. Renders frames for one local player and optionally additional remote clients.
5. **Cloud Host Client**: Same as a game client, but does not have a local rendering service, instead running rendering only for remote clients.
6. **Remote Client**: Includes only a thin client that streams frames from a game or cloud host client.

Any additional entity world types required for the online game must be added by the developer.

4
Opencraft 2: Minecraft-like Game (MLG) Built on PolkaDOTS

MLGs are a genre of online game where players can modify almost all parts of the game environment. MLGs are extremely popular, with the canonical example of an MLG, Minecraft, being the best-selling game of all time [64]. They are also a performance engineering challenge, modern MLGs support only a few hundred players under optimal conditions, and experience frequent and severe performance variation under typical workload [20, 56]. These properties make MLGs a good genre for evaluating Polka.

In order to evaluate Polka's differentiated deployment mechanism on a MLG, we must either to integrate PolkaDOTS into an existing MLG, or develop a new MLG on top of it. Polka is implemented on Unity because it is an industry standard game development ecosystem. However, there are no open-source MLGs built on state-of-the-art tools such as Unity that meet our requirements. Therefore, we design and build such a MLG part of this project.

In this chapter, we design and implement Opencraft 2, a representative MLG built on the PolkaDOTS framework. Finally, we discuss how Opencraft 2 enables future work by acting as a research platform for differentiated deployment, MLGs, and online games in general.

4.1. Design

In this section we outline the design requirements of Opencraft 2 and how they are met by our proposed architecture.

4.1.1. Requirements Analysis

Opencraft 2 has requirements imposed by PolkaDOTS, requirements to be considered a representative MLG, and requirements to be usable as a research platform in future work. Additionally, there are several "good to have" properties that improve Opencraft 2 representativeness and ease of use. We show a list of these **G**ame **R**equirements in Table 4.1. The first 9 game requirements (**GR1** to **GR9**) we cover with our design and implementation of Opencraft 2. We outline how Opencraft 2 covers the last 4 game requirements (**GR10** to **GR13**) relating to supporting research in section 4.3.

Requirements for a MLG

In order for our evaluation to be generalizable, the behavior and performance patterns of Opencraft 2 must be representative of existing MLGs. Therefore, we first identify the properties of MLGs. The core feature of MLGs that separates them as a unique genre is the use of a modifiable virtual environment (MVE). A MVE can be freely and extensively modified by players of the MLG. Thus, supporting a MVE is a necessary requirement (**GR1**). Additionally, MLGs allow real-time interaction not only between players, but between players and the MVE. Opencraft 2 must then support real-time interactions between players and terrain (**GR2**). A final necessary requirement for Opencraft 2 to be considered a representative MLG is the use of procedural content generation (PCG). In MLGs, it is common to use PCG to generate new areas of the MVE on-demand. While it is not necessary for an online game to use PCG to be considered a MLG, it is a popular technique and

Table 4.1: List of **G**ame **R**equirements for Opencraft 2. Necessity is given as either **N**ecessary or **G**ood to have

	Source	**Description**	**Necessity**
GR1	MLG	Modifiable terrain	**N**
GR2		Real-time interaction	**N**
GR3		Procedural-content generation	**N**
GR4		Non-player characters	**G**
GR5		Dynamic terrain	**G**
GR6	PolkaDOTS	Built on Unity DOTS	**N**
GR7		Multiple players per client	**N**
GR8		Polymorphic application	**N**
GR9		Game-specific metrics	**G**
GR10	Research	Extensible game features	**N**
GR11		Configurable workload	**N**
GR12		Reproducibility	**N**
GR13		Documentation	**N**

strongly impacts the performance and behavior of the online game. Therefore, Opencraft 2 must support a PCG technique approximating those found in popular MLGs (**GR3**).

The use of a MVE, real time interaction, and PCG are all core features of MLGs. However, there are additional properties that while do effect the performance of MLGs, are not as essential to being considered a MLG. NPCs are a form of simulated entity that exist in a virtual environment but are not part of the terrain. NPCs exist in a wide variety of online-game genres, not just MLGs. However, in MLGs, their performance impact is more significant as NPC behavior must be continuously computed using the current state of a changing MVE. This makes support of NPCs a good-to-have feature (**GR4**). In MLGs, it is not only NPCs that are continuously simulated. Many MLGs include dynamic terrain behavior, wherein the MVE changes over time without direct player input. The extent and types of dynamic terrain behavior vary significantly between individual MLGs. This makes dynamic terrain a good-to-have feature (**GR5**).

Requirements imposed by PolkaDOTS

The PolkaDOTS framework imposes several restraints on online games built on it. These requirements are crucial to the operation of PolkaDOTS's deployment and render streaming mechanisms. A base requirement for Opencraft 2 to be built on PolkaDOTS is using Unity DOTS (**GR6**). This entails splitting game logic into systems, components, and entities. A core feature of Polka's design is allowing dynamic deployment of the rendering component. Therefore, Opencraft 2 must support multiple players, both remote and local, on each client (**GR7**). Finally, PolkaDOTS requires Opencraft 2 to be packaged as a polymorphic but singular executable (**GR8**), this allows PolkaDOTS's deployment mechanism to enable and disable services on each game instance as needed. While it is not necessary, it is a good-to-have requirement to include game-specific metrics in Opencraft 2 (**GR9**) in order to demonstrate how such metrics can be added to PolkaDOTS.

Requirements for a research platform

We include the design and implementation of Opencraft 2 in this work as we find no suitable existing open-source MLGs built in a modern game development ecosystem. Therefore, Opencraft 2 is intended not only as an evaluation artifact for PolkaDOTS, but also to fill this gap by acting as a research platform. In order to fulfill this role, there are several crucial requirements for Opencraft 2. First, Opencraft 2 must extensible (**GR10**). This requirement entails the ability for future researchers to add to or modify existing features of Opencraft 2. A related requirement is configurability of Opencraft 2 workload (**GR11**). Configurable workload facilitates experiment design, and allows researchers to fine-tune Opencraft 2 to better approximate existing MLGs. Of course, Opencraft 2 must be valid as an experiment artifact, and support reproducible experiments (**GR12**). Finally, an extensible and configurable system needs suitable documentation (**GR13**). Documentation is necessary both to explain how the system can be configured and extended, but also to present the design decisions that lead to the existing system. Without suitable documentation, the practical use of Opencraft 2 as a research platform is limited by the difficulty of adoption.

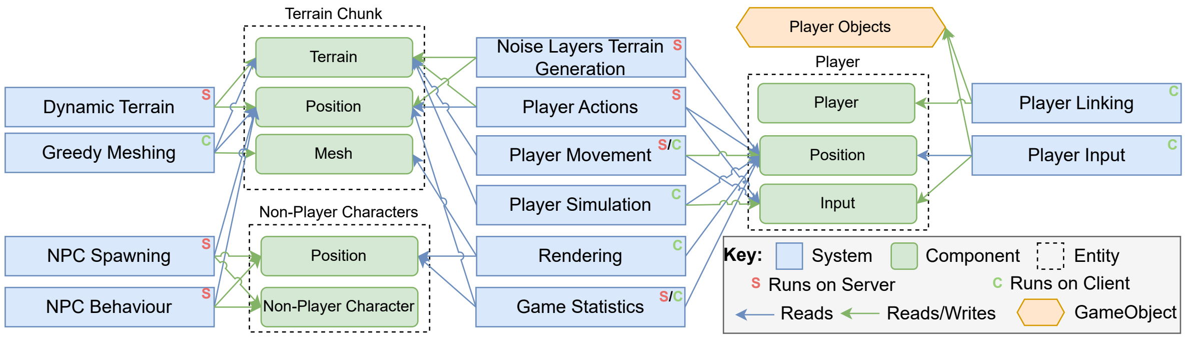


Figure 4.1: Design of Opencraft 2 expressed as relations between systems and entities.

4.1.2. Design Overview of Opencraft 2

In Figure 4.1 we show the the design of Opencraft 2. To meet **GR6**, we design Opencraft 2 in the context of Unity DOTS (see section 3.2.2). Therefore, in this figure we depict Opencraft 2's design using Unity ECS concepts. All game behaviours are expressed as *systems* that act on *components*, with groups of components corresponding to different *entities*.

To satisfy **GR1**, Opencraft 2 includes a MVE. We utilize a MVE design that splits the environment into discrete *chunks*. The MVE is created on-demand using PCG through a terrain generation system, satisfying **GR3**. We explain the details of Opencraft 2's MVE design and PCG in section 4.1.3.

Real-time interaction (**GR2**) is handled by several systems. The player movement system allows players to move and see other players moving, while the player actions system handles direct interactions between players and terrain. For instance, certain player actions can modify the MVE. We present a detailed view of the real-time interactions systems alongside Opencraft 2's netcode approach in section 4.1.4.

The good-to-have requirements of NPCs and dynamic terrain (**GR4** and **GR5**) are included as three systems. Dynamic terrain is simulated by a single system, that only needs to read and write to terrain entity components. NPC behavior is split into two systems, a *spawning* system that determines where to place NPCs in the MVE, and a NPC behavior system that controls NPCs once they have been spawned.

In order to support PolkaDOTS's multiple players per client requirement (**GR7**), Opencraft 2 uses a player linking system to handle creating player entities as needed and linking them to player objects creates by PolkaDOTS. The player input system them uses the links to collect player inputs from the player object (see section 3.2.4), and register them to the input component on the player entity. Additional game specific metrics (**GR9**) are collected by a game statistics system that queries entity data.

To satisfy the requirement that Opencraft 2 is packaged as a polymorphic singular executable (**GR8**), all systems are included in the packaged executable. The separation of server and client logic is handled by the PolkaDOTS deployment system adding or removing entity worlds. We discuss the distinction between server and client systems further in section 4.1.4.

Towards supporting both configurable workload (**GR11**) and reproducibility (**GR12**), Opencraft 2 includes a player simulation system. Simulated players generate reproducible inputs and can use player behavior models to approximate the inputs of real players. Notably, we specify player simulation as a client-only system. Therefore, the server systems will not be able to discern a difference between real or simulated players inputs, and will handle them identically.

4.1.3. Modifiable Virtual Environment (MVE)

A MVE is a core feature of MLGs. However, there is significant diversity in the design of MVEs that can significantly alter both the experience of the online game, and its performance characteristics. For example, MVEs differ in their use and type of PCG, and dynamic terrain behavior. A crucial distinction between MVEs is their approach to dividing the virtual environment. MLGs such as *Space Engineers* and *Deep Rock Galactic* use a continuous coordinate system, where the environment consists of arbitrary polygons that can freely shaped by players. Other MLGs such as *Minecraft* or *Eco* utilize discrete coordinates in a *voxel engine*, where terrain is divided into cuboids (*voxels*) of identical volume. Discrete and continuous MVE approaches differ both in how they are simulated and how they are graphically represented to players. In the remainder of this subsection, we outline the decisions and trade-offs made in Opencraft 2's MVE design.

The voxel engine

In Opencraft 2, we choose a voxel engine based approach to dividing and representing the MVE. Compared to a continuous, polygon-based implementation, there are several advantages. The discrete voxel engine method is used in *Minecraft* itself, the canonical example of a MLG. This allows Opencraft 2's MVE to be representative of the most popular MVE in the market today. Similarly, because of *Minecraft*'s popularity, there has been extensive development and refinement of voxel engine techniques, leading to many known and widely-utilized optimizations. These techniques both make the design and implementation of Opencraft 2's MVE less complex, and ensure that its behavior approximates MVE behavior in existing voxel-engine based MLGs. For instance, there are established techniques for performing PCG in voxel-engine MVEs.

Our MVE design uses a core optimization ubiquitous to voxel engines: terrain *chunks*. While the entire environment is split into discrete voxels, sets of nearby voxels are grouped into chunks. Each chunk is handled as a single unit in terms of storage in memory and processing by game systems. The terrain chunk technique decreases memory use significantly but comes at the cost of parallelism. Each terrain chunk can only be modified by a single thread at a time. Therefore, the size and structure of terrain chunks has a significant impact on voxel-engine based MVE performance.

Procedural content generation (PCG)

PCG is common feature of MLGs, allowing game developers to create expansive and detailed environments without manually positioning objects. Minecraft itself supports generating environments with up to 1.152×10^{12} voxels. Voxel engine MLGs in particular make extensive use of PCG, utilizing properties of the voxel environment to simplify generation. For instance, since all voxels have a discrete position and identical size, each voxel is neighbored by only a finite amount of other voxels (6 for cuboid voxels), and each neighbor voxel position can be calculated with a fixed offset. This property makes PCG significantly less complex for voxel engines: the PCG process is reduced to deciding if a voxel exists in a specific location, and what type of voxel it is. Consequently, there are a variety of existing and well-utilized PCG algorithms for voxel engines.

Opencraft 2 uses *noise layers* voxel engine PCG algorithm. Noise layers can be used in voxel engines with the terrain chunk abstraction. Each chunk is generated independently, using multiple layered pseudo-random noise maps. Using different combinations of noise layers, generated terrain can be highly customized. The primary benefits of the noise layers algorithm are determinism and parallelism. Noise maps are sampled using a *seed* value, the same seed will always result in the same terrain. Each chunk does not need any data from surrounding chunks, and can thus be generated in parallel.

Rendering

Counter-intuitively, voxel engines using PCG can be taxing to 3D graphics render pipelines. As opposed to games without PCG, data used to render the game world must be generated and updated during runtime. For the rendering pipeline to display these newly generated voxels in rendered frames, voxel terrain must first be converted to a format compatible with the rendering pipeline, a process called *meshing*. Several meshing techniques exist, our design uses the *greedy meshing* technique, which uses properties of the voxel environment as heuristics to group multiple voxels into a single mesh face. We explain greedy meshing in detail in Section 4.2.1.3.

4.1.4. Opencraft 2 Netcode

An online game's netcode determines what, when, and how game content is distributed (see section 2.3). The Opencraft 2 design makes uses of the netcode library integrated in the PolkaDOTS framework (NFE). Therefore, the netcode-related design decisions made for Opencraft 2, or any other game built on the PolkaDOTS framework, are in terms of game state dependencies and assigning game systems to PolkaDOTS-supported entity worlds. In Unity ECS, all game systems must explicitly list all game state types they access, and all game state types they modify. In traditional ECS this information is used to perform optimal multi-threaded scheduling, allowing many game systems to execute concurrently. In the PolkaDOTS framework, game state dependencies are still used for multi-threaded scheduling, but also to inform deployment and migration decisions through the use of the service graph (see section 3.1.3.1). Therefore, Opencraft 2's system game state dependencies directly affect netcode through specifying what and where game state is synchronized.

PolkaDOTS's deployment mechanism requires game systems to be assigned to one or more entity worlds. Performing separation of systems in this manner allows Opencraft 2 as a whole to be packaged and operated as a single polymorphic application, satisfying **GR8**. The two most important worlds in the context of

Opencraft 2 are the game server and game client entity worlds. Deciding the initial system placement between server and client strongly impacts netcode behavior. For example, the two player interaction systems are assigned to entity worlds differently. The player actions system is run only on the server entity world, and player movement is run on both server and client entity worlds. Therefore, player actions will not be subject to client-side prediction, and will instead be interpolated. Conversely, player movement will be predicted by the client, decreasing the response time of movement but potentially leading to mispredictions.

The design decision to predict movement but not actions is made based on misprediction cost and game state dependencies. The player movement system reads player input, player position, and terrain, then modifies player position. Player input is collected by the player input system on clients, thus clients have authority over player input. Mispredictions of player movement can then only occur when clients and servers do not have the same terrain state. While such a situation is expected to occur, the cost of a misprediction is expected to be small: the player position will simply interpolated to the position the server computed. By contrast, the player action system reads player input and player position, then modifies terrain. The client again has authority of player input, but now the cost of mispredictions is much higher. When terrain changes, such as through player actions, the terrain must be processed again by the meshing system to update its visual representation. When a misprediction occurs, the client would first predict a terrain change, and perform meshing. Then, when the correct state is received, the terrain would be changed again, and meshing performed once more.

Since Opencraft 2 is a MLG, an important netcode design decision is determining how terrain is synchronized. Simplifying synchronization is another motivating reason that voxel engines typically utilize a chunk abstraction: synchronization can be performed at a per-chunk granularity. While synchronizing an entire chunk is data intensive, its performance impact is reduced by the practice of delta encoding, in which an entire chunk to be synchronized once per player, then only modifications are synchronized. Additionally, in voxel engines chunks tend to be highly compressible due to strong locality: voxels of a given type tend to be close to voxels of the same type. As a trade-off, per-chunk state synchronization means that both the size of chunks, and the state synchronization frequency, have a strong impact on bandwidth usage and quality of experience.

4.2. Implementation

We implement our design of Opencraft 2, a representative MLG, on the PolkaDOTS framework. In this section, we explain how our design is actualized within Unity and PolkaDOTS, and the specific challenges encountered during the implementation process. We intend this section both to demonstrate the validity of Opencraft 2's design as a MLG, and to show the process and important steps when implementing an online game on the PolkaDOTS framework.

4.2.1. Implementing the MVE

The distinguishing feature of MLGs is a modifiable environment, referred to as a MVE. Opencraft 2's design uses a voxel engine based MVE, and adapts existing voxel engine optimization techniques for the PolkaDOTS framework stack.

When implementing any MLG, the primary performance concern of is volume. Individual environment components such as a single voxel may require only minimal processing and small amounts of memory, but MLGs typically have environments with millions of components. Minecraft itself supports environments with up to 1.152×10^{12} components. Managing such high-volume environments is not trivial, and Minecraft, as well as other voxel-engine MLGs, use many optimizations to make this complexity feasible.

Opencraft 2's voxel engine

The first implementation decision is about Opencraft 2's voxel engine. Specifically, we must determine how voxel terrain is stored and accessed. In Unity DOTS data-oriented ECS pattern, the simple, naive approach would represent each individual voxel as an entity. Such an approach does not scale well: quick random-access to specific voxels is crucial to MLG performance, but unsorted ECS entity representation provides only an $O(n)$ access time. Even worse, voxels that are spatially close in the environment are not guaranteed to be close in memory. Finally, because NFE snapshots are performed per-entity, having this large volume of voxel entities causes poor netcode performance.

Opencraft 2's design uses basic voxel engine optimization of grouping many adjacent voxels into a *chunk*. In the ECS pattern, hierarchical grouping of entities is discouraged, as it requires making a new entity to hold a component that contains only the entity IDs of child entities. ECS maintains a lookup table for entity IDs, so adding this chunk hierarchy to the naive ECS approach does marginally improve voxel random-access time. However, this system increases memory usage, and greatly complicates terrain generation, modification, and synchronization which now have to access and update both voxel and chunk entities independently.

To solve these issues, Opencraft 2 adapts the existing chunk optimization to ECS by not representing voxels as individual entities, but instead as having voxel terrain stored as one entity for each chunk. Each chunk entity is given two components, a chunk transform position, and a fixed-size flattened 3D byte array of voxels. This flattened array is sorted by voxel position, so accessing a voxel within a given chunk is $O(1)$. Chunks, being stored as entities, are not sorted. Therefore, random access of voxels is $O(m) + O(1)$, where m is the number of chunks.

To improve voxel random access time further, Opencraft 2 adds to the chunk entities a *chunk-neighbors component*. This component contains entity IDs of neighboring chunks. Unlike voxels, chunks are never removed. Thus, the neighbor connections need only be updated when a new chunk is generated. With this optimization, random access of voxels now has two modes: global or heuristic tree based. The global search iterates through all chunks as normal to read their chunk transform position ($O(m) + O(1)$). In tree search, a starting chunk is required, but we see an improvement to $O(log(m) + O(1))$, as we can search by traversing chunk neighbor links. The heuristic tree search is not suitable for all voxel search tasks, since if there happens to be two or more groups of chunks that are not connected by any neighbor links, the tree search will be confined to one group. Since terrain generation and modification actions tend to effect localized regions of the game world, a common search pattern is an initial global search followed by many subsequent tree searches starting from the chunk found by the global search.

Opencraft 2 exposes chunk size as a configurable parameter. Chunk size has a considerable effect on performance, but with several trade-offs. Larger chunk sizes are synchronized to clients with less NFE snapshots, and result in a faster block-access time from having to iterate through fewer chunks overall. However, larger chunk sizes take longer to generate, and to convert to a visible format on the client (e.g. *meshing*).

Opencraft 2 terrain generation

Opencraft 2 adapts the existing voxel engine PCG technique of *noise layers* to the PolkaDOTS stack. This algorithm works bottom-up, working upwards through the vertical positions in a column of chunks. Depending on the current vertical position, the presence and type of voxel is determined by a different set of noise layers, and their associated noise maps. A noise map refers to a flattened 2D array of floating point values, generated by sampling a pseudo-random value for each point and applying a noise smoothing algorithm. In Opencraft 2, both Perlin and Simplex noise smoothing algorithms are supported through a custom random noise library. Each noise map is associated to a noise layer, with the parameters of the noise sampling algorithm being specified by the noise layer configuration.

To control Opencraft 2's PCG process, the amount, types, and order of noise layers is configurable. Each noise layer has six parameters: the layer type, priority, the voxel type, frequency, minimum height, and maximum height. The minimum and maximum height specify the range of vertical positions the layer applies to. Frequency is a noise generation parameter, higher values tend to result in more voxels. The voxel type refers to what type of voxel will be created from this noise layer. Priority values specify what order layers are applied in when more than one layer apply to a vertical position. Layer type changes how the sampled noise is used to place voxels. Opencraft 2 supports four different noise layer types: *Absolute*, *Additive*, *Surface*, and *Structure*.

The absolute layer type adds a column of voxels starting and the minimum height value, and up to a value between the minimum height + 1 and the maximum height, if there are already voxels in any positions that the absolute layer would add to, those voxels are not replaced. The additive layer adds between 1 and maximum height - minimum height, but starts placing voxels at the current top of the column, rather than at the minimum height value. The surface layer is similar, but it only adds either 1 or 0 voxels to the top of the voxel column, regardless of minimum and maximum height values. The structure layer is a special case, it does not immediately change any voxels in the terrain column, but instead directly samples noise to generate a list of structures to spawn at associated locations per chunk. Structure layers do not used smoothed noise maps, instead they directly sample noise at a position to determine if a structure should be placed there. To ensure that structures do not overlap, structure layers also sample within a region around the current position, to determine if there are already structures nearby.

In typical voxel engine PCG, *structures* are considered distinct from terrain, even when used to add environment features to the terrain such as trees or caves. This distinction stems from the common implementation decision to generate structures in a separate pass after completing terrain generation. In Opencraft 2,

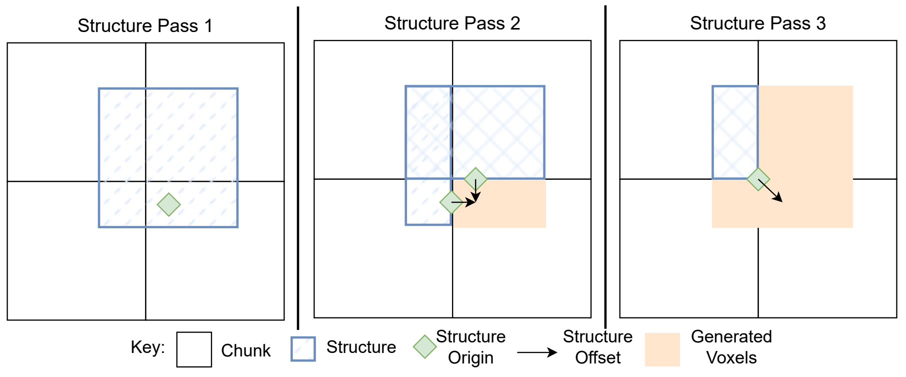


Figure 4.2: Opencraft 2 multi-pass structure generation.

we do precisely this, and extend the basic noise layer technique with a multi-pass structure generation algorithm. In the DOTS, a single-pass structure generation technique does not work without adaptation. This is because structures may cross borders between chunks, and thus require modifications to more than one entity at once, while each of those chunk entities are also modifying themselves. To avoid race conditions and undefined behavior while maintaining parallelism, Opencraft 2 uses multi-pass sub-structures process.

In Figure 4.2 we show an example of Opencraft 2 structure generation. In this process, each chunk has a list of structure components representing segments of a structure to generate. During each step, all structures in each chunk's list are processed: the region of the structure overlapping the current chunk is generated within the chunks voxel array. Then, the regions of the structure overlapping with neighboring chunks are added to those chunk's structure list as a sub-structure, to be processed in the next step. To enable the multi-pass structure technique, each structure must be generated deterministically, based upon a seed. In Opencraft 2, this seed is sampled per-structure by the structure noise layer. When structures are split into sub-structures, each sub-structure has the same noise value as the original, as well as an offset to keep track of the original starting point of the structure.

Opencraft 2's PCG is highly parallelizable, allowing both high performance and making the PCG services more suitable for differentiated deployment. However, achieving this requires several trade-offs. The noise layers algorithm requires terrain regions to be generated in vertical columns; individual chunks cannot be generated standalone. When chunk size or chunk column height increase, the computational load of performing terrain generation increases, and the benefits from parallelism decreases. Similarly, the noise layers algorithm is highly extensible: detailed terrain is achieved through the addition of more noise layers. However, each noise layer increases the amount of time to generate a chunk, even if it does not effect the resulting voxels. Structure layers in particular have a high performance impact as they perform multiple noise samples per position and require a variable number of passes during processing. Structures that have regions overlapping with neighboring chunks cannot be processed until all neighboring chunk has been generated. Similarly, the more neighboring chunks a structure overlaps, the more passes it requires, as demonstrated in Figure 4.2.

Opencraft 2 rendering voxels

Internally, voxels are stored as flattened arrays. For the rendering pipeline to display voxels in rendered frames, chunk voxel arrays must first be converted to a format compatible with the rendering pipeline, a process called *meshing*. Voxel engines decompose the game environment into discrete spaces. Therefore, the naive voxel engine chunk meshing process is a direct mapping: create a grid of vertices of the same width, depth, and height as a chunk. Then, iterate over the chunk's array: for each non-empty voxel create 6 faces, stored as 12 triangles. This technique is simple to implement and extremely parallelizable, but will result in very poor performance, especially on devices without dedicated graphics hardware.

The reason the naive meshing algorithm is insufficient is again a problem of volume. If we take a single chunk to have an edge length of 16, then each chunk will have 4096 vertices. Each of these vertices will be represented internally as three 32-bit floating-point values. Therefore, each chunk mesh vertex array will be 49.152kb, a manageable size for modern graphics. The issue comes from the faces, or more accurately the triangles. In the worst case, a single chunk can have 49152 triangles, but *only 1536 of these triangles can possibly be visible!* To put this into perspective consider the following scenario: a player has a screen with

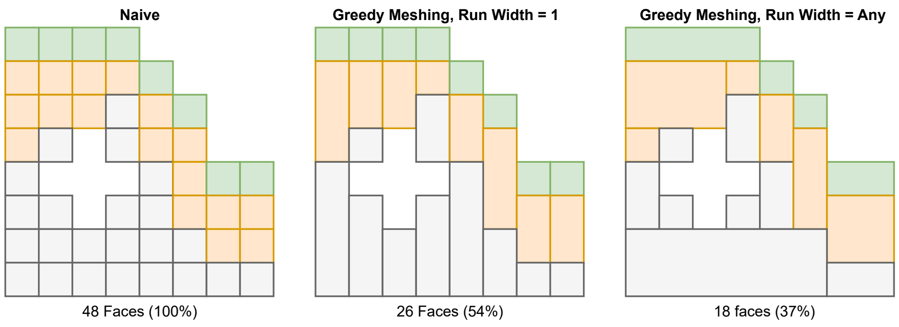

Naive
Greedy Meshing, Run Width = 1
Greedy Meshing, Run Width = Any
48 Faces (100%)
26 Faces (54%)
18 faces (37%)

Figure 4.3: Comparison of meshing techniques. Color indicates voxel type.

1080 by 1080 pixel resolution and has aligned their view to show exactly one face of a single chunk meshed as above. Each triangle aligned with their axis of view (8192 triangles) covers roughly 1/32 of their screen. Each triangle that is on any other axis covers a much smaller amount, approximately 1/512. In this scenario, the pixel shader must be run $(1080\times1080)\times(\frac{1}{32}\times8192)+(1080\times1080)\times(\frac{1}{512}\times40960) = 391910400$ times per frame, yet only $(1080\times1080)\times(\frac{1}{32}\times512) = 18662400$, or about 5%, of these iterations have a visual effect on the final rendered frame. As the number of chunks increases, the total number of pixel shader iterations increases and the ratio of iterations that have a visual impact decreases.

Opencraft 2 adapts a *greedy meshing* algorithm [38]. Greedy meshing has many optimizations compared to the naive meshing approach, trading computational complexity for triangle-minimized chunk meshes. The core of the greedy meshing approach is greedily maximizing the size of *runs*. A run is a single face that spans multiple voxels, with runs generated for all six face axis orientations. Runs are started from the first visible voxel face and are extended along a run axis until either an empty or different type of voxel is encountered. Voxel face visibility is decided by the presence of any voxel in the neighboring position of that face. Since voxel faces may be neighboring voxels from other chunks, the meshing algorithm is no longer self-contained but must read chunk arrays of neighboring chunks. This neighboring voxel lookup is greatly accelerated by the neighboring chunk component (see section 4.2.1.1).

In Figure 4.3, we compare the number of faces resulting from different meshing approaches on a single face of a chunk with size 8. In Opencraft 2's implementation of greedy meshing, we limit run width to a single voxel. This is because the decrease in face count compared to the single width run is typically small, but leads to a significant increase in time spent on performing meshing. Since greedy meshing checks voxel visibility, the worst case scenario of the naive approach, a completely filled chunk, is now reduced to the 1536 visible triangles if no voxels are next to voxels of the same type. A chunk that is entirely full, but of only a single voxel type, is now only 192 triangles! As a final improvement to the greedy meshing technique, Opencraft 2 keeps track of the height of lowest and highest voxels per column in each chunk. These height maps only increase the size in memory a small amount, but can save significant time during meshing, especially when a chunk has large areas of empty space.

The second major optimization that we adapt in Opencraft 2 is a custom voxel engine shader pipeline. Since we intend Opencraft 2 to operate on low-powered devices and to remain performance even when acting as a cloud game host, minimizing the baseline load from rendering is important. The Unity Engine supports extensions to render pipelines in the form of custom shader programs. Opencraft 2 includes a shader program that accepts vertices in a custom, packed, format: 24 bits for vertex x, y, and z position within a chunk, 3 bits for vertex normal, and 5 bits for texture identification number. Since in a voxel engine faces can only point in one of six directions, we only need 3 bits to encode for normal. The texture ID number is an index in a texture array, and since all faces are of the same size, we know texture coordinates in advance. As the general purpose Unity render pipeline with position, normals, and texture information takes five 32-bit values, we reduce the space requirement of each vertex by 85%, and thus reduce the copying time for getting and setting vertex arrays after a meshing operation.

4.2.2. Interfacing with DOTS

PolkaDOTS requires that games utilize Unity's data-oriented technology stack (DOTS) interface. For Opencraft 2, we fulfill this requirement by implementing gameplay behavior and data as entity-component system (ECS) systems and components. In Figure 4.4 we show the systems and components we implement on top of

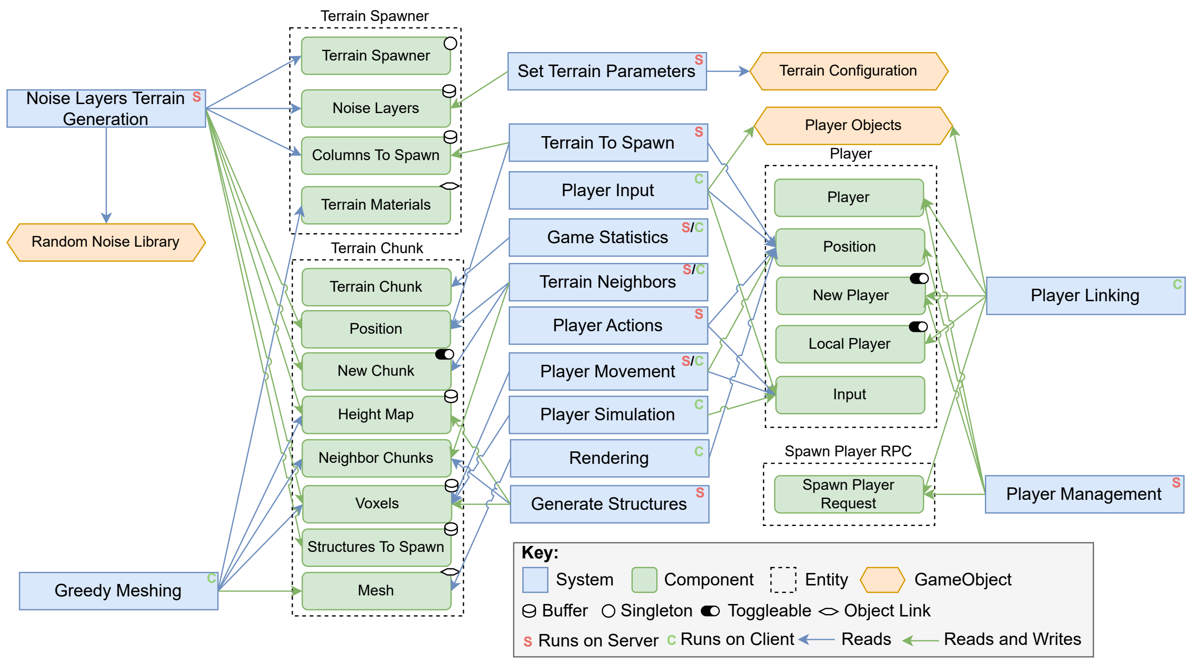


Figure 4.4: Systems and data in Opencraft 2.

the PolkaDOTS framework. Since PolkaDOTS services manage deployment and network connections, Opencraft 2 needs only add gameplay systems and data, and assign these systems and data to entity world types.

Opencraft 2, as an archetypal MLG, has a core feature of a procedurally generated, modifiable environment. Terrain generation is performed in a just-in-time fashion, based on player locations. In our implementation, we split the terrain generation system of Opencraft 2's design into three game systems that operate sequentially to procedurally generate environment terrain. The *Terrain To Spawn* system reads the positions of both existing chunks and player locations, and determines based on proximity what chunk columns must be generated. The *Terrain Generation* system reads the list of chunk columns to spawn. It then creates a new entity for each chunk in each new chunk column, and starts a multi-threaded, asynchronous job to populate these new chunks. The chunk population job runs the noise layers algorithm per-chunk (see section 4.2.1.2), which both fills the chunk's voxel array, and adds any structures created by structure noise layers to the chunk's structures-to-spawn array. The *Generate Structures* system reads the structures-to-spawn array of all chunks. It then determines if any structures are ready to be spawned by checking if they overlap with any chunks that have not been spawned yet. If any structures are ready, it generates them using the multi-pass structure algorithm (see fig. 4.2). The Terrain To Spawn, Terrain Generation, and Generate Structures systems are run only in Game Server entity worlds.

Maintaining bi-directional links between neighboring chunk entities allows fast voxel searching (see Section 4.2.1.1). In Opencraft 2, the *Terrain Neighbors* systems maintains these links by updating them whenever a new chunk is created. The Terrain Neighbors system is run both on Game Server and Client entity worlds. While chunks are generated only on Game Server worlds, a generated chunk may not yet have been synchronized. Thus, a Client world must maintain neighbor links independently of a Game Server, as it receives new chunks. This type of inconsistency is critical to keep in mind when designing a system that operates on synchronized game data.

Only Client worlds run the *Meshing* system. This system performs the voxel engine greedy meshing algorithm variant shown in section 4.2.1.3. The Meshing system reads and writes several types of game data that is not necessary on game Servers, and is thus not synchronized. Notably, the Meshing systems operates primarily on GameObjects. DOTS does not currently have an ECS-native implementation of their Mesh or Material data structure. To associate meshed chunks with their graphical representation, the Meshing system uses *Object Link* components. These are ECS components, but that contain links to GameObjects. Using these object links, the Meshing system generates a mesh per chunk according to the values in that chunks voxel array. Meshing is performed for a chunk each time the voxel array changes.

PolkaDOTS requires game behaviors to be implemented with the DOTS, but allows other behaviors to be present with the caveat that they will operate on all PolkaDOTS applications. In Opencraft 2, we use several GameObject and pure C# behaviors that are not in the DOTS. Static libraries and configuration, such as the pseudo-random noise generation library and terrain configuration, do not run any behavior automatically, as they are only accessed by game systems within entity worlds. GameObjects that are created by game systems and handled through Object Links, such as the Mesh and Material objects used during meshing, also do not run behaviors on their own. The most crucial GameObject behaviors outside of the DOTS is the handling of *Player Objects*, which we explain in detail in the following Section.

4.2.3. The Player objects and handling linkage

PolkaDOTS removes the assumption that there is only a single local player on Client entity worlds. It also uses two Unity packages outside of the DOTS: Render Streaming and Input System. These packages are built around GameObjects and are incompatible with ECS entities, and both packages interact closely with low-level features of the Unity engine, specifically managing input received from hardware and generating and displaying rendered frames.

Therefore, to be compatible with PolkaDOTS, Opencraft 2 defines player entities and links them to associated Player Objects using two symmetrical systems. The *Player Management System* runs only on the Server and handles creating new Player entities when it receives a *Spawn Player Request* remote procedure call (RPC). The *Player Linking System* checks for Player Objects instantiated by PolkaDOTS, sending Spawn Player Request RPCs when it finds new player objects. It then checks any new player entities and links them to an associated player object by checking that the unique identifiers match. Since PolkaDOTS handles separating local and remote (guest) players, Opencraft 2 handles both identically without needing to create a distinction between them. Notably, Player entities for players that are not local or guests do not need an associated GameObject, as they do not require either collecting input or rendering frames.

Once Player Object and entity have been linked, the *Player Input* system handles reading the input collected by Player Objects to an Input component on the associated Player entity. It also handles synchronizing the position of the Player Object to the position of the Player entity in order to update the player's camera view.

Two systems are responsible for computing the game state results of Player Input, *Player Movement* and *Player Actions*. The Player Movement system runs on both Game Server and Client worlds, using NFE prediction mechanisms. This allows the Client world to move local players without waiting for the Server to perform the same action, decreasing the response time of movement, but potentially leading to mispredictions. In Opencraft 2, as in other MLGs, inconsistency between Server and Client terrain is a common source of mispredictions, but the cost of misprediction is low. The Player Actions system is responsible for modifying the environment, and is run only in the Game Server. While this system could client-side predicted, the cost of a misprediction is high. Modifying a chunk requires the Meshing system to update the associated chunk Mesh. Upon a misprediction, the expensive meshing process would be repeated several times in quick succession.

4.2.4. Additional game metrics

The last requirement that the PolkaDOTS framework imposes is that games should add any relevant game-specific performance metrics as an extension to the Unity Profiler. In Opencraft 2 we add two game-specific metrics *Chunk Count* and *Player Count*, both to act as an example of how to extend the Unity Profiler and as a useful performance metric for debugging and analyzing experiments. The number of total chunk entities at a given time is a useful metric when comparing the generation rate of new environment chunks on Game Servers to the rate of receiving chunks on Game Clients. We show the code required to add this metric to the Unity Profiler in Listing 1.

To set the values of the NumTerrainChunks counter, Opencraft 2 has a *Game Statistics* system which runs on both Game Server and Clients. This system queries the total number of chunk entities each frame and sets the counter's value. The Unity Profiler itself is responsible for reading the value of the NumTerrainChunks counter and depending on configuration, either displaying it within the Unity Profiler Editor Window, or writing it to a performance metric log file during runtime.

4.3. Opencraft 2 as a research platform

An important requirement is that Opencraft 2 act as a platform for future research into differentiated deployment, MLGs, and online games in general. To meet the requirements to be act as a research platform, we

```
public class GameStatistics{
    public static readonly ProfilerCategory GameStatisticsCategory = ProfilerCategory.Scripts;
    public const string NumTerrainChunksName = "Number of Terrain Chunks";
    public static readonly ProfilerCounterValue<int> NumTerrainChunks =
        new ProfilerCounterValue<int>(GameStatisticsCategory, NumTerrainChunksName,
            ProfilerMarkerDataUnit.Count);
}
#if UNITY_EDITOR
[System.Serializable]
[ProfilerModuleMetadata("Game Statistics")]
public class GameProfilerModule : ProfilerModule{
    static readonly ProfilerCounterDescriptor[] k_Counters = new ProfilerCounterDescriptor[]{
        new ProfilerCounterDescriptor(GameStatistics.NumTerrainChunksName,
            GameStatistics.GameStatisticsCategory),
    };
    static readonly string[] k_AutoEnabledCategoryNames = new string[]{
        ProfilerCategory.Scripts.Name
    };
    public GameProfilerModule() :
        base(k_Counters, autoEnabledCategoryNames: k_AutoEnabledCategoryNames) { }
}
#endif
```

Listing 1: Adding the chunk count performance metric as a Unity Profiler extension.

make several design and implementation decisions, which we outline in this section.

4.3.1. Configurable workload

We implement Opencraft 2 as an archetypal MLG. In on our previous work we found that in a MLG, the game workload is comprised of two factors: players and environment [20]. Players effect workload both through requiring processing of their in-game avatar, and by increasing the total amount of game state that must be synchronized across the network. Environment workload has two primary categories: initial creation of the environment (called *terrain*), and dynamic behaviors of the environment after its creation. Additionally, all online games experience workload from network operation overhead. In the following subsections we describe how these workload categories can be configured in Opencraft 2.

Player Workload

The player contribution to MLG workload is both the amount of players and the behavior that they exhibit. The amount of players is trivial to configure, it can be increased by deploying more instances of Opencraft 2 clients. In our experiments, we use two player amount techniques: a *fixed* number of players, or an *increasing* player amount setup where more players are added over time.

For player behavior, we utilize two approaches. PolkaDOTS includes integration with the Unity Input system package to emulate players using recorded player input. Configuring player behavior is then performed by selecting an input recording trace. For convenience, we include in the container of Opencraft 2 we use in these experiments several prerecorded player input traces. These input traces include key player behaviors observed in MLGs: *Exploration*, *Collection* and *Building* [24]. Because these input recordings rely on the terrain being identical to when the input was recorded, we use a fixed environment generation configuration as well as a set seed value: '42', for all input recordings. This is a drawback of the input playback approach, future researchers that modify the environment generation process of Opencraft 2 will need to record new input traces.

The second approach to player emulation is through behavior simulation. As opposed to input playback, behavior simulation requires access to the current game state. In Opencraft 2, the player simulation system runs on the client, transparently to the server. It picks pseudo-random positions for the player to move to based on a simple model of player behavior. Finally, it determines the path to arrive at the target and generates the correct inputs to move along that path. In our experiments, we utilize a *BoundedRandom* be-

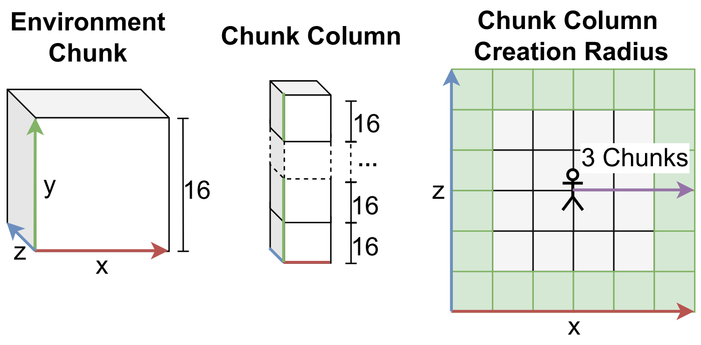


Figure 4.5: Opencraft 2 environment configuration.

Table 4.2: Opencraft 2 *Complex* and *Flat* environment generation layer configurations. The complex configuration results in a column height of 3 and the flat configuration a column height of 1.

Config Name	Layer Type	Terrain Type	Frequency	Minimum Height	Maximum Height
Complex	Absolute	Stone	110	0	26
	Absolute	Stone	32	10	26
	Additive	Stone	7	1	4
	Absolute	Dirt	50	20	25
	Additive	Dirt	20	2	5
	Surface	Grass	1	1	1
	Structure	Tree	0.01	3	5
Flat	Surface	Grass	1	1	1

havior model. In this model, players stay within a small area, by default a 32 by 32 square, and continuously move short distances before briefly pausing and changing direction. This model approximates player behavior when staying within an in-game structure such as a home base, or when interacting with other players during socialization or combat.

Environment Workload

The modifiable virtual environment (MVE) is primary form of content of an MLG, and its generation, modification, and delivery all contribute to game workload. Opencraft 2 has many parameters relating to its MVE that can be freely modified. We set default values of these parameters, and visualize this configuration in Figure 4.5. Chunks are set to a cuboid of size 16, with new environment chunks being generated in vertical columns of a variable height. Environment columns are generated when any player moves within a 3 chunk radius of the new columns position. With this configuration, when a player moves 1 chunk (16 voxels) in either the x or z axis towards environment that has not been created, 6 new columns are generated.

During the column generation process, Opencraft 2 populates each chunk in a column using the two-pass layered noise algorithm described in section 4.2.1.2. Configuring MVE generation in Opencraft 2 is done through specifying different sets of noise layers, which each encode parameters for a given layer of the terrain. In Table 4.2 we include two sets of noise layers. The *Complex* set of layers result in chunk columns with a height of 3 and resembling a hilly landscape with sparse trees. The *Flat* noise layer set results in chunk columns with a height of 1 that contain only a single flat plane of grass blocks.

Network Workload

Online-game netcode is complex, there are many configurable parameters that have a significant impact on game performance. In Opencraft 2, netcode is separated into game and stream networking. The game networking is operated through NFE. The most important NFE parameter is *netcode tick rate*. This rate governs how often snapshots are synchronized between servers and clients. Importantly, this tick rate does not need to be identical to the game's desired frame rate, or even the game's simulation rate, e.g. ticks per second (TPS). In our experiments, Opencraft 2 uses a TPS or 20 or 60, and sets the netcode tick rate to the same value.

NFE includes an interest-management mechanism that allows individual entities to be given a per-connection importance score. Entities can also be given varying importance depending on what components they have, and how often they are expected to update. By default, Opencraft 2's interest management mechanism is configured to prioritize entities based on distance, and to prioritize players over environment chunks. The importance mechanism only activates when snapshot size exceeds a maximum value. This max value is set to the max-transfer unit (MTU) of the Unity Transport Pipeline, or ~1400 bytes.

When performing cloud gaming, Opencraft 2 does not use NFE between a cloud host and client. Instead, video, audio, and input streams are established using WebRTC [8]. The specific stream encoder library as well as the encoder parameters are configurable. Opencraft 2 defaults to the hardware-accelerated NVIDIA NVENC encoder and decoder [9], with a video stream at 1080p 60FPS quality, an accepted standard setting of cloud gaming [10]. Certain encoder libraries, including NVENC, use a variable bitrate system, which automatically regulates bitrate to balance visual quality and bandwidth usage. How much the bitrate is allowed to vary can be configured, but it is not limited in Opencraft 2 by default.

4.3.2. Extensible game features

Since future research will explore ideas and features that Opencraft 2 does not currently enable, Opencraft 2's systems and mechanisms are extendable. PolkaDOTS enforces modularity of games built upon it, to an extent. For game services to be supported by the differentiated deployment mechanism, they must explicitly declare their game state access dependencies. Opencraft 2 features are implemented as ECS systems, with clearly defined component access dependencies. This structure decouples independent features, allowing them to be modified in isolation. The voxel engine is a key feature of MLGs and thus likely to be a subject of future research, Opencraft 2 includes special interfaces to extend voxel engine capabilities. New block types, noise layers, and generation policies can be easily added by implementing the associated interfaces.

Opencraft 2 uses PolkaDOTS's metric collection system. This system allows the additions of new metrics as extensions to the Unity Profiler. Thus, it is simple to extend Opencraft 2 with new performance metrics. As the metric collection systems both of PolkaDOTS and Opencraft 2 are ECS-based, it is simple to replace or modify these systems to extend the metric collection tooling.

Where PolkaDOTS uses NFE for remote performance monitoring and configuring deployment, Opencraft 2 uses NFE for gameplay snapshot synchronization. Future researchers may want to extend NFE functionality for Opencraft 2, and this is enabled both through the entity world abstraction as well as the ability to selectively deploy NFE systems using PolkaDOTS's world bootstrap system. NFE operates within entity worlds and operates as typical ECS systems, and PolkaDOTS allows fine-grained control of ECS system placement within entity worlds.

Opencraft 2 uses the Unity Render Streaming package for streamed gaming deployments. As the ability to dynamically switch deployment to a streamed gaming configuration is a novel feature of PolkaDOTS, it is likely to be the subject of future work. Opencraft 2 uses a custom version of the Unity Render Streaming package, with the source code bundled. Researchers can freely modify this bundled version of the package to add or modify streamed gaming functionality.

4.3.3. Reproducibility

For experiments utilizing Opencraft 2 to be useful in research, they must be reproducible. In other words, it must be possible for a given workload to be recreated. Our approach to reproducibility in Opencraft 2 has two factors: pseudo-random workloads and containerization.

Opencraft 2 workloads uses pseudo-random generation when random values are needed. For instance, when generating the MVE terrain, every noise layer's noise map is created in a deterministic order using a user-provided seed. With the same seed and noise layers, the same terrain is generated with no variance. Similarly, player emulation using input recordings allows Opencraft 2 to receive the exact same inputs from clients at the same time between experiments, and player emulation using simulation uses pseudo-random targets for movement based off a seeded value.

Reproducibility of experiments is not only about receiving an identical workload, it is also important to ensure the operating environment is replicable. To ensure Opencraft 2 is always run on an identical software stack, we include in Opencraft 2's build pipeline a containerization step which packages Opencraft 2 into a Docker container. Containerization eliminates the possibility of experiments differing between operating systems and when using different libraries.

However, as in all distributed systems, reproducibility is limited by external factors outside of direct control. For instance, network conditions between Opencraft 2 instances separated by public internet infrastructure may vary between experiments.

4.3.4. Documentation

It is not feasible to configure or extend a system without a baseline understanding of it. Therefore, we provide substantial documentation for Opencraft 2. Following recommendations for good software engineering practices, we provide detailed in-code documentation in the form of structured comments. Additionally, we provide more broad descriptions of Opencraft 2's implementation structure in markdown files included in the source code repository. When using a structured file, such as Opencraft 2's terrain noise layers, we include reference files as templates, which include associated explanations of each fields possible values. Finally, an important source of documentation for Opencraft 2 is Chapter 4 of this report.

5 Evaluation of differentiated deployment

In this chapter, we address the third research question (RQ3) by designing, conducting, and analyzing a series of real-world experiments on Opencraft 2 using our experiment tool, Dither. These experiments, listed in Table 5.1 result in 4 **K**ey **F**indings:

KF1 Dynamic migration of game services allows online games to adapt to resource constraints (Section 5.3).

The ability to move game services away from resource constrained nodes to nodes with available resources allows online games built on Polka to flexibly adapt to resources constraints, with minimal interruption to the player due to the migration. In our experiments, game services migration can decrease average frame time variability (measured as 95th to 5th percentile range) by 32.71% and memory use by 34.29%, but at the cost of an increase in average bandwidth usage from 1.05mbps to 3.21mbps.

KF2 Dynamic migration of the rendering component of can decrease round trip time (Section 5.4).

The round trip time (RTT) between a game server and client is a key indicator of player experience. Our experiments show that when a client node is CPU constrained, it can experience an increase in average RTT of 12ms compared to a client that is not CPU constrained. In this scenario, migration of the rendering component can result in a reduction of total RTT by 8.87ms and a reduction in variation of RTT.

KF3 Service initialization causes significant instability (Section 5.5).

Due to game asset loading times, CPU cache saturation, and memory management, online games can exhibit delayed reductions in performance after either normal game starting or dynamic service migration, by up to a *2x reduction in performance* in our experiments, and after a delay of up to 70 seconds.

KF4 Opencraft 2 supports up to 127% more players than commercial MLGs (Section 5.6).

Under favorable conditions, Opencraft 2 supports up to 250 players, an increase of 127% compared to Minecraft. However, supported player count is strongly reduced by increased workload, down to 25 players in the worst case.

Table 5.1: Experiment overview. Our experiments focus on Differentiated Deployment (DD) or Scalability (SC). Player behavior is either input playback (P) or simulated (S). See Section 4.3.1 for an overview of workload parameters. We detail the hardware used in Section 5.2.

Focus		Workload				Hardware	
		Players	Behaviour	Terrain	TPS	Server vCPU	Client vCPU
Section 5.5	DD: Render Baseline	1	P:Explore	default	60	4	4
Section 5.3, 5.4	DD: Render Constrained	1	P:Explore	default	60	4	4, 1.5
Section 5.6	SC: Scalability Baseline	25-300	S:Bounded	flat	20, 60	16	4
Section 5.6	SC: Scalability Terrain	25-150	S:Bounded	default	20, 60	16	8

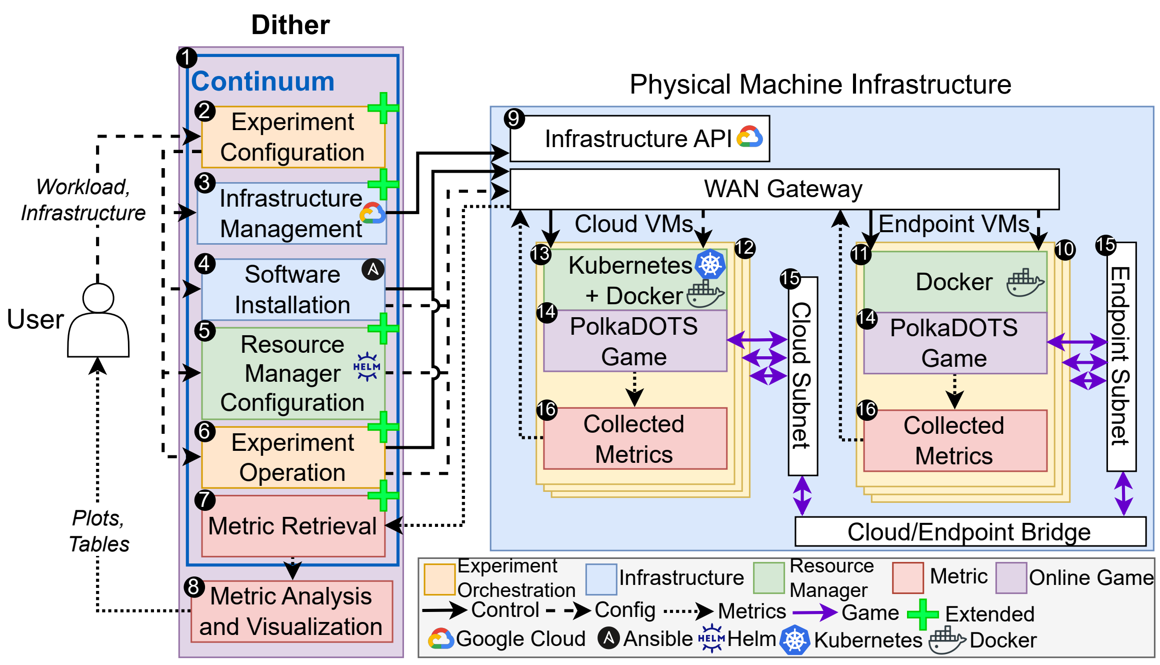


Figure 5.1: Dither experiment tool design.

5.1. Dither: a Reproducible Real-World Experiment Runner for PolkaDOTS

Running reproducible real-world experiments in modern networked distributed systems is a non-trivial problem. Configuration can be painstaking and mistakes can be difficult to diagnose. To enable reproducible experimentation with online games built on PolkaDOTS, we design and implement Dither.

5.1.1. Design of Dither

Figure 5.1 shows the design of Dither, a tool for running experiments using online games built on the PolkaDOTS framework. Dither automatically deploys, configures, and runs infrastructure, resources managers, and the online-game application. The core of Dither is based on *Continuum* [26], a framework for conducting real-world experiments for edge-computing. Continuum automates infrastructure deployment and benchmarking: it can deploy a variety of infrastructures and networks both locally and in the cloud, handle software installation for operating services and resource managers, and deploy and benchmark applications with diverse configuration options [26]. To enable experiments and benchmarking with PolkaDOTS, we make modifications to various Continuum components. We provide a detailed overview of how we extend Continuum in Section 5.1.2.

Dither allows users to run reproducible and representative experiments by controlling the the experiment environment, workload, and metric collection. To create a controlled experimental environment, Dither's infrastructure management component (label ❸ in Figure 5.1) deploys virtual machine (VM) infrastructure through Google's Cloud Computing API (❾). Dither then instantiates a set of VMs on the managed infrastructure. Dither distributes the online-game workload between sets of two types of VMs: “Endpoint" and “Cloud" (❿ and ⓬). These categories are representative of real-world online-game deployments, where endpoint devices run a game frontend client and cloud devices run the game's server. Accordingly, endpoint and cloud VMs differ in resource managers. Endpoint VMs use only self-contained resource management through Docker (⓫), and Cloud VMs resource management (⓭) uses Kubernetes to manage resources of multiple cloud VMs. To allow experiments based on network conditions, Dither configures each type of VM to communicate through a subnet (⓯) which can be configured to limit network performance to chosen parameters.

Users interact with Dither by specifying parameters through a simple yet expressive configuration interface (❷). This interface lets users specify infrastructure details and associated access credentials, as well as the relevant experiment workload configuration. We provide an example of Dither's configuration interface in Appendix A.

Using infrastructure details and access credentials provided by the user, Dither's infrastructure management component (❸) manages VM infrastructure through Google's Cloud Computing API (❾).

After ensuring VMs are fully configured and deployed, the software installation component (❹) installs all prerequisite software, as well the PolkaDOTS online-game application itself (⓮). As a final preparatory step, the resource manager configuration component (❺) uses Helm to specify the behavior of Kubernetes and Docker on all VMs. After all preparation is complete, the experiment operation component (❻) instructs the resource manager to run the online game, and continuously checks experiment progress to await completion or report errors.

During the experiment, the PolkaDOTS collects performance metrics (⓰), and after the experiment is complete these metrics are gathered by the metric retrieval component (❼) and processed by the metric analysis and visualization component (❽) for the user to view.

5.1.2. Extending Continuum to Support Online Games

We use the *Continuum* framework as the core of Dither, and extend it to support online games built on PolkaDOTS. The components of Continuum that we extend are marked with a green plus in Figure 5.1.

PolkaDOTS performs application-level deployment and resource management, requiring individual PolkaDOTS instances to be given unique configuration. We add PolkaDOTS-specific configuration support to Continuum's application configuration component (❷).Therefore, we add a mechanism to the experiment configuration component to allow users to specify PolkaDOTS deployment configuration at a per-instance granularity, rather than one set of configuration parameters for each category of VM. Continuum's resource manager configuration component (❺) uses Helm to control the behavior of Kubernetes on Cloud VMs and controls Docker directly on Endpoint VMs. We extend Continuum's resource manager configuration to support PolkaDOTS configuration by adding automatic conversion of PolkaDOTS configuration to Kubernetes ConfigMaps, and creating metric collection shared data volumes.

Dither adds GPU support to endpoint VMs. The use of GPUs allows Dither experiments to better represent real-world online-game deployments. Many online games require players to utilize hardware with graphics acceleration. Cloud gaming as a service is not feasible without GPU acceleration. We extend Continuum's Docker support to enable GPU-accelerated 3D graphics workloads both on devices with physical screens and on headless devices with a virtual display.

We extend Continuum's VM network emulation by adding an Endpoint VM subnet. This allows experiments involving deployments with Endpoint VMs directly communicating, rather than routing workload traffic exclusively through Cloud VMs.

Continuum supports the use of an existing distributed performance monitoring system: Prometheus. Prometheus integrates well with Kubernetes, and can publish performance to a live dashboard during experiment operation through Grafana. Dither retains support for Prometheus and Grafana, but we find these tools sample data at an insufficient rate for detailed analysis of online games. Therefore, we modify Continuum's metric retrieval component to aggregate PolkaDOTS's fine-grained metric logs. To read these aggregated data logs, we implement a custom metric analysis an visualization tool, explained in the following subsection.

5.1.3. Analysis of PolkaDOTS Metrics

Dither's handling of performance metrics constitutes a significant advantage over running experiments manually. Without Dither, it is still possible to run experiments with PolkaDOTS and view performance traces in the Unity Profiler window. While the Unity Profiler window is extremely useful for detailed and low-level analysis of the performance of a single instance of a game, it is less practical for analysis of an entire experiment: it can only analyze and display performance from a single node at a time and in a range of up to 2000 frames.

To enable users to perform a holistic analysis of experiments, Dither includes a metric analysis and visualization component (❽). This component operates after an experiment has run, and takes as input the aggregated PolkaDOTS performance metric logs from all VMs. Using these logs, it then performs three actions: *alignment*, *processing*, and *visualization*.

Different categories of VMs, varying workloads, and heterogeneous hardware all influence the initialization time and processing speed of games built on PolkaDOTS. In other words, each instance of the game starts and stops sampling at slightly different times, and with varying sampling periods. Consequently, the performance metric traces that Dither receives are not aligned. To synchronize performances traces for comparison, Dither first calculates the elapsed time as a rolling sum of frame times, measured in nanoseconds. Achieving alignment is then a process of determining a correct time offset to apply to each trace. To deter-

Table 5.2: Experiment deployment hardware parameters.

Parameter	**Values**
Cloud VM Type	**E 4vCPU**, E 16vCPU
Endpoint VM Type	E 2vCPU, E 4vCPU, **N 4vCPU**
Endpoint GPU	**Yes**, No
Cloud VM Amount	1, **2**
Endpoint VM Amount	**2**, 12
Endpoint to Cloud Latency	**0 ± 0ms**, 20 ± 5ms

mine the correct offset, we compare values of the NFE snapshot counter, allowing traces to be aligned by a logical clock rather than a physical one. In cases where this snapshot counter value is not being collected, such as during cloud gaming on thin clients, traces can be aligned through event indicators, such as game streaming being marked as started.

Once traces are aligned, the metric analysis and visualization component then processes the traces. An important processing step is applying statistical operators. PolkaDOTS gathers noisy data at high frequencies, making useful analysis of performance trends difficult. Therefore, Dither applies a rolling window operator over 1 second to calculate the median. As aggregate statistical measures can hide performance problems, we also report variation over the same window period through calculating the 95th and 5 percentile.

Finally, Dither creates visualizations of the aggregated, aligned, and processed performance data. We include a set of general-purpose visualization scripts for common performance metrics such as frame times, memory, and round-trip-times. The visualization scripts are written in Python, and use widely used and well-documented visualization libraries. For experiments evaluating game-specific metrics, additional scripts can be easily added using the general-purpose scripts as templates.

5.2. Experimental Setup

We deploy experiments using Dither on Google Cloud Platform (GCP) VMs. In Table 5.2 we show the relevant deployment hardware parameters and their values. Default values are in bold. We use two different GCP VM families: *e2-standard* (E) and *n1-standard* (N). The E machine family are intended for general compute workloads, and are a popular choice for many workloads due to a favorable cost-to-performance ratio compared to more specialized machine families. N VMs are also intended for general compute workloads, but unlike the E family, they support the addition of several dedicated hardware resources, including SSD storage and the use of accelerators such as GPUs. In our experiments, we do not utilize SSDs, but do attach to the N VMs a single Nvidia Tesla T4 GPU. In all experiments, nodes indicated as a “Server" do not have an attached GPU and do not run a graphical frontend. Such nodes are run as Cloud VMs by Dither on E VMs running a standard Ubuntu 20.04 headless OS image.“Client" nodes are deployed as a Docker container. If a frontend is required, these nodes are deployed via Dither onto N VMs with an attached T4 GPU. These N VMs run a custom Ubuntu 20.04 image that we preconfigure for supporting hardware-accelerated graphics applications in Docker containers. If a frontend is not required, Client nodes are deployed on E VMs running the same Ubuntu image as the Server node.

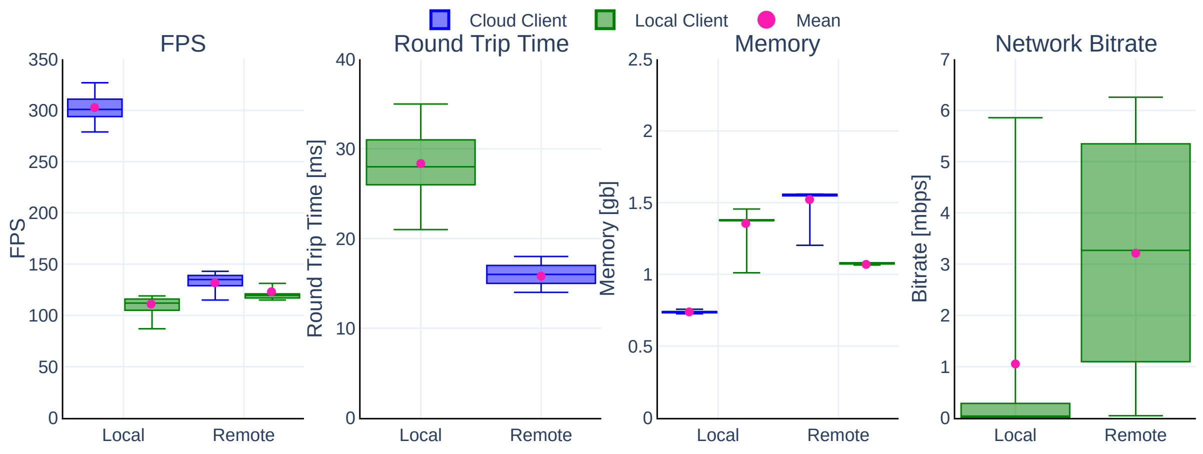


Figure 5.2: Distribution of frames per second (FPS), RTT, Memory, and Network Bitrate of Opencraft 2 when switching from local rendering to cloud gaming with CPU restrained on the local client. Whiskers extend from 5th to 95th percentile. Pink dot indicates arithmetic mean.

5.3. KF1: Dynamic migration of game services allows online games to adapt to resource constraints

As described in Chapter 3, online games can experience sudden changes in hardware resource availability. In this experiment we show that differentiated deployment via dynamic migration of game services allows the online game to adapt quickly to resource constraints. We posit that this adaptation may improve player experience and reduce overall risk.

In Figure 5.2 we show the distribution of FPS, RTT, memory use, and network bitrate of the two Opencraft 2 clients in the Render Component CPU-Constrained experiment. In this experiment, the local client is run with a limited amount of CPU cores, set at 1.5. The cloud client is run with 4 cores.

With the 1.5 CPU limit the local client (green) is experiencing significant performance variability before the switch to remote rendering, with the 5th percentile of FPS at 87 and the 95th percentile at 119, a range of 32 FPS. Once the migration to remote rendering is complete there is an immediate performance benefit, FPS is slightly increased overall on the local client, with a average (pink dot) increase of 5 FPS and performance variability is reduced, with the post-switch 5th and 95th percentiles at 115 and 131 FPS respectively for a range of 16 FPS.

The cloud client (blue) is idle before the switch to remote rendering. After the switch to remote rendering, the cloud client connects to the game server and starts streaming frames to the local client. Even while streaming frames, the cloud client exhibits less performance variation than the local client before the switch, with a range of only 28 FPS.

FPS is not the only measure of performance we use to analyze the effect on the Local Client. In Figure 5.2 we show the distributions of game round-trip time (RTT), memory use, and network bitrate for both clients. After migration to remote rendering, median memory usage of the local client drops from 1.38GB to 1.08GB, a 21.74% decrease. Surprisingly, the median game RTT of the Local Client before the migration is 28.36ms, 44.29% higher than the RTT of the Cloud Client after the migration of 15.8ms. We analyze the implications of this effect in detail in section 5.4.

While running rendering locally, the local client receives an average of 1.05mbps in the form of delta-encoded game state snapshots from the Server. After migrating to remote rendering, the local client receives on average 3.21mbps from the cloud client as a stream of video frames, a 305% increase in network bandwidth. This value is in line with previous work that demonstrates the high network bandwidth requirements of streamed gaming. However, we find that bandwidth use Opencraft 2 without render streaming can be comparable, at least in short bursts. The 95th percentile of bitrate of the local client before switching to render streaming is 5.86mbps, with the high bitrate usage occurring when receiving new terrain chunks. This phenomenon suggests that the suitable network requirements for local and remote rendering are similar, rather than being much higher for render streaming deployments. Indeed, remote streaming configuration allows setting a maximum bitrate, allowing players to trade stream quality for bandwidth usage. Therefore, for network connections with bandwidth in the range of 2 to 5mbps players may receive better quality of service through a bitrate limited render streaming connection than a local rendering deployment.

The demonstrated ability to dynamically switch to a cloud gaming deployment with minimal interruption to player experience and a resulting increase in performance and decrease in resource utilization is extremely promising. There are many scenarios in which such a switch could be beneficial. For instance, mobile devices could switch to a cloud gaming deployment to conserve battery life, or a virtual-reality (VR) headset device could switch between local and remote rendering depending on the complexity of the game environment.

However, PolkaDOTS enables this switch by moving both resource management and render streaming into the application layer. This has the implication of requiring game providers to pay for GPU-accelerated frontend clients alongside servers, which may raise the cost of the game as a whole and prevent adoption. To mitigate this effect, players could use their own devices as hosts for streamed gaming over LAN rather than cloud gaming.

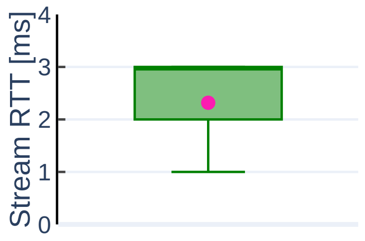


Figure 5.3: Distribution of stream RTT of Opencraft 2 during Render Component CPU-Constrained experiment. Whiskers extend from 5th to 95th percentile. Pink dot indicates arithmetic mean.

5.4. KF2: Dynamic migration of the rendering component can effectively reduce round trip time

Maintaining low latency for game actions is a core principle of online-game performance. High latency of certain game actions, notably actions that have direct visual impact, such as player movement, or actions tied to latency sensitive information, such as enemy player position, can strongly reduce player experience. In fact, causing increased latency is frequently cited as a downside of cloud and streamed gaming [39].

However, we find that when CPU resources are constrained, game state RTT is increased. An example of this can be seen in Figure 5.2. In this scenario, switching to *cloud gaming results in lower total RTT*. Before the switch, the local client has an average game state RTT of 28.42ms. After the switch, the cloud client has an average game state RTT of 16.55ms, a reduction of 41.77%. However, to find the total RTT that the local client is experiencing, we must compute the total RTT including the *stream RTT*, which is the RTT of rendered frames between the cloud client and local client.

In Figure 5.3 we show the distribution of stream RTT of the remote rendering stream. Stream RTT has a maximum of 3.0ms. The total RTT experienced by the local client is the sum of stream RTT from the local to cloud client, and the game RTT between the cloud client and server. Therefore, the total RTT of the local client after switching to cloud gaming is 3.0ms + 16.55ms = 19.55ms, a 31.21% decrease in total RTT.

Increasing latency is a well-known downside of cloud gaming, yet we find that in certain scenarios cloud gaming can decrease experienced RTT. To support this surprising result, we analyze in detail when cloud gaming can be expected to decrease latency. We observe that decreased CPU in game clients leads to increased game RTT. Since RTT is commonly reported as a measure of network quality alone, we stress that game RTT measures the amount of time it takes for a game snapshot to be applied to game state. This metric includes the potentially substantial time it takes for a game client to determine and perform any necessary game state rollback or interpolation. Further, a decreased ability to handle game state rollback or interpolation can create a positive feedback loop which can directly overload or crash a game if not prevented. This occurs when there is insufficient CPU resources to operate the game at or above the *netcode rate*. When the client is performing netcode updates at a slower rate than it is receiving them, rollback must account for a longer duration in-between updates, which leads to more chance of mispredictions and more game state needing to be rolled back. This increased netcode workload uses more CPU resources, which can decrease the client's netcode rate even further. Thus, decreased CPU resources directly effect game RTT. As a consequence, it is inaccurate to use RTT solely as a measure of network quality in Service Operation Policies.

We stress that *decreased total RTT does not directly lead to reduced latency for all actions*, and even when total RTT is decreased by switching to cloud gaming, the latency for certain actions may increase. This is because game RTT is not a direct measure of action latency, but an upper-bound. Online games utilize several latency-hiding techniques, most notably client-side prediction. For predicted game state, increased game RTT does not increase latency. It does, however, increase both the frequency and severity of client mispredictions. For any game state that is not predicted, action latency is exactly the RTT value.

Therefore, when a local client has insufficient CPU resources to maintain a low game RTT, and there exists a cloud client with sufficient CPU resources and a suitably low ping to both the local client and server, switching to cloud gaming will reduce the latency of non-predicted actions, and may reduce the frequency and severity of mispredictions for predicted actions.

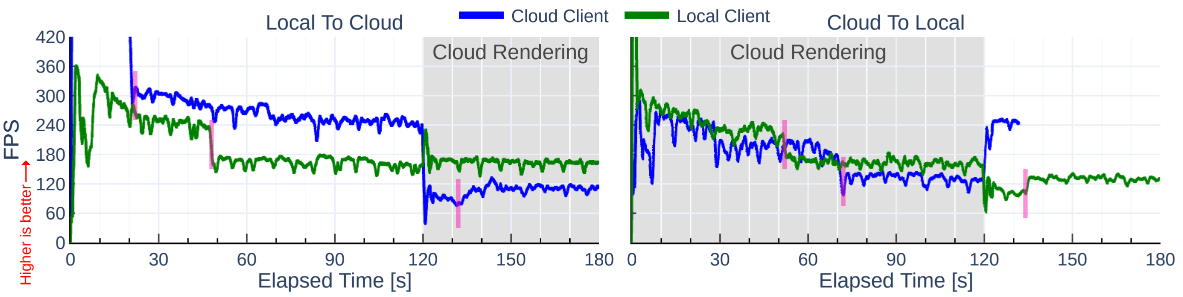


Figure 5.4: FPS of Opencraft 2 when switching between remote rendering and local rendering. Vertical pink bars indicate stabilization times.

5.5. KF3: Service initialization causes significant performance variability

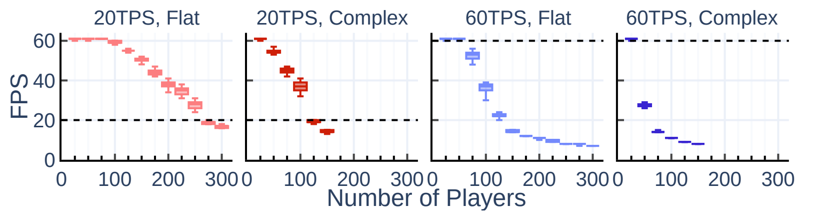
As discussed in Section 2.3.2, online games require low performance variability to operate correctly. However, our experiments indicate that, without effective latency hiding techniques, service initialization steps in a differentiated deployment system such as PolkaDOTS can cause significant performance variability. We find that online games experience sudden increases in performance variability after a delay. The duration of this delay we refer to as *stabilization time*. We find that causes of these performance variability increases are numerous and can result in different stabilization times between identical game executables that are configured differently.

In Figure 5.4 we show the FPS of Opencraft 2 during the Render Component Baseline experiment. In this case, all nodes have sufficient CPU and network resources. The period where the local client is streaming the game from the cloud client is indicated with a gray background.

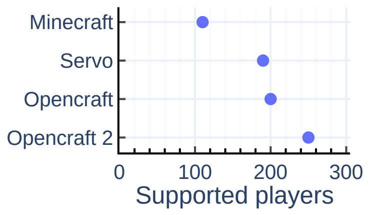
Each node has a different stabilization time. When the local or cloud client begin running the rendering component, they experience an initial spike of increased frame time from loading all necessary game assets, connecting to the Server, and consequently receiving and performing meshing of the environment game state. This initial spike lasts ~12 seconds. A second spike occurs when all reserved memory capacity is used. The garbage collector system then begins aggressively reclaiming memory, resulting in a significant increase in frame times. On the local client, this occurs at t=48 (Local To Cloud, green), whereas on the cloud client, this occurs at t=72 (Cloud To Local, blue). The performance spike caused by memory saturation also occurs when running the local client in a cloud gaming mode, at t=52 (Cloud To Local, green), and when running the cloud client in a stand-by mode at t=20 (Local To Cloud, blue).

Stabilization time changes after a service migration. After switching to cloud gaming on the local client, performance is immediately stable, as memory capacity has already been saturated before the migration occurred. Similarly, when switching to local rendering on the local client and when switching to cloud gaming on the cloud client, the stabilization time occurs immediately after connecting to the server and receiving environment game state, at t=132 for the cloud client (Local To Cloud, blue) and t=134 for the local client (Cloud To Local, green).

The phenomenon of stabilization time has several consequences to the design of service operation policies. First, policies that are based on sudden decreases in performance must make a distinction between performance decreases caused by the expected sources involved in stabilization, or if the performance decrease is unexpected. A service operation policy that is not aware of these stabilization times could decide, for instance, that a service is causing too much performance degradation, even if that service is only experiencing a temporary performance drop due to synchronizing game data. Second, designing service operation policies with triggers only based on frame time is likely to result in insufficient granularity. If a service operation policy does not take into account performance metrics related to the causes of stabilization it will not be able to determine if a certain performance drop is expected or not. Finally, policy actions must be designed with the knowledge that performance of a service after a migration is not identical to performance of the same service before a migration. Due to factors such as memory management or CPU caching, migration actions modify the state of the system and can affect stabilization times.



(a) FPS distribution after stabilization for increasing number of players. Whiskers extend from 5th to 95th percentiles. Higher is better. Black dashed line indicates the TPS threshold.



(b) Supported players by MLG. Values from Donkervliet et al 2023 [13].

Figure 5.5: Scalability of Opencraft 2.

5.6. KF4: Opencraft 2 supports up to 127% more players than commercial MLGs

High scalability in terms of player count is an important metric both for game developers seeking to create MMOGs and for game providers hosting them. Supporting more players per instance entails reducing the total number of instances required to achieve a target player count. We find that Opencraft 2 achieves higher scalability under favorable conditions than state-of-the-art commercial and research MLGs.

In Figure 5.5a we show the FPS distribution after stabilization of the Server node when increasing numbers of players are connected. We repeat the experiment with different workload configurations. Plots in red indicate we run the game at a TPS of 20, and plots in blue at 60 TPS. A darker shade indicates that we use the default, complex, terrain configuration (detailed in Section 4.3.1), and a lighter shade a minimal, “flat" terrain configuration. The horizontal black dashed lines indicate thresholds of 20 and 60 TPS, respectively.

At 20 TPS with flat terrain (light red), Opencraft 2 supports up to 250 connected players before dropping below the 20 TPS threshold. In Figure 5.5b we compare this value of maximum supported players to those found by Donkervliet et al 2023 [13] for Minecraft, Servo, and Opencraft under similar workload and on comparable hardware. Opencraft 2 supports 127% more players than most popular commercial MLG, Minecraft, and 25% more than the existing state-of-the-art research MLGs, Servo and Opencraft.

We find that both increased TPS and terrain complexity strongly impact scalability. At 20 TPS with more complex terrain (dark red), the maximum number of players supported is limited to 100, a 60% decrease compared to the flat terrain workload.

At 60 TPS with flat terrain (light blue), only 50 players are supported above the 60 TPS threshold. In the worst case, when TPS is high and the terrain is complex (dark blue), only 25 players can connect before the server cannot maintain the tick rate.

From these trends, we find several implications. We observe that the scalability decrease from increasing TPS is multiplicative with the decrease caused by complexity of game state. It follows that to increase the maximum supported players, TPS and environment complexity must decrease. As these two parameters are typically static, this makes them important targets from a differentiated deployment perspective. For example, a differentiated deployment scheme that dynamically increases TPS, or conversely, limits environment complexity, may improve player experience when player count is low, and revert these changes to improve maximum supported player count when required.

6 Related Work

In this section, we present an overview of related literature. The core of our work relates to application-specific resource allocation of a distributed system. This is an eminent and timely research topic, thus there are many related publications. We restrict our discussion of related work to research that is directly comparable or relevant; we do not include resource allocation techniques for data-center wide scheduling or optimizations for applications that are not real-time constrained.

6.1. Online-Game Frameworks and Resource Management

Performance engineering of online games has been an active area of research for over two decades, resulting in the discovery and population of many of the online-game networking techniques explained in Section 2.3. Despite the long history and many groundbreaking advancements, the core performance issues facing modern online games are much the same as those facing online games more than two decades ago. Heterogeneous hardware, unreliable networking, and increasing complexity of online-game simulations remain crucial engineering challenges for contemporary game developers. In this section, we discuss key work towards these online-game engineering challenges and compare them to the approach we take in this work.

A seminal work in the field and a direct inspiration to Polka's design is Bharambe, Pang, & Seshan, 2006, who design, implement, and evaluate Colyseus, a distributed architecture for interactive multiplayer games [3]. The Colyseus architecture increases online-game scalability by distributing game simulation over multiple servers. Colyseus takes advantage of a game's tolerance for weak inconsistency and predictable workload to distribute the authoritative control of individual in-game objects across multiple server nodes. Each node has read-only access to all objects, but must request up-to-date object state from the node that has authority of it. Through pre-fetching object state using a simple location-based area-of-interest (AOI) policy, latency from fetching state is mitigated. While Colyseus improves online-game server scalability in terms of maximum supported players, it does not scale well in terms of supported object counts or maximum number of nodes. Similarly to other AOI based techniques, it cannot decrease the performance impact of particularly high-density areas of the game environment. Colyseus also does not change the requirements and performance of the online-game client.

Dynamic handling of AOI partitioning technique is published in Diaconu & Keller, 2016, in their Kiwano system [11]. Kiwano splits the game environment into "cells," with each server node responsible for one cell. The amount and size of cells is dynamically updated to better load-balance the total number of objects being handled by each cell. Updates within each cell are propagated by sending state to neighboring cells. Like Colyseus, Kiwano's zoning mechanism is inadequate for high-density game areas, and does not effect the online-game client.

Shen et al, 2015, introduce area-of-simulation (AOS), a scalability mechanism that dynamically trades-off consistency for scalability [41]. AOS allows players to have multiple AOI that are not necessarily located nearby in the game environment. Like in Colyseus and Kiwano, Shen et al's AOS architecture splits the game environment into regions with each region being run by a single server node. Unlike Colyseus and Kiwano, AOS has clients performing simulation locally in a form of client-side prediction. What regions a client is simulating, receiving updates from, or not receiving any updates from, are dynamically updated based on player activity.

Jiang et al, 2017 propose Mirror, a policy-based computation offloading system for mobile games [27]. The Mirror technique characterizes behaviors in the online game as either “fine-grained," not large but frequently called, and “course grained," large but infrequent. Configurable policies then allow offloading percentages of each category of function to a remote server. Jiang et al focus on optimizing the performance of a single-player, rather than online, game, but their concept of course and fine-grained behaviors and the associated policies may have comparable functionality in the context of differentiated deployment.

Ghobaei-Arani, Khorsand, & Ramezanpour, 2019, design an autonomous resource-provisioning framework for MMOGs in a Cloud environment [23]. This framework is designed to reduce resource over-provisioning for online games hosted on the cloud, by continuously monitoring and predicting system load and using a fuzzy decision tree algorithm to estimate necessary resources. Similar to our work, Ghobaei-Arani et al propose using game-specific knowledge to inform resource allocation. However, their framework operates at the cloud host level, considering the online-game system only from view of servers hosted in the cloud. It does not receive information directly from game clients or affect their operation. Additionally, their framework expects online-game servers to use statically sized environment zones.

The Polka framework supports differentiated deployment of the rendering component. Shi & Hsu, 2015, survey interactive remote rendering systems [43]. Many of the techniques they identify for latency-hiding mechanisms are applicable to Polka’s remote rendering approach, and should be explored in the future work. These techniques include pre-fetching frames in advance, performing frame prediction through warping previously received images, and accelerating the rendering pipeline with game-specific knowledge provided by the game engine.

6.2. Service Allocation and Migration in the Edge

Online games, and other latency-sensitive applications, have been identified as workloads of interest for *edge* computing [28]. Cloud computing is a popular choice for hosting both online-game servers and hosting cloud gaming. However, cloud computing datacenters can be physically distant from many end users, which increases network latency to unacceptable amounts for real-time applications. Edge computing mitigates latency by hosting applications in *cloudlets*, smaller computing centers that are physically closer to end users. Resource management paradigms used in cloud computing are not sufficient for edge computing [28, 62]. Cloudlets have limited resources and must be physically close to end-users, who may be moving between cloudlet ranges. Many resource allocation and scheduling frameworks have been proposed for real-time applications on the edge, and there is conceptual overlap between these frameworks and Polka. We explain in this section work related to our approach, and compare them to Polka.

Zhang et al, 2017, design and implement EC+, a hybrid game architecture that splits game services between edge and cloud [62]. EC+ identifies two classes of events: local view changes and global game events. Local view change events have tight latency requirements, but do not need to be distributed to more than one player. Global game events do need to be distributed to all players. Thus, EC+ handles local view change on edge nodes for immediate response, and global game event handling on cloud nodes for scalability. The EC+ framework includes a service placement and migration policy as users move between access points. While EC+ does not perform fine-grained differentiated deployment of specific game services across both server and client, it does perform limited dynamic migration of the online game during runtime. Handling local view changes separately and placing importance on the rendering service is shared between Polka and EC+, and the migration mechanism in EC+ may be a useful template for developers creating service operation policies in Polka.

Talaria is a integrated migration mechanism for online games in the edge, designed and implemented by Braud, Alhilal, & Hui, 2021 [4]. Much like Polka’s migration mechanism, Talaria does not migrate the entire online game, but instead proactively pre-loads a game server, then migrates specific game state based on a prioritization policy. Talaria has a three stage migration mechanism: first, migrating any essential game state to immediately resume game operation, then all visible game objects, and finally any remaining game state. The implementation of Talaria’s migration mechanism is incompatible with PolkaDOTS, but an adaptation of the three stage migration mechanism to Polka’s service graph-based migration may be beneficial, especially in an edge environment.

Maheshwari et al, 2017, provide an in-depth performance evaluation of real-time applications run on the edge [30]. They measure application response time under a variety of workloads, deployments, and network conditions. Using that performance evaluation, they recommend an edge resource allocation approach called ECON, which uses a centralized resource controller with a complete network view. The centralized controller prioritizes placement on un-saturated edge nodes according to node location and recent latency values. The ECON strategy performs better than a cloud-only baseline when edge nodes are not saturated. While the performance evaluation done by Maheshwari et al is not specific to online games, their evaluation of the effect of network conditions on response time of edge applications is valuable for informing Polka service operation policies.

EC+ and the ECON approach are primarily concerned with the selection and migration of applications services between edge nodes, which are assumed to be largely homogeneous. In practice, cloudlet distance, resources, and availability vary considerably, thus creating a spectrum between true “edge" and “cloud". The term “fog computing" is used to refer to computing services including edge and cloud nodes as well as everything in-between. Yousefpour et al, 2019, introduce FOGPLAN, a framework for QoS-aware dynamic service provisioning on fog nodes [60]. Yousefpour et al also provide two greedy algorithms for dynamically adjusting service placement with FOGPLAN, one optimizing cost and another optimizing QoS. FOGPLAN is not designed for online games, and indeed makes assumptions that are incompatible with the stateful nature of online-game operation. However, several conceptual contributions of the FOGPLAN approach are transferable to the context of online games. This includes considering the entire fog spectrum as deployment targets rather than the edge/cloud binary, as well the design of the two greedy algorithms which may generalize well to service operation policies.

Rejiba, Masip-Bruin, & Marín-Tordera, 2019, survey the policies and mechanisms for service migration between edge nodes [36]. When to perform a service migration and how to perform it are important questions for all forms of resource allocation, but are especially crucial to the operation of applications on edge nodes. User mobility between access points can mandate service migration at any time, and handling that migration without service interruption is crucial to real-time applications. The authors find that what exactly is migrated varies between frameworks and applications, and that there are several migration strategies. Using their taxonomy, Polka uses a stateful, live, hybrid migration strategy, with a QoS optimization target. Exploration of other migration strategies and optimization targets is a promising direction for future research involving Polka.

An alternative view of resource management in edge and cloud deployment is given in Zakarya et al, 2022, who design and implement the epcAware resource allocation technique [61]. epcAware views resource allocation through a lens of adversarial game theory. Conceptually, there are three players: the Infrastructure as a Service (IaaS), Software as a Service (SaaS), and Network as a Service (NaaS) providers, each with different wants and conflicting aims. Effectively, this converts resource allocation into a multi-objective optimization problem to allow each provider type to maximize their specific wants. In epcAware, service allocation and migration is modelled as an auction, with provider systems bidding using promised resource. Viewing resource allocation in edge and cloud computing as a competition between business interests is a realistic lens, but primarily useful for performing resource allocation at an infrastructure, rather than application, level.

7 Conclusion and Future Research

In this chapter, we analyze our work in the context of the current state-of-the-art, and present a variety of promising directions for future work based off both our own findings and adapting existing techniques to Polka. We then conclude with an overview of our conceptual and technical contributions and their practical implications.

7.1. Future Improvements

We intend the Polka framework and Opencraft 2 to act as platforms for future research. We have therefore focused on the core features of the PolkaDOTS design. Accordingly, there are many potential improvements for future work to explore. In this section, we outline the limits of Polka and Opencraft 2 and discuss how they can be improved.

7.1.1. Dynamic and Differentiated Deployment

The differentiated deployment mechanisms in the design of Polka and their implementation in PolkaDOTS are not definitive. Our evaluation of building a game on the PolkaDOTS framework and the resulting performance is only done on a single genre of online game (Minecraft-like); other genres may yield different performance characteristics and require different implementation of their features to integrate with PolkaDOTS. Beyond differences in genre, there are several potential improvements to the Polka's differentiated deployment mechanisms.

In our evaluation, we focused on differentiated deployment of the rendering component. Since this is a novel functionality enabled by our work, further research in this direction is required. The design of service operation policies determining when to switch to a streamed gaming mode and where to host it require further data to determine feasibility. Similarly, service migration mechanisms can be improved through using render-pipeline specific information. For instance, a rendering service migration system could determine what exact game state is currently visible, and prioritize synchronizing that data.

PolkaDOTS does not currently support partitioning techniques to enable horizontal scaling across multiple server nodes. This is an established and proven technique both in research and in-industry [3, 11, 41], though out of scope for our proof-of-concept implementation. We expect (dynamic) partitioning and other horizontal scaling techniques to be a primary method of improving the performance of differentiated deployment techniques beyond what we have implemented.

A considerable amount of existing research on application migration in edge and fog computing may be applicable to Polka. Improvement to PolkaDOTS's service operation policy system should look to that field for guidance: many of their migration algorithms and formulation of resource allocation optimization problems are directly transferable to the domain of distributed online games. In particular, experiments utilizing Polka on commercially available edge cloudlets will be useful for designing service operation policies and mechanisms that handle user mobility between access points.

While online games are typically stateful, prior work has identified specific game services that can be operated either stateless or with extremely minimal state. For example, the environment terrain generation service requires no game state when it uses deterministic generation from a seed value. Such game services are of interest due to the possibility of offloading them using *serverless* functions. Serverless offloading allows applications to run stateless functions on commercial computing platforms, and benefit from the elasticity and scalability such platforms provide. The functionality to mark specific services as “stateless" and have PolkaDOTS automatically utilize serverless offloading is a future improvement worth exploring.

Finally, our system requires extensive engineering by game developers and operators to discover a viable configuration for their online game. Through further experimentation and data analysis it may be possible to create an auto-configuration tool to create baseline configurations for various online games. Such an auto-configuration tool may benefit from machine learning techniques to determine viable configuration parameters and service operation policies in the wide design space.

7.1.2. Application-Level Multi-View Remote Rendering

Remote rendering is the practice of offloading rendering workload to remote systems, allowing complex and demanding visual applications to operate on low-powered devices. A popular use-case of remote rendering is streamed gaming, which uses a thin client wrapper around existing games. Polka supports integrated remote rendering, moving the thin-client and remote rendering hosting logic into the application layer, rather than as an external wrapper. However, PolkaDOTS currently does not take advantage of many potential optimization opportunities this configuration exposes.

Shi & Hsu in their 2015 survey on interactive remote rendering systems cite several techniques for optimizing remote rendering [43]. Some of the optimization techniques they list have become standard practice for streamed gaming, specifically the use of custom network protocols and using adaptive bitrate to optimize bandwidth usage. However, many techniques are incompatible with existing streamed gaming systems, which do not have access to a game's internal architecture. These techniques are promising candidates for integration into the PolkaDOTS framework.

Because PolkaDOTS can access Unity's rendering pipeline, it may be possible to benefit from *Jointly Encoding*. Joint encoding uses information provided by the game engine to optimize remote-rendering. This includes using 3D scene context such as rendering viewpoints, depth, and movement to predict future frames as well optimize video stream compression through augmenting edges or identifying foreground and background.

Since the thin client is also on the application layer in PolkaDOTS, potential improvements could modify it for specific games. For instance, a thin client could perform limited *pre-fetch*, predicting multiple possible player inputs and requesting frames for them in advance. Alternatively, the thin client could perform *image-based rendering*, which uses both previously received video frames and information on the 3D scene to warp the existing view to the requested one. Functionally, both techniques increase the computational demand of the thin client but hide latency caused by remote rendering.

Other possibilities for optimizing PolkaDOTS's integrated remote rendering not discussed by Shi & Hsu stem from Polka's support of multi-user remote rendering. In Polka, one client can act as a remote rendering host for multiple thin clients. PolkaDOTS already benefits from this: the meshing process for terrain chunks needs only be performed once per host, rather than per player. The same mesh data is used when rendering frames for all thin clients on that host. Further improvement could use a form of joint encoding, using information from one rendering pipeline to provide context to others.

7.1.3. The Online-Game Ecosystem

Online games are complex distributed systems that do not act in isolation, but instead communicate with and are managed by a variety of systems through their life-cycle. Online games are continuously developed and updated after release, operations management system deploy and monitor game services, and of course, users connect to these systems as players, operators, and content creators. Integrating Polka with Unity as PolkaDOTS allows our work to integrate into Unity's wider game development ecosystem. However, there are still many aspects of online gaming as a whole that PolkaDOTS does not specifically handle.

PolkaDOTS does not provide a built-in technique for saving and restoring persistent game state, instead requiring game developers to add this feature manually. Since performing consistent state persistence while supporting Polka's differentiated deployment mechanism is non-trivial but outside the scope of the current project, future work extending PolkaDOTS framework in this direction is recommended.

A research topic that has been continually present in work on online games is concerned with online-game security. In many online games, players have taken advantage of design oversights to gain unfair advantages. Such player exploits are called *cheats*. PolkaDOTS does not have any explicit cheat detection or prevention measures, but could be a useful platform for developing anti-exploit measures in future work. For instance, an *action validation* service may be well suited for differentiated deployment, and could be configured to be deployed only when an exploit is suspected. Similarly, players that are suspected of performing exploits could be relegated to a streamed-gaming only mode, as it is difficult, if not impossible, to modify the behavior of render streaming hosts running on remote systems.

However, the same properties that make PolkaDOTS good for exploring anti-exploit measure may make achieving beneficial features more difficult. Not all player modifications to game behavior is harmful, indeed an extremely popular phenomenon is *modding*: users modifying game files to add, modify, or remove content. Modding is generally considered beneficial, they allow an online-game's community to supplement a game's development with desired features, patch specific bugs, or improve performance. Mods that are popular may eventually be added to the game by its developers, and in rare cases, become a separate game in their own right. However, PolkaDOTS makes online games more complex that the traditional server-client model: requirements imposed by differentiated deployment and integrated remote rendering make user modding infeasible. Supporting user modifications to PolkaDOTS and Opencraft 2 through a *modding interface* would be a significant improvement. Such an interface may take the form of a script language to allow users to write custom services to be deployed alongside built-in services. It may also be possible to develop a *modding framework*, which would allow users to create modded versions of an entire game built on PolkaDOTS, though this approach requires users to built the game from its source using the Unity editor.

Another way online games interact with a broader ecosystem is through acting as a content creation platform. This includes content created *within* the game: player-made content such as environment structures, player-lead storytelling, and community building, as well as content created *of* the game. Online gaming is incredibly popular as a spectacle: with gameplay videos achieving billions of views on video publishing websites such as YouTube, and millions of viewers on live-streaming platforms such as Twitch. There are a variety of methods in which the PolkaDOTS framework could interact and integrate with content creation and distribution. For instance, using multicast networking with the integrated remote rendering would allow PolkaDOTS to support real-time spectating to a large audience.

7.1.4. Improvements to Opencraft 2

The conceptual and technical contributions of this work primarily relate to Polka. However, we intend the online game we design on the PolkaDOTS framework, Opencraft 2, to act as a research platform for future research on MLGs. To the best of our knowledge, Opencraft 2 is the only open source MLG implemented in a modern game development ecosystem. Another benefit of Opencraft 2 compared to other open source MLGs is its completeness: it allows modifying not only MLG a server implementation, but an associated client front-end, the networking protocols used between them, and the compilation and packaging pipeline used to build and deploy them.

However, as of the date this report is published Opencraft 2 is a minimal MLG, it does not contain many of the more complex features present in many MLGs. These include distinctive functionalities of MLGs such as dynamic terrain that modifies itself over time, or the presence of non-player characters which interact both with the modifiable environment and with player avatars. These systems, if implemented in future work, would both improve Opencraft 2's similarity to popular MLGs but also enable experiments involving these features.

Finally, PolkaDOTS uses player behavior simulation for generalizable player emulation. However, player simulation does not function in experiments involving thin clients as they do not have access to the necessary game state. Object recognition and deep learning could theoretically allow thin clients to respond to the game world through the view they receive in a video stream. Implementing deep learning based player emulation and performing experiments would yield interesting results but would prohibitive in terms of necessary hardware resources.

7.2. Conclusion

Video gaming is one of the largest entertainment industries, and online games have become extremely popular. Online games have attempted to create continuously more detailed and expansive worlds while connecting larger numbers of players. However, online games remain limited by their static deployment methods which have remained largely unchanged for two decades and leave them incapable of responding to changes in resource availability in their highly dynamic and heterogeneous environment.

In this work, we introduce the concept of differentiated deployment, a novel deployment method which incorporates resources management into the application layer in order to dynamically modify the deployment of an online game in response to changes resource availability.

To explore the potential of differentiated deployment, we design Polka, a framework for dynamic differentiated deployment in online games. We implement a prototype of Polka named PolkaDOTS in Unity, an industry standard game ecosystem.

Finally, we evaluate PolkaDOTS by using it to design and build Opencraft 2, a representative Minecraft-like Game. We then create an experiment deployment tool Dither to run reproducible, real-world and large scale experiments with Opencraft 2.

We find that differentiated deployment can modify game deployments with minimal interruption to player experience and result in increased performance and reduced resource consumption. Additionally, we find that differentiated deployment enables previously unfeasible deployment options, such as switching from local to remote rendering at runtime with no interruption to a player's gaming session.

In future work, we intend to expand on PolkaDOTS's differentiated deployment mechanisms to improve distributed performance monitoring, allow arbitrary game systems to be migrated, and further explore the possibilities of dynamic remote rendering.

A
Example Configuration Files for Experiment Run Using Dither

In Listing A.1 we show an example configuration file which Dither uses to automatically deploy and configure infrastructure. In this case, Dither uses the provided GCP credentials at *credentials.json* to interact with the *opencraft2* GCP project. Within this project, it deploys 1 endpoint VM of the type *n1-standard-4* and 2 cloud VMs of the type *e2-standard-4* in the *europe-central2-c* zone.

Dither then configures each deployed VM with the pre-requisite software for the *kubernetes* resource manager and the *opencraft2* application. This includes setting a number of application parameters for the resource manager, such as the amount of CPU and memory resources to provide to the online-game instances. Finally, Dither copies the *LocalToStream.json* deployment configuration file to the cloud VMs.

The contents of *LocalToStream.json* are given in Listing A.2. This deployment configuration details a three node experiment where a local client switches to a streamed client configuration. The configuration is split into a set of *node* descriptors, which tell the PolkaDOTS game how to perform remote deployment configuration of three three nodes, and a set of *experimentActions* to take during the experiment. The first node, with a *nodeID* of 0, runs a *Server* world and starts listening for connections immediately. The node with *nodeID* 1 creates a *Client* and *StreamedClient* world, but only connects the *Client* world to the *Server* world run on the node with *nodeID* 0. The final node, with *nodeID* 2, starts a *HostClient* but does not connect it to the *Server*.

The list of *experimentActions* is a set of actions for the deployment component to take and an associated trigger to take the action. In this case, there is a single *experimentAction* set to trigger after an timed delay of 120 seconds. When this action triggers, the *Client* on node 1 is stopped and the *StreamedClient* is connected, in this case to the *HostClient* on node 2. At the same time, that *HostClient* on node 2 is connected to the *Server* on node 0.

Listing A.1: Example experiment configuration file.

```
[infrastructure]
provider = gcp
# GCP details
gcp_region = "europe-central2"
gcp_zone = "europe-central2-c"
gcp_project = "opencraft2"
gcp_credentials = "~/credentials.
    json"

# Endpoint VM details
endpoint_nodes   = 1
gcp_endpoint     = "n1-standard-4"
use_gpu_endpoint = True
endpoint_cores = 4
endpoint_memory = 15
endpoint_quota = 1

# Cloud VM details
cloud_nodes     = 2
gcp_cloud       = "e2-standard-4"
use_gpu_cloud = False
cloud_cores = 4
cloud_memory = 16
cloud_quota = 1

base_path = ~/experiment

[benchmark]
resource_manager = kubernetes
application = opencraft2
opencraft_experiment_duration = 180
# Only run 1 server in cloud
applications_per_worker = 1
# Number of clients per endpoint
    node
applications_per_endpoint = 1
# What deployment configuration to
    use
deployment_config="LocalToStream.
    json"
# CPU (in cores) and memory (in GB)
application_worker_cpu = 2.0
application_worker_memory = 8.0
# list of cpu percents by
    application
opencraft_endpoint_cpu = 4.0,4.0
application_endpoint_memory = 8.0
```

Listing A.2: LocalToStream.json: example deployment configuration file.

```
"nodes":[
 {
  "nodeID":0,
  "worldConfigs":[
    {
    "worldName": "GameServer",
    "worldType":"Server",
    "initializationMode":"Connect",
    }
  ]
 },
 {
  "nodeID":1,
  "worldConfigs":[
    {
    "worldName": "GameClient",
    "worldType":"Client",
    "initializationMode":"Connect",
    "serverNodeID":0,
    },
    {
    "worldName": "StreamedClient",
    "worldType":"Client",
    "initializationMode":"Create",
    "multiplayStreamingRoles":"
        Guest",
    "streamingNodeID":2,
    }
  ]
 },
 {
  "nodeID":2,
  "worldConfigs":[
    {
    "worldName": "CloudHostClient",
    "worldType":"Client",
    "initializationMode":"Start",
    "multiplayStreamingRoles":"
        CloudHost",
    "serverNodeID":0,
    }
  ]
 }
],
"experimentActions":[
 {
  "delay": 120,
  "actions": [
    {
    "nodeID": 1,
    "worldNames": ["GameClient", "
        StreamedClient"],
    "actions": ["Stop", "Connect"]
    },
    {
    "nodeID": 2,
    "worldNames": ["CloudHostClient
        "],
    "actions": ["Connect"]
    }
  ]
 }
]
```

Acronyms

AOI	area-of-interest. 49
AOS	area-of-simulation. 49
CSV	comma-separated value. 23
DOP	data-oriented programming. 21, 22
DOTS	data-oriented technology stack. vii, 21–23, 26, 27, 29, 31–34
ECS	entity-component system. 21, 22, 27–30, 32–34, 37
FPS	frames per second. 43, 46, 47
GCP	Google Cloud Platform. 42
IaaS	Infrastructure as a Service. 51
LAN	local area network. 6, 7, 12, 13, 44
MLG	Minecraft-like Game. 3, 4, 25–29, 33–37, 39, 47, 55
MMOG	massively multiplayer online games. 5, 47, 50
MVE	modifiable virtual environment. vii, 25–29, 36, 37
NaaS	Network as a Service. 51
NFE	Netcode for Entities. 21, 22, 28–30, 34, 36, 37, 42
NPC	non-player character. 16, 26, 27
OGS	Online-Game Services. 5, 6, 8, 9
OOP	object-oriented programming. 21, 23
P2P	peer to peer. 6
PCG	procedural content generation. 25–28, 30, 31
QoE	quality of experience. 12, 13
QoS	Quality of Service. 16, 51
RPC	remote procedure call. 34
RTT	round-trip time. 23, 43, 45
SaaS	Software as a Service. 51
TPS	ticks per second. 36, 47
UGS	Unity Gaming Services. 20
VM	virtual machine. 40–42, 57
VR	virtual-reality. 44

Bibliography

[1] ACM. Artifact review and badging version 1.1. https://www.acm.org/publications/policies/artifact-review-and-badging-current, 2020.

[2] Emery D. Berger, Stephen M. Blackburn, Matthias Hauswirth, and Michael W. Hicks. A checklist manifesto for empirical evaluation: A preemptive strike against a replication crisis in computer science. https://blog.sigplan.org/2019/08/28/a-checklist-manifesto-for-empirical-evaluation-a-preemptive-strike-against-a-replication-crisis-in-computer-science, 2019. [Online; accessed 9-Dec-2021].

[3] Ashwin R Bharambe, Jeffrey Pang, and Srinivasan Seshan. Colyseus: A distributed architecture for online multiplayer games. In *NSDI*, volume 6, pages 12–12, 2006.

[4] Tristan Braud, Ahmad Alhilal, and Pan Hui. Talaria: In-engine synchronisation for seamless migration of mobile edge gaming instances. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, pages 375–381, 2021.

[5] Chris Byskal. Moving from p2p to cloud: How for honor & friday the 13th the game improved player experience. https://aws.amazon.com/blogs/gametech/for-honor-friday-the-13th-the-game-move-from-p2p-to-the-cloud-to-improve-player-experience/, Mar 2018. [Online; accessed 7-June-2023].

[6] Caleb Chen. Activists in minecraft made a digital library to bypass government censorship. https://www.privateinternetaccess.com/blog/activists-in-minecraft-made-a-digital-library-to-bypass-government-censorship/, 2020.

[7] Frederikke Christiansen. Online activism meets digital gaming: Protesters are now taking to the virtual streets. https://mastersofmedia.hum.uva.nl/blog/2020/09/27/online-activism-meets-digital-gaming-protesters-are-now-taking-to-the-virtual-streets/, 2020.

[8] World Wide Web Consortium. Webrtc: Real-time communication in browsers. https://www.w3.org/TR/webrtc/, 2023.

[9] NVIDIA Corporation. Nvidia video codec sdk. https://developer.nvidia.com/video-codec-sdk, 2013.

[10] Andrea Di Domenico, Gianluca Perna, Martino Trevisan, Luca Vassio, and Danilo Giordano. A network analysis on cloud gaming: Stadia, geforce now and psnow. *Network*, 1(3):247–260, 2021.

[11] Raluca Diaconu and Joaquín Keller. Kiwano: Scaling virtual worlds. In *2016 Winter Simulation Conference (WSC)*, pages 1836–1847. IEEE, 2016.

[12] Jesse Donkervliet, Jim Cuijpers, and Alexandru Iosup. Dyconits: Scaling minecraft-like services through dynamically managed inconsistency. In *41st IEEE International Conference on Distributed Computing Systems, ICDCS 2021, Washington DC, USA, July 7-10, 2021*, pages 126–137, 2021. doi: 10.1109/ICDCS51616.2021.00021. URL https://doi.org/10.1109/ICDCS51616.2021.00021.

[13] Jesse Donkervliet, Javier Ron, Junyan Li, Tiberiu Iancu, Cristina L. Abad, and Alexandru Iosup. Servo: Increasing the scalability of modifiable virtual environments using serverless computing. In *43st IEEE International Conference on Distributed Computing Systems, ICDCS 2023, Hong Kong, China, July 17-21, 2023*. IEEE, 2023.

[14] Liam Doolan. Nintendo is "replacing its multiplayer server system" dating back to the wii u and 3ds era. https://www.nintendolife.com/news/2021/02/nintendo_is_replacing_its_multiplayer_server_system_dating_back_to_the_wii_u_and_3ds_era, Feb 2021. [Online; accessed 7-June-2023].

[15] Jeff Drake. 24 great games that use the unity game engine. https://www.thegamer.com/unity-game-engine-great-games/, 2023.

[16] Binny Edmondson. Why are graphics card so expensive. https://robots.net/tech/why-are-graphics-card-so-expensive/, 2023.

[17] Jerrit Eickhoff. Opencraft 2. https://github.com/atlarge-research/Opencraft-2, 2023.

[18] Jerrit Eickhoff. PolkaDOTS. https://github.com/atlarge-research/PolkaDOTS, 2023.

[19] Jerrit Eickhoff. Polka data and plotting tools, March 2024. URL https://zenodo.org/doi/10.5281/zenodo.10794249.

[20] Jerrit Eickhoff, Jesse Donkervliet, and Alexandru Iosup. Meterstick: Benchmarking performance variability in cloud and self-hosted minecraft-like games. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*, pages 173–185, 2023.

[21] Epic. Epic online services. https://dev.epicgames.com/en-US/services, 2023. [Online; accessed 7-June-2023].

[22] Bungie Forum. Destiny's networking and p2p connectivity: What you should know. https://www.bungie.net/en/Forums/Post/261526661?path=0, 8 2022. [Online; accessed 7-June-2023].

[23] Mostafa Ghobaei-Arani, Reihaneh Khorsand, and Mohammadreza Ramezanpour. An autonomous resource provisioning framework for massively multiplayer online games in cloud environment. *Journal of Network and Computer Applications*, 142:76–97, 2019.

[24] William H Guss, Brandon Houghton, Nicholay Topin, Phillip Wang, Cayden Codel, Manuela Veloso, and Ruslan Salakhutdinov. Minerl: A large-scale dataset of minecraft demonstrations. *arXiv preprint arXiv:1907.13440*, 2019.

[25] Alexandru Iosup, Laurens Versluis, Animesh Trivedi, Erwin Van Eyk, Lucian Toader, Vincent van Beek, Giulia Frascaria, Ahmed Musaafir, and Sacheendra Talluri. The atlarge vision on the design of distributed systems and ecosystems. In *39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, TX, USA, July 7-10, 2019*, pages 1765–1776. IEEE, 2019.

[26] Matthijs Jansen, Linus Wagner, Animesh Trivedi, and Alexandru Iosup. Continuum: Automate infrastructure deployment and benchmarking in the compute continuum. In *Proceedings of the First FastContinuum Workshop, in conjuncrtion with ICPE, Coimbra, Portugal, April, 2023*, 2023. URL https://atlarge-research.com/pdfs/2023-fastcontinuum-continuum.pdf.

[27] M.H. Jiang, O.W. Visser, I.S.W.B. Prasetya, and A. Iosup. Mirror: A computation-offloading framework for sophisticated mobile games. In *2017 IEEE 18th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pages 1–3, 2017. doi: 10.1109/WoWMoM.2017.7974351.

[28] Li Lin, Xiaofei Liao, Hai Jin, and Peng Li. Computation offloading toward edge computing. *Proceedings of the IEEE*, 107(8):1584–1607, 2019.

[29] Bryan Lufkin. How online gaming has become a social lifeline. https://www.bbc.com/worklife/article/20201215-how-online-gaming-has-become-a-social-lifeline, 2020.

[30] Sumit Maheshwari, Dipankar Raychaudhuri, Ivan Seskar, and Francesco Bronzino. Scalability and performance evaluation of edge cloud systems for latency constrained applications. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 286–299, 2018. doi: 10.1109/SEC.2018.00028.

[31] Microsoft. Azure playfab. https://azure.microsoft.com/en-us/products/playfab/, 2023. [Online; accessed 7-June-2023].

[32] Mirror. Mirror networking. https://mirror-networking.gitbook.io/docs/, 2023. [Online; accessed 7-June-2023].

[33] Normcore. Normcore - seamless multiplayer for unity. https://normcore.io/, 2023. [Online; accessed 7-June-2023].

[34] Alessandro Vittorio Papadopoulos, Laurens Versluis, André Bauer, Nikolas Herbst, Jóakim von Kistowski, Ahmed Ali-Eldin, Cristina L. Abad, José Nelson Amaral, Petr Tuma, and Alexandru Iosup. Methodological principles for reproducible performance evaluation in cloud computing. *IEEE Transactions on Software Engineering*, 47(8):1528–1543, 2021.

[35] Photon. Photon. `https://www.photonengine.com/`, 2023. [Online; accessed 7-June-2023].

[36] Zeineb Rejiba, Xavier Masip-Bruin, and Eva Marín-Tordera. A survey on mobility-induced service migration in the fog, edge, and related computing paradigms. *ACM Computing Surveys (CSUR)*, 52(5):1–33, 2019.

[37] AtLarge Research. Opencraft 2. `https://github.com/JerritEic/continuum`, 2023.

[38] Mitchell Robinson. Voxel world optimisations. `https://vercidium.com/blog/voxel-world-optimisations/`, 2019.

[39] Saeed Shafiee Sabet, Steven Schmidt, Saman Zadtootaghaj, Babak Naderi, Carsten Griwodz, and Sebastian Möller. A latency compensation technique based on game characteristics to mitigate the influence of delay on cloud gaming quality of experience. In *Proceedings of the 11th ACM Multimedia Systems Conference*, pages 15–25, 2020.

[40] Amazon Web Services. Aws for games. `https://aws.amazon.com/gametech/`, 2023. [Online; accessed 7-June-2023].

[41] Siqi Shen, Shun-Yun Hu, Alexandru Iosup, and Dick H. J. Epema. Area of simulation: Mechanism and architecture for multi-avatar virtual environments. *TOMCCAP*, 12(1):8:1–8:24, 2015. doi: 10.1145/2764463. URL `http://doi.acm.org/10.1145/2764463`.

[42] Arjun Darshan Sheth. The impact of minecraft: Education edition. `https://gamerant.com/minecraft-education-edition-impact/`, 2022.

[43] Shu Shi and Cheng-Hsin Hsu. A survey of interactive remote rendering systems. *ACM Computing Surveys (CSUR)*, 47(4):1–29, 2015.

[44] SteamDB. Steamdb charts. `https://steamdb.info/graph/`, Oct 2022.

[45] Mojang Studios. Minecraft. `https://www.minecraft.net/en-us`, 2023. [Online; accessed 7-June-2023].

[46] Unity Technologies. Inputrecorder.cs. `https://github.com/Unity-Technologies/InputSystem/blob/develop/Assets/Samples/InputRecorder/InputRecorder.cs`, 2020.

[47] Unity Technologies. Unity gaming services. `https://unity.com/solutions/gaming-services`, 2023. [Online; accessed 7-June-2023].

[48] Unity Technologies. Input system. `https://docs.unity3d.com/Packages/com.unity.inputsystem@1.8/manual/index.html`, 2023.

[49] Unity Technologies. Mobile games. `https://unity.com/solutions/mobile`, 2024.

[50] Joe Tidy. Fortnite's travis scott virtual concert watched by millions. `https://www.bbc.com/news/technology-52410647`, 2020.

[51] Unity3D. Netcode for gameobjects. `https://docs-multiplayer.unity3d.com/`, 2023. [Online; accessed 7-June-2023].

[52] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan S. Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. Is big data performance reproducible in modern cloud networks? In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 513–527. USENIX Association, 2020.

[53] Valve. Steam remote play. `https://store.steampowered.com/streaming/#together`, 2023. [Online; accessed 6-June-2023].

[54] Valve. Steamworks. https://partner.steamgames.com/, 2023. [Online; accessed 7-June-2023].

[55] Valve. Source multiplayer networking. https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking, 2023. [Online; accessed 7-June-2023].

[56] Jerom van der Sar, Jesse Donkervliet, and Alexandru Iosup. Yardstick: A benchmark for minecraft-like services. In Varsha Apte, Antinisca Di Marco, Marin Litoiu, and José Merseguer, editors, *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, ICPE 2019, Mumbai, India, April 7-11, 2019*, pages 243–253. ACM, 2019. doi: 10.1145/3297663.3310307. URL https://doi.org/10.1145/3297663.3310307.

[57] Tom Wijman. Newzoo's year in review: the 2023 global games market in numbers. https://newzoo.com/resources/blog/video-games-in-2023-the-year-in-numbers, Dec 2023.

[58] Wilkinson et al. The FAIR Guiding Principles for scientific data management and stewardship. *Nature SciData*, 3, 2016.

[59] Tianyin Xu and Yuanyuan Zhou. Systems approaches to tackling configuration errors: A survey. *ACM Computing Surveys (CSUR)*, 47(4):1–41, 2015.

[60] Ashkan Yousefpour, Ashish Patil, Genya Ishigaki, Inwoong Kim, Xi Wang, Hakki C Cankaya, Qiong Zhang, Weisheng Xie, and Jason P Jue. Fogplan: A lightweight qos-aware dynamic fog service provisioning framework. *IEEE Internet of Things Journal*, 6(3):5080–5096, 2019.

[61] Muhammad Zakarya, Lee Gillam, Hashim Ali, Izaz Ur Rahman, Khaled Salah, Rahim Khan, Omer Rana, and Rajkumar Buyya. epcaware: A game-based, energy, performance and cost-efficient resource management technique for multi-access edge computing. *IEEE Transactions on Services Computing*, 15(3): 1634–1648, 2022. doi: 10.1109/TSC.2020.3005347.

[62] Wuyang Zhang, Jiachen Chen, Yanyong Zhang, and Dipankar Raychaudhuri. Towards efficient edge cloud augmentation for virtual reality mmogs. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350877. doi: 10.1145/3132211.3134463. URL https://doi.org/10.1145/3132211.3134463.

[63] Yu Zhonggen et al. A meta-analysis of use of serious games in education over a decade. *International Journal of Computer Games Technology*, 2019, 2019.

[64] Zack Zwiezen. Minecraft still the best-selling video game of all time (it's not even close). https://www.kotaku.com.au/2023/10/minecraft-still-the-best-selling-video-game-of-all-time-its-not-even-close/, 2023.