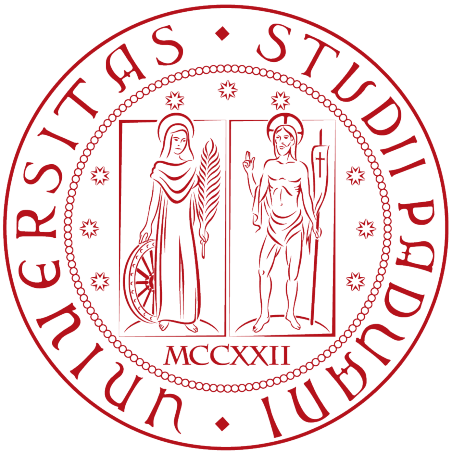
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Degree Course in Computer Science



A Practical Approach to Containerization and Distribution of Game Engines

Master degree thesis

Thesis supervisor

Prof. Claudio Enrico Palazzi

Co-advisors

Prof. Dario Maggiorini
Prof. Paolo Giaccone

Graduate student

Alessandro Sgreva

Academic Year 2022-2023

Summary

In the last years, the development of video game software has become increasingly harder, as their structure is getting ever more complex. As such, the role of Game Engines in this field is now crucial, for allowing a more efficient and effective reuse of functionalities which may be too expensive to develop from scratch for every single project.
Still, most popular Game Engines present a monolithic structure, which is problematic for state-of-the-art video game projects that are often aimed towards a multiplayer online environment. Resource scalability, in fact, is not easy with a monolithic structure and multiple software changes may require constant refactoring of the whole architecture.
Our work aims to address the shortcomings of monolithic Game Engines (Legacy Game Engines), by researching and implementing a distributed alternative (Distributed Game Engine), where the modules that compose the software are decoupled and hosted on different virtual containers.
Considering the required communication of these modules inside a network environment, we also consider and research the impact of network-specific elements on the performance of such system.

With the aim of preventing possible ambiguity, the technical terms used in the present document are clarified and elaborated on in the *appendix A - Glossary*. Furthermore, in order to facilitate the reading of the document, said terms are marked with a subscript *'G'*.

Special thanks

First of all, I would like to express my gratitude to Prof. Claudio Enrico Palazzi, supervisor of my thesis, and to professors Paolo Giaccone and Dario Maggiorini, for their availability and support during the project experience, as well as during the drafting of the thesis document.

I would like to thank my mother for the help, support and patience brought during these years of university study.

Finally, I would also like to thank my friends and colleagues for the company, the support and the experiences lived together.

Padova, February 2023 Alessandro Sgreva

Contents

List of Figures

List of Tables

Chapter 1

Introduction

*This chapter introduces the concept of **Game Engine**, discusses the main problems of **Legacy Game Engines** with respect to **Distributed Game Engines**, and presents the main purpose of our research work.*

1.1 Background & Problems

Nowadays, *Game Engines* have become a core element in the development of most video game software. They are often identified as a composite software, that is able to provide core and important functionalities for managing basic features of video games, such as: image rendering, physics management, animation and many more [1, 38]. These type of features, while still being paramount for the execution of a video game, are often expensive to implement from scratch for each new software being developed. Furthermore, considering their nature, the value they provide can often be reused or adapted to different projects, making the development process much more efficient.

On a more technical level, the Game Engines are comprised of a tool suite and a run-time component [9]. The tool suite allows creators to merge together various kinds of multimedia and audiovisual assets, while the run-time is a layered software that takes care of background operations (e.g. resource management, interaction with the hardware, ...) and is also transferred into the game executables, which has important consequences. In fact, even if many Game Engines present a certain degree of modularization, with clear separation of the provided functionalities in different modules, they are still designed with a monolithic structure.

This type of Game Engine, which we call *Legacy Game Engine*, is presented with the strong requirement of high performance and low delays for the user interfacing with the image rendered on screen. This is particularly problematic in contexts where the available resources are limited or network latency is present (e.g. multiplayer). Furthermore, due to their monolithic nature, even small changes can require extensive refactoring of their whole codebase and their interfacing with the hardware can introduce problematic platform dependency, which hinders portability.

In an effort to find a solution to these problems, research has been conducted on more modular and decentralized Game Engines [3, 7, 13], which we will call *Distributed Game Engines*. This type of Game Engine can be implemented with many different

approaches [7, 8, 13], but they are generally hosted on multiple physical or virtual machines. In particular, this structure decouples the GE_G functionalities in various modules (e.g. rendering, physics, AI_G, ...), which communicate with each other in order to provide the same features of the full Game Engine. The aim of this type of architecture is to provide the user with a low delay full-functioning Game Engine, without the limitations of a monolithic structure and with more flexible resource allocation for the single functionalities.

This paradigm has proven to be quite successful, especially in the context of *Cloud*$_G$ *Computing*, allowing the offloading of the most computation-intensive operations to dedicated hosting services [8, 13], thus overcoming many limitations of local hardware. However, when distributing the components of a software into a network environment, additional elements such as network latency are introduced into the picture. As such, when designing technical solutions, it is important reason on the impact on performance of positive and negative side effects [14].

In this context, even if some previous works have defined the requirements for such architecture [8, 17], little research has been conducted on a practical implementation, considering its possible problems and technological tools to implement it.

1.2 Purpose & Research questions

In this work, we verify the possibility of turning a Legacy Game Engine into a Distributed Game Engine, through modularization and containerization of its main components. We propose a Docker-based architecture where the Game Engine modules, such as: graphic engine, physics engine, AI_G, music player, ...; are connected and communicate in a network environment.
Additionally, we also introduce original functionalities fitting for a peer-to-peer$_G$ distributed system, such as game image remote streaming and game state synchronization, through dedicated middlewares.

Considering the requirement of distributing Game Engine data between various virtual containers, we also compare possible distributed database solutions (e.g. ETCD, Redis), understanding their positive and negative aspects, with the aim of identifying the most fitting solution for our purpose.

In order to provide a more practical approach to the design of our architecture, we also consider realistic problems of distributed network environments (e.g. network latency, network traffic, synchronization, ...) and perform dedicated experiments to quantify their impact on the system performance and functionalities.

To sum up, the main research questions this work aims to answer are as follows:

* is it possible to decouple and containerize Game Engine modules or libraries, while maintaining the original system functionalities?
* what are the most fitting and performing options for distributing Game Engine data across multiple components in a network environment?
* what are the effects of network latency and traffic on the performance of a Distributed Game Engine?

Chapter 2

State of the art

*This chapter discusses the **state-of-the-art** in relation to the scope this project, with a coverage of multiple topics which constitute the background of our work.*

2.1 An overview on Game Engines

2.1.1 Definitions

Game Engines (GEs) are generally defined as software frameworks designed for the development and execution of interactive multimedia software, namely video games. More specifically, we define video games as the subset of games that take place in a 2-3 dimensional virtual world, with a variable number of players [13]. Considering that:

* the world where they take place is time dependant (temporal or dynamic);
* the time in the simulated world is a mathematical approximation and simplification of reality;
* there are multiple distinct entities known as "agents" that interact in this world;

video games can be seen as *soft-time$_G$ interactive agent-based temporal computer simulations*, where the interaction of an user with an interface or input device provides them with a visual feedback [38, 9].

If we look at them on the surface level, it might be hard to differentiate Game Engines from video games software.
In this context, according to Jason Gregory in his book "Game Engine Architecture" [9], the main characteristic that defines a Game Engine is its *data-driven architecture*. Video games software usually contains hard-coded logic or game rules, to manage their functionalities through non-reusable code. On the other hand, Game Engines aim towards the possibility of creating multiple products with new elements and functionalities, through minimal changes performed to the original "engine" software.

As such, we can define a Game Engine as a *"piece of software that is extensible and can be used as the foundation for many different video games without major modifications"* [9].

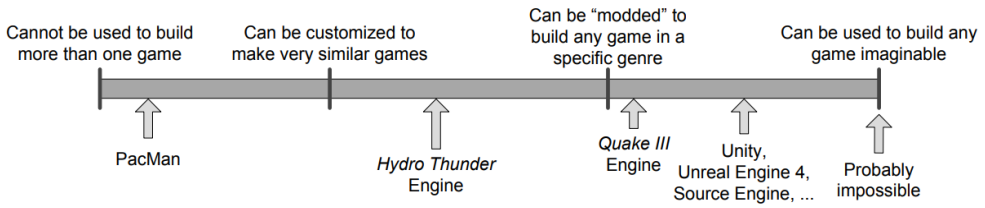


Figure 2.1: Game engine reusability gamut - Gregory.

2.1.2 Structure

As we discussed in the previous section, reusability is the most paramount characteristic of Game Engines. As such, Game Engines cannot be composed just of libraries and tools for merging multimedia assets [8]. A portion of Game Engine software needs to become a part of the developed video game itself, with the purpose of managing these reusable functionalities.

For this reason, Game Engines generally consist of two main components:

* a **tool suite**, which allows non-technical users to create and manage audiovisual assets, such as: 3D meshes, textures, sounds and animations. This component is left behind after the game development process, since it is not useful while playing [13];
* a **runtime component**, which is transferred into the game executable in order to manage resources, schedule events and implement complex functionalities (e.g. 2D-3D graphics rendering, physics management, collision detection) that are reusable across multiple video games [8, 9].

The runtime component, in particular, is structured in multiple layers with an increasing level of abstraction. [8]

As we can see in Figure 2.2, the lowest level acts as an interface to the kernel and the hardware the game is running on. This is especially important for the implementation of core video game functionalities, such as graphics rendering and physics simulation, which heavily rely on the processing performed by the physical hardware (e.g. CPU, GPU, memory).

On the other hand, the upper layers focus on providing an entry point for the game APIs_G, a certain degree of platform independence and game specific functionalities (e.g. AI_G, player mechanics, game cameras and online multiplayer capabilities).

2.1.3 Shortcomings of a monolithic architecture

Even if many Game Engines are carefully crafted and fine-tuned to run particular games on specific hardware platforms [9], in most of them the core original *monolithic architectural model* is still present.

As we discussed in the previous section, each instance of video game executable needs to embed the runtime component of the Game Engine used to develop it. This component runs alongside the main game instance, managing its core functionalities

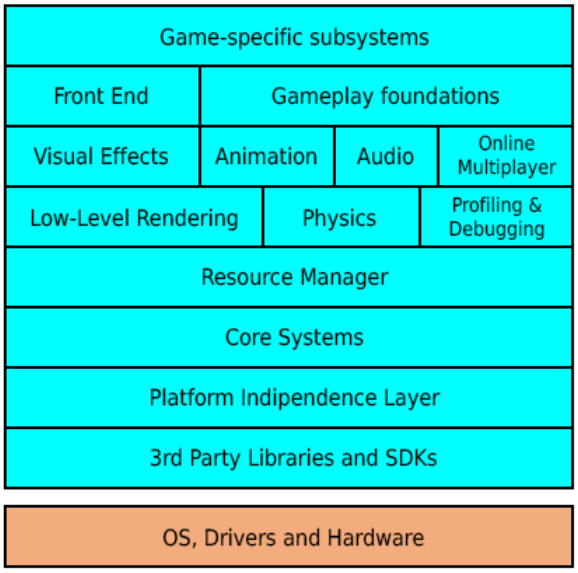


Figure 2.2: Simplified layered structure of the runtime component.

and thus creating an high degree of dependence on it.

Game Engines implementing such monolithic architecture, which we will call *Legacy Game Engines*, present three major shortcomings [8]:

1. while providing high performance, a monolithic software requires refactoring of the whole project (or large part of it) at every change in the codebase. This operation can become quite expensive in the long run and turn into a production bottleneck, especially in context where changes are frequent;

2. monolithic software operations processing can hardly be split among different server, as such we often run the risk of putting an excessive amount of CPU workload on a single centralized server. In this context, when resources are insufficient the centralized server becomes a bottleneck, significantly reducing the performance of the system. Moreover, this kind of services are hard to scale, since their only option is to vertically increase the quantity of resources available, in case of necessity. So, considering the online multiplayer scalability requirements of most recent video games, which connect up to even hundreds of players [47] to the same service, this limitation can be considered a significant hindrance;

3. even if many Game Engine aim to guarantee cross-platform functionalities, using their lowest layer to adapt to the specific hardware targeted for the deployment, there is still a significant degree of platform dependence. As such, technical problems may arise when trying to deploy across multiple different platforms, including: undocumented/proprietary APIs_G limitations, loss of performance due optimization for specific hardware, developers' hardware-related skill specialization [12].

Taking these elements into consideration, we can now understand the reasons behind the need for Game Engines that stray away from this legacy monolithic architecture.

2.2 Distributed Game Engines

*In this section we discuss the general **design proposals** in the context Distributed Game Engines, which are presented as a solution to the shortcomings of Legacy Game Engines. Nonetheless, Distributed Game Engines are not perfect, as such we also discuss possible **problems and limits** of such architecture.*

2.2.1 Proposed architectures

In term of Game Engines implementations, there are multiple approaches that have been considered, which leverage the characteristics of different network infrastructures. More specifically, game developers have three options [14]:

* *Client-server architecture*, which consists of a Game Engine running the most computationally intensive tasks on client-side, while the server manages the shared game state (e.g. position of players and non-player characters called "NPC"). This approach is widely implemented and mastered [14]. However, it does not provide a solution to the problems of Legacy Game Engines, as it is still mostly centralized and game state synchronization hinders scalability.

* *Cloud$_G$ gaming*, which consists of offloading all the modules that manage Game Engine functionalities to a remote server, while streaming back an encoded video to the client. This approach has been considered by multiple works on the topic [8, 13], but the impact of network latency is still a problem object of research. Moreover, this approach does not solve the problem of centralization when implementing Legacy Game Engines. In fact, difficulties in the management of virtualized resources can still cause performance degradation and resource congestion, even in these remote environments [14].

* *Computation offloading*, which consists of separating (decoupling) the Game Engine modules related to its main functionalities and offloading their execution to nearby servers. This approach has the potential to solve most of the shortcomings of Legacy Game Engines, as in this context it would be possible to also provision each server with the amount of resources needed to process their specific Game Engine functionalities. Nonetheless, considering the requirement of multiple interactions and data exchanges between distant modules, problems related to network latency and traffic can still arise.

Considering these three approaches, much work [3, 7, 13] has been conducted in order to study the possibility develop Game Engines with a non-monolithic structure, which we will call *Distributed Game Engines*. Overall, the most promising alternatives are *Cloud$_G$ gaming* and *Computation offloading*, since they are able to provide different solutions to some of the shortcomings of Legacy Game Engines.
We can now see some of the most interesting proposals that apply these approaches, while moving towards our project's idea of Distributed Game Engine.

A Distributed Game Engine for Mobile Games on the Android Platform [7]

As we mentioned before, many Legacy Game Engines are implemented following the classical client-server approach.
This work proposes an architecture with the same structure, but a different principle

at its basis: all functionalities should be present in every node and used seamlessly by the runtime component of the Game Engine. This is an interesting first approach to Game Engine distribution, which focuses on replicating Game Engine components and creating generic software nodes to build a distributed system.

In this architecture, client and server are both divided in modules dedicated to specific Game Engine functionalities.
The server, in particular, is responsible for: managing the connections with the clients, creating matches and allocating the required resources, updating the state game state. The client, on the other hand, focuses on functionalities that interact directly with the player (e.g. input management and graphics rendering), sharing only a specific subset of network-related modules with the server. The main purpose of this shared modules is communicating with the server, by sending client's requests, or instantiating offline games.

The communication inside this client-server architecture, consists mainly of game actions generated by the client and sent to the server for validation, all inside a Game Loop whose module is shared between the two components.

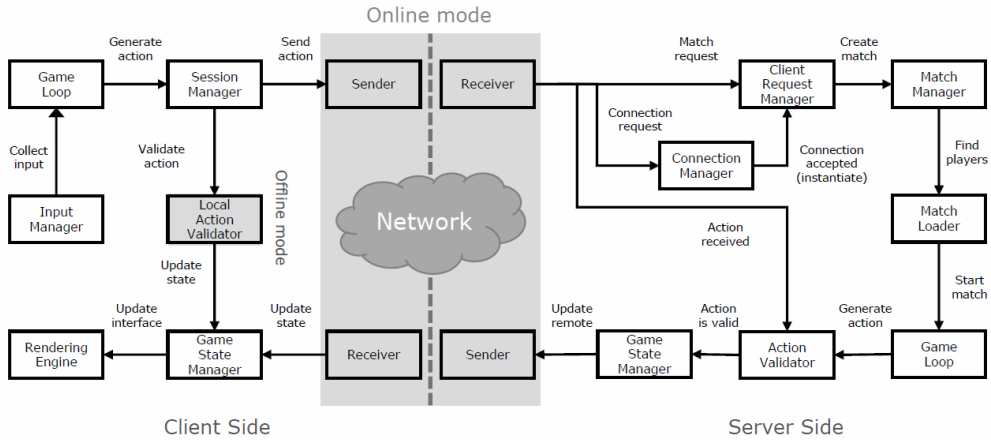


Figure 2.3: Client-server communication diagram.

Overall, this architecture provides an interesting evolution of the widespread client-server approach, typical of Legacy Game Engines, towards a more modularized and distributed alternative.

Integrating Game Engines into the Mobile Cloud$_G$ as Micro-services [10]

This proposal is set in the context of Cloud$_G$ computing, where the aim of the work is to divide an application (Game Engine) in multiple components and deploy them over a device Cloud$_G$.

In particular, the envisioned architecture should be able to encapsulate different business logics of a game or visualization application as modules, distributing them over several devices and exposing their function as micro-services. As such, we can see how this approach can also fall under the category of Computation offloading, since these decoupled modules are expected to communicate with each other and to be

flexible in terms of resource allocation, in order to provide singleplayer and multiplayer scalability.

Their modularization logic follows the Model-View-Controller (MVC) design pattern, where the Controller sends update requests to the Model, the Model updates data accordingly and then the View updates its rendering data by observing the Model. This pattern can be quite helpful in identifying how to categorize the Game Engine functionalities.

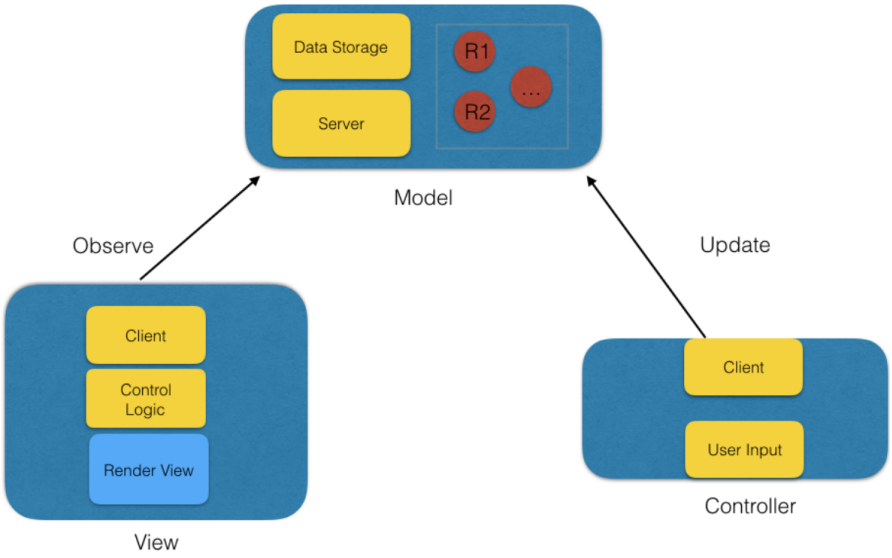


Figure 2.4: Overview of the distributed MVC architecture.

Overall, this work provides an interesting approach to the problem of identifying a reasonable logic with which to modularize the functionalities of a Game Engine, by taking the MVC pattern as reference and implementing the system with micro-services.

Still, distributing all Game Engine components into just three modules might be not enough in order to provide the flexibility and scalability we are looking for in a Distributed Game Engine. Moreover, the components inside these macro-modules would still be dependent on each other, taking us back to the same problems present the original monolithic architecture.

SMASH: a Distributed Game Engine Architecture [12]

In the context of Computation offloading, the SMASH architecture provides a general idea of the approach that can be used to design and develop a Distributed Game Engine.
More specifically, this work proposes an execution environment that takes inspiration from microkernel architectures and focuses on providing three basic functionalities:

* a *soft real-time scheduler*, in charge of timely calling scheduled functions of the modules and setting the pace of the system;
* a *dynamic game modules manager*, allowing for flexible implementation and change of the Game Engine modules;

* a *messaging system between modules*, implemented through a message bus, able to carry function calls and replies between the different modules of the distributed system.

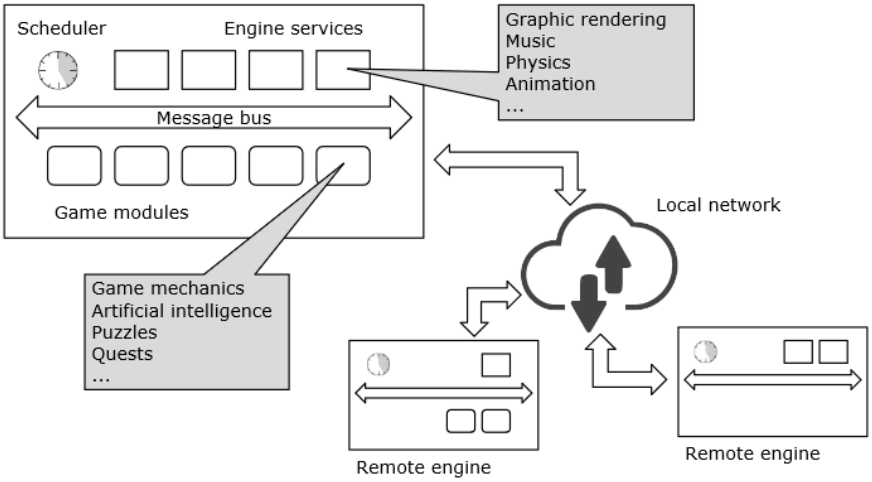


Figure 2.5: The SMASH system architecture.

The Game Engine modules in this work are defined as independent entities that provide gaming functionalities. In particular, this refers to components which are reusable between different games (e.g. graphics engine or physics engine), making the development of games much more efficient thanks to the flexibility here provided. The dynamic game modules manager allows, in fact, to swap in and out modules at runtime, thus giving the developers the possibility to extend and modify games in a much more agile manner, if compared with Legacy Game Engines that require full (or almost full) software refactoring in this context.
Moreover, the possibility to build, compile and debug these components as stand-alone entities also makes the development of games with this type of engine substantially easier.

Outside of the game development aspect, this modular design offers interesting benefits also in terms of runtime execution. In situations where the system is overloaded and more resources are required, this architecture is able to also scale horizontally, allowing the introduction of new engines or modules, as well as relocating existing ones on different machines. This way, the computational load can be distributed over more than one node.

Naturally, the highly distributed nature of this system requires intense communication between its various components, which is efficiently carried out through the message bus in local environments with no network latency involved. In their work, the decision to exclude network latency from the benchmarking scenarios is reasonable for evaluating the actual system performance. However, if we consider practical and realistic implementations of Distributed Game Engines, network effects such as latency and traffic should be taken into consideration, since they can introduce technical limitations also on the design level.

Still, this paper is an important inspiration for our work, setting the basis for development and research of new modularized Distributed Game Engines.

Distributed Cloud$_G$ Gaming Pipeline [13]

This work proposes a conceptual architecture for the modularization of a Legacy Game Engine in a Cloud$_G$ environment.
In order to test the architecture without a full Legacy Game Engine modularization, only a limited amount of functionalities was actually implemented. These include:

* the graphics rendering service;
* the encoding/streaming service;
* the state manager.

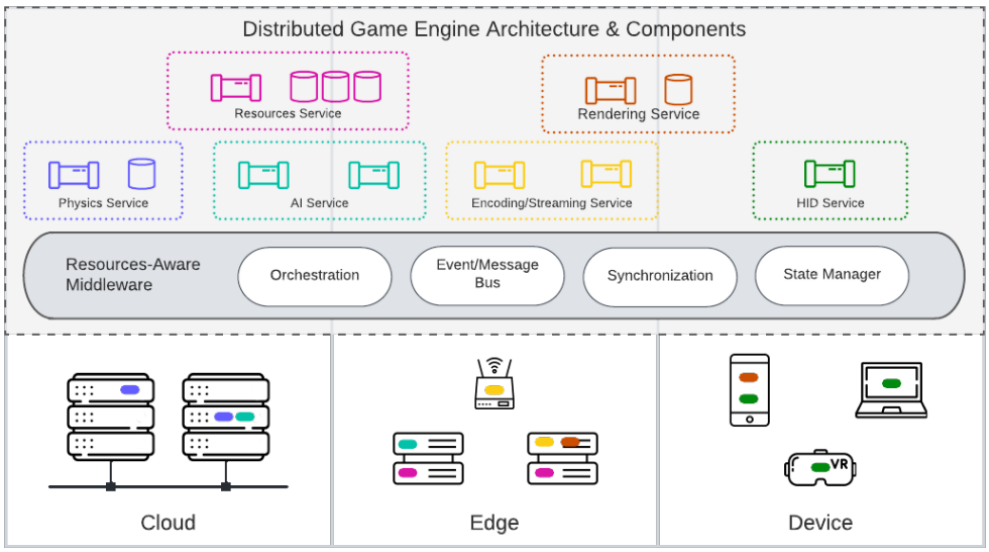


Figure 2.6: Distributed architecture leaveraging a resource-aware middleware support.

High priority was placed on the efficient management of virtual resources, implemented through a *resource-aware support middleware*. This middleware exploits various techniques to abstract all aspects related to low-level management of edge-cloud$_G$ resources, which are mainly composed of physical and virtual nodes with different computational, network and storage capabilities.
This is very relevant in the context of Game Engine modularization, since it ensures not only that the modules original functionalities are maintained, but also that the resources allocated for them actually meet their requirements.

On a more technical level, the modules are implemented through Docker containers, which are both lightweight and interactive virtual environments, that fit well with the requirements of these distributed components.

Overall this architecture proposes a very interesting approach to the modularization of Legacy Game Engines, especially with the introduction of the concept of

Containerization, which provides the characteristics needed for a practical approach to system distribution. However, even if some performance tests have been conducted on the final architecture, there is still room for additional evaluations on the impact of network-related phenomenon (e.g. network latency and traffic) on the performance of such distributed systems.

2.2.2 The impact of network latency on the User Experience

As we saw in the previous sections Distributed Game Engines, regardless of the approach, need to work inside a network environment. Whether we decide to implement the distributed modules on Cloud_G virtual machines or on a local network, there is still a network layer that needs to be introduced into the picture, in order to allow these modules to communicate with each other.

Often times researches do not include experiments related to the impact of network latency or traffic on the performance of their proposed architectures, as the focus is generally directed towards the actual performance of the system.
However, if we aim to provide a practical implementation of a Distributed Game Engine architecture, we cannot simply ignore the presence of these effects, which are present in realistic network environments. Moreover, the development of the distributed architecture itself should take these effects into account, when technical choices are made during the initial design.

There have been studies, in fact, on the impact of network latency and game responsiveness on the players performance [4, 5, 18], particularly in the context of Cloud_G-based games. The result is that, even with modest amounts of latency, the user performance can degrade up to 25% with each 100 milliseconds of network latency, with a directly proportional general perception of QoE_G. Moreover, comparing these results for Cloud_G-based games with traditional games, shows that they are as sensitive to latency as first person avatar games, which is the most latency-sensitive class of games.

Network latency, however, does not impact the User Experience only through image delay. Legacy Game Engines are generally described as highly performant [8], due to their monolithic nature that allows the different internal components to interact very efficiently between each other.

On the other hand, in the context of Distributed Game Engines, the communication between the various modules can be slowed down or become unstable, due to problems in the network infrastructure. Considering role of the Game Engine modules in managing core functionalities of the software, it is to be expected for delays in their communications to also impact the actual performance of the system. For instance, if the module responsible for managing the graphics rendering of the game image is impacted by network latency, the graphics framerate is likely to become object of degradation, thus indirectly affecting the User Experience.

Researches on the topic [4] have demonstrated how excessively low framerates have a very significant impact on the playability and QoE_G of the games. Moreover, gaming genres which require quick real-time reactions to the displayed image (e.g. FPS_G, racing games) are particularly sensible in this regard, thus the user tolerance is generally lower.

As such, we can finally assert that the network infrastructure used for modules communication is a paramount aspect to consider when designing a Distributed Game Engine.

2.2.3 Distributed systems' scalability

In terms of scalability, Distributed Game Engines provide meaningful improvements, by allowing systems to not only scale vertically the quantity of available resources, but also horizontally through the offload of computational tasks to additional nodes or replicas. Still, this improvement comes also with some limits and drawbacks to consider.

While Legacy Game Engine internal components are able to communicate directly with each other inside the same environment, this is different for Distributed Game Engine modules, which require an additional means to exchange data. This requirement is not necessarily a problem, since there are multiple ways to satisfy it, whether through socket-based communication between the modules or storage of data on shared memory.

Nonetheless, as studies have shown [15, 16, 19], the amount of remote calls that happen between the distributed modules is generally quite high and tends to generate a significant amount of workload for the means of communication. As the number of distributed modules increases, so does the amount of communication traffic introduced, since there are more components which require to have their remote calls carried out through the network.

In this context, there are obviously limits to the amount of traffic a communication means can handle, whether because of network bandwidth constraints or finite amount of resources available for the shared memory. Reaching such limits can cause system congestion, and thus generate problematic phenomena, such as packet loss or network latency, that can undermine the performance of the whole distributed system.

Considering this possibility, if we aim to effectively turn a Legacy Game Engine into a Distributed Game Engine, an evaluation needs to be conducted on which functions or components to actually offload into distributed modules [14]. The modularization of a needlessly large number of Game Engine functionalities could, in fact, pose us in this exact situation, whereas if too many of them are grouped inside the same modules, the benefits of a distributed approach could be significantly reduced.

To sum up, it is important to strike a reasonable balance between the amount of modules in a Distributed Game Engine and the capabilities of the means of communication used to support them, considering the scalability limits present also in this kind of system.

2.2.4 Synchronization of a shared game state

The management of the game state has always been a problem even in classic client-server architectures [7, 15]. In such architectures, there is a constant exchange of information about the global state and the next player action, between the local client and the server. This process is generally managed as follows:

1. the client receives a network message/packet, containing information about the

current global state, often including data related to the specific client's player situation;

2. the client computes the next action to be sent to the server, whether through Artificial Intelligence (AI_G) or through human input;
3. the client sends the action to the server as a network message/packet, and the server proceeds to verify its validity with respect to the current game state;
4. if the action is valid, the server computes the new global state and updates it, sending new information about it to the client;
5. the process then repeats until the game is finished.

While seemingly quite simple, this process becomes much harder to manage in the context of multiplayer gaming [7, 16], where there is more than one client interacting with the server and receiving game state updates. Moreover, in order to guarantee fairness to all players, it is important that they are all constantly provided with the same updated view of the game state.

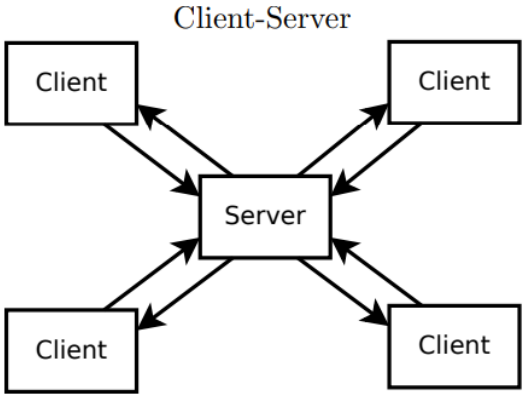


Figure 2.7: Simple view of a multi client-server architecture.

With a monolithic architecture it is reasonable to identify the server as a single compact component, able to efficiently manage and update the game state. However, in the context of Distributed Game Engines this may not necessarily be the case, since the server itself is a distributed system with multiple components that need to interact with the game state.

The distributed modules require the same updated view of game state data, in order to correctly implement the original Game Engine functionalities, as well the possibility to modify such data concurrently with the other modules. This, however, is not a trivial task, since the management of consistent data can be hard without a dedicated infrastructure and, even in that context, the synchronization process can be computationally expensive for wide distributed systems or large amounts of data. Moreover, the actual storage of the game state is usually responsibility of a single component, which may incur in bottlenecks if the number of read/write requests is excessively high.

For this reason, multiple researches [6, 7, 15] have studied how to make this process more efficient, with different approaches, such as:

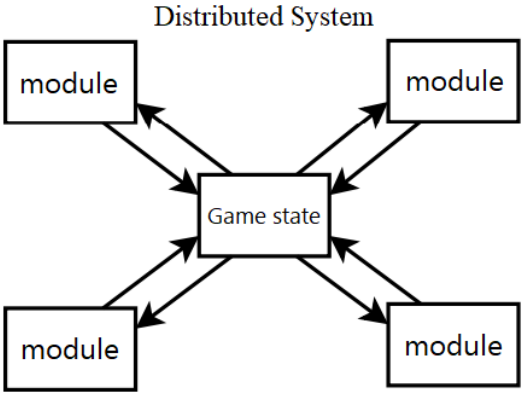


Figure 2.8: Simple view of a distributed system's state.

- usage of *local independent game state*, with changes computed locally, and a *remote game state* with which the local one is periodically synchronized [3]. This helps reducing the load on the game state management component, by reducing the frequency of updates;
- usage of *multiple game state management components* [2], with different characteristics, depending on the nature of the game state data to be stored. This could help distributing the traffic related to read/write requests among multiple components and providing the resources needed for processing different levels of workload;
- usage of *Area of Interest (AoI$_G$)* [3, 15], for deciding which game state data should be requested to the storage component and which should be managed locally (cached).

The approaches to this problem may vary greatly, also in relation to the characteristics of the architecture that is being developed.
Still, considering the general performance requirements of Game Engines, we can understand how important it is to carefully consider how the distributed modules communication and game state data management are implemented in such systems.

Chapter 3

Technologies

*This chapter discusses the technologies **used** or **considered** during the project, providing an in-depth description and motivating their choice.*

3.1 Docker

Docker is defined as a set of platform as a service (PaaS_G) products that use OS_G-level virtualization to package and run applications in loosely isolated environments called *containers* [23].
These containers can be seen as lightweight virtual machines, containing all the software needed to run applications. The flexibility of such environments makes it easy to run multiple simultaneous instances of them on the same host, allocating them a defined quantity of resources.

In general, the main features of Docker are [22]:

- *contained environments*: considering the isolation provided by Docker containers, its possible atomically perform changes to the software that is being hosted, without impacting any component outside of that environment;

- *responsive deployment and scaling*: Docker's lightweight nature makes it possible to dynamically manage workloads, scaling up or tearing down applications and services as needed, in real time;

- *running more workloads on the same hardware*: Docker containers require much less computational resources to manage, with respect to complete hypervisor-based virtual machines;

- *inter-container communication*: despite being isolated, it is possible for applications running inside Docker containers to interact with each other, if they are configured with the same Docker network.

If we compare these characteristics with the needs of Game Engine modules in a distributed architecture, it is easy to see them matching reasonably well with each other. Thus we can consider Docker as an interesting option for the implementation of Distributed Game Engine modules.

3.1.1 Architecture

Docker is designed with a client-server architecture [25], composed of:

* *Docker client*: which talks to a daemon and it's one of the primary ways to interact with Docker;
* *Docker daemon*: which performs the operations related to building, running and distributing Docker containers.

Both these components can be run on the same system or remotely, communicating using REST API$_G$ and using UNIX sockets or a dedicated network interface.

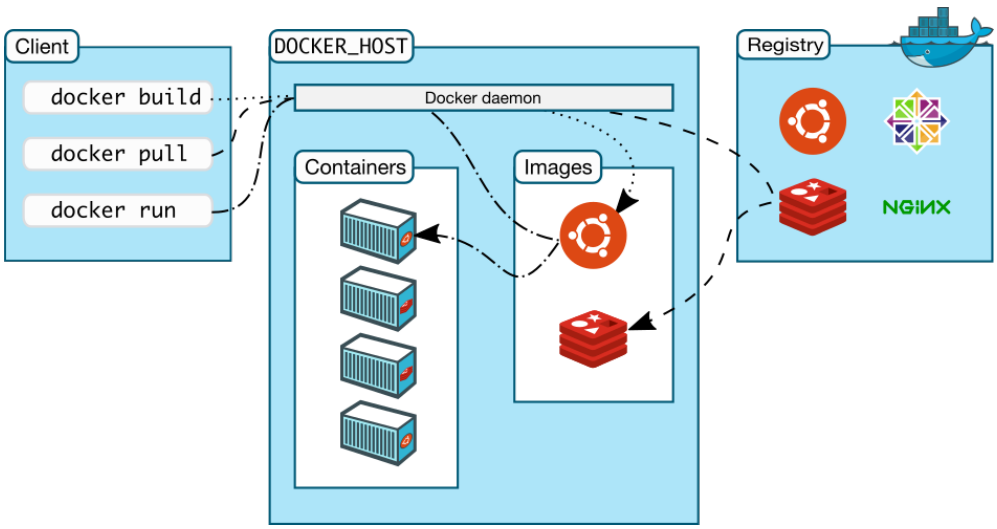


Figure 3.1: Representation of the Docker architecture.

3.1.2 Docker-compose

Compose is a tool used for defining and running applications which require multiple Docker containers, in a simplified manner [24]. Using Compose, it's possible to provide a single YAML$_G$ file to configure the services needed by the application and then, with a single command, create and start all these containerized services.
Some of the main features provided by this tool are:

* *multiple isolated environments on a single host*, which, even if possible without this tool, is made much easier and configurable;
* *preservation of volume data*, useful for applications that needs to be restated multiple times;
* *only recreates containers that have changed*, removing unnecessary build time from the application development process;
* *support for environment variables*, very important for correctly configuring software which interacts with the hardware (e.g. X11).

Overall, this tool can prove quite helpful when developing distributed systems using Docker.

3.1.3 Nvidia-docker

The NVIDIA Container Toolkit allows users to build and run GPU accelerated Docker containers. The toolkit comprises both a container runtime library and utilities to automatically configure containers to leverage NVIDIA GPUs [48].

The main components [49] include:

* the `nvidia-docker` wrapper;
* the *NVIDIA Container Runtime*, which mostly includes NVIDIA specific code and additional hooks;
* the *NVIDIA Container Runtime Hook*, which includes a script that implements the interface required for the start-up of the container and settings of configurations;
* the *NVIDIA Container Library and CLI*$_G$, providing a library and a simple CLI$_G$ utility to configure GNU/Linux containers leveraging NVIDIA GPUS.

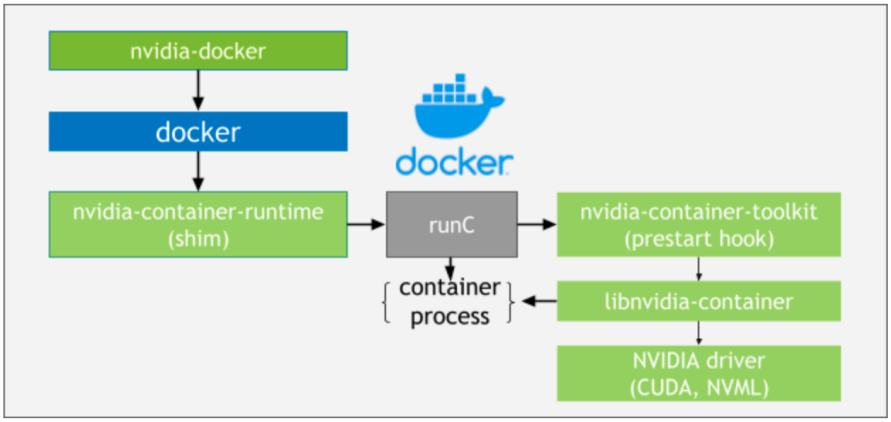


Figure 3.2: Nvidia-docker architecture.

This toolkit can be an interesting addition for implementing Game Engine graphics modules. However, it is currently not compatible with Docker-compose, which is also an important tool for building distributed system applications.

3.2 ETCD

ETCD is described as a disk-based, distributed and reliable Key-Value Store, which allows the storage of data that can be accessed by a distributed system [39]. More specifically, ETCD is designed to reliably store infrequently updated data, exposing previous versions of key-value pairs to support inexpensive snapshots of the DB$_G$ and watch history events [27].

In order to provide these features, ETCD presents a persistent, multi-version, concurrency-control data model for its key-value storage. This component preserves the previous version of any key-value pair when its value is overwritten with new data. As such,

this storage system is described as immutable, since its operations do ot update the structure in-place, but instead generate a new updated structure, where all past versions of the keys are accessible and watchable after modification.

In order to guarantee the storage of strongly consistent data, ETCD uses a leader-based consensus protocol called *Raft* for data replication and log execution. The implementation of such algorithm, as well as the rest of ETCD architecture, is developed in the Go programming language, which is particularly fitting for distributed system environments.

3.2.1 The Raft Algorithm

The Raft algorithm is typically adopted in the context of *state machine replication (SMR)* [60], which is a general method for implementing a fault-tolerant service. This is done by replicating servers (or nodes) and coordinating client interactions with the server replicas.
The state machine can be defined as a combination of:

* a set of *states*;
* a set of *inputs*;
* a set of *outputs*;
* a transition function, described as: $input \times state \rightarrow state$;
* an output function, described as: $input \times state \rightarrow output$;
* a specific *state* called *start*;

The maximum number of machine replicas that can fail, while still keeping the system operating, is defined by the *crash-fault tolerance*. Moreover, the state machine is required to be *deterministic*, which means that its replicas, starting from the same state and receiving the same input, should return the same output.

In this context, a consensus algorithm is required in order to determine and preserve the only authoritative version of the command execution history, in a strongly consistent log.

Algorithm overview

The main goal of the Raft algorithm is to allow distributed nodes to achieve *consensus* on the system state. This means that multiple servers or nodes must agree on some data value that is needed during computation [20, 54]. This is, generally, a primary objective in the design of fault-tolerant distributed systems.

The Raft algorithms defines three main roles, which may be assumed by any node in certain specific situations:

* *Follower*: which is a simple node that composes the replicated state machine. This is the default role assigned to nodes when they become part of the *server cluster*, which we define as the subset of nodes that implements the Raft algorithm in a distributed system. Follower nodes may only respond to Candidate and Leader nodes;

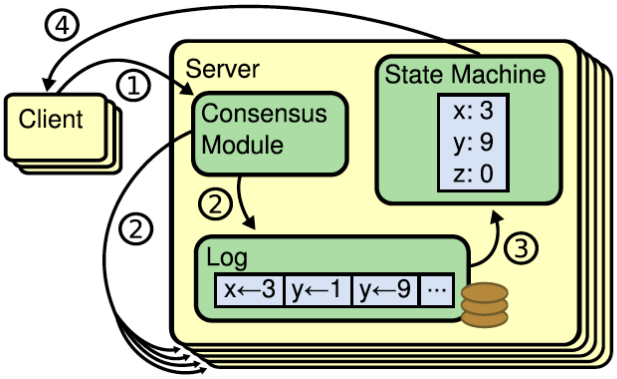


Figure 3.3: Replicated state machine interactions.

* *Candidate*: which is a node competing with other Candidate nodes to assume the role of Leader of the distributed system. Nodes may assume this role to request a Leader election, in a situation in which the current Leader node is unresponsive;
* *Leader*: which is unique and the only node that interacts with the client. Any request made to Follower nodes is redirected to the Leader node.

As previously mentioned, these roles are very important for Raft to be implemented for asymmetric multi-server distributed systems.

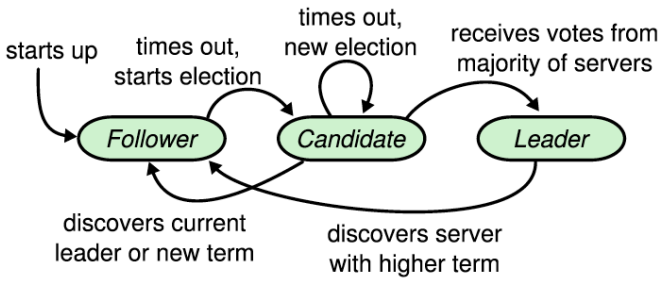


Figure 3.4: Representation of the Raft roles.

Time management

In order to correctly handle server states, the Raft algorithm manages the time by dividing it in *terms* [54].
Working in an asynchronous environment, these terms are not identified by a fixed real-time length. Different servers may observe the transition between terms at different times, while the most significant element of reference is actually the local state of each server. As such, the terms are meant to act as a logical clock, allowing servers to detect obsolete information and stale Leaders, regardless of the asynchronous nature of the system.

Each term is uniquely identified by a monotonically increasing integer number called *term number*. This number is stored in each node and attached in node communications.

Each term starts with a Leader election, where the Candidates request votes to other server nodes (Followers) in order to obtain a majority consensus:

* if a Candidate node manages to obtain the majority of the votes, that Candidate becomes the Leader for the current term;
* if no Candidate node manages to obtain the majority of the votes, this situation is called *split vote* and the term will conclude with no elected Leader.

As such, a term can only have one single Leader at any time. Moreover, the term number is also a very important indicator for various system tasks, such as:

* *term number update*: if a node term number is lower than the one of the other nodes in the cluster, it will be updated at the beginning of a new term. The term numbers used for reference are the ones of the Candidates and the one of the Leader, which usually takes the priority;
* *role demotion*: if a Candidate or Leader term number is lower than the other nodes in the cluster, they are demoted to Followers;
* *node communication*: the term number of a node is sent as an attachment to each request they make to other nodes. If a request is made with an outdated term number, it is always discarded.

This makes the term number a crucial factor in time management, for the Raft algorithm.

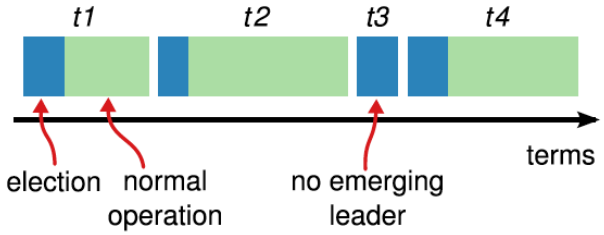


Figure 3.5: Representation of terms ($t\#$).

Leader election

In order to maintain the authority as Leader of the cluster, the node sends periodic messages to the other Follower nodes. These communications are called *heartbeats* [54].

When a Follower node doesn't receive a heartbeat within a given time bound, it assumes the Leader is not active anymore and promotes itself to the role of Candidate. It then votes for itself and requests other nodes to vote for itself, in order to establish a majority. The result of the election can be one of:

* the Candidate node receives a vote from the majority of the nodes of the cluster, making it the new Leader. In that moment, the node begins to send heartbeats to notify other nodes of the presence of a new Leader. Candidate nodes receiving the heartbeat will return to the Follower role;

- the Candidate node does not manage to get votes from the majority of the nodes of the cluster. In this case, the election ends with no Leader and the node returns to the Follower role;
- if the term number of the Candidate node who requested the votes is lower than other Candidate nodes in the cluster, the request is immediately refused and the other nodes keep their Candidate role.

This whole process is performed multiple times during the Raft algorithm lifecycle in distributed systems. It ensures that a Leader is always present to serve clients requests and that is condition to do so.

Log replication

Each request made by a client is stored by the Leader in the *logs*, which are then replicated in the other Follower nodes [54].
Generally, a log entry contains the following information:

- *command*: which is the instruction specified by the user to be executed;
- *index*: which is the number used to identify the position of the entry in the node log, starting from 1;
- *term number*: which is used to ensure the time of a command specific entry.

The Leader node requests, through a broadcast, that all the Follower nodes synchronize their logs with its own. The Followers shall reply with an acknowledgment, to confirm the completion of such operation, as the Leader will continue to broadcast synchronization requests until all the other nodes have performed it.

A new entry, in this context, is considered *committed* when the majority of the nodes in the cluster has successfully copied it in their logs. At that point, the Leader itself confirms the addition of the entry to its own log, representing the successful outcome of the replication. Following this procedure, it is possible to guarantee that all the previous entry of the log are also committed, otherwise they would not be present inside the log.
After an entry has been committed, the Leader carries out the request present in the entry and responds to the client with its result. In this way, the entries are always executed in the same order they are received and confirmed. If two entries in different logs have the same index and term number, it is guaranteed that they contain the same command and that the logs are identical until the specified index.

In a situation in which the Leader node crashes, the logs may become inconsistent, since it is the entity responsible for fixing conflicts in the Follower nodes. In this case, the protocol ensures a new Leader is elected, which will look for the last coherent Index in the Followers logs and then overwrite every entry beyond that specific Index with its own. This whole process ensures that the logs of the Followers always match with the Leader and guarantees that the system can provide *Strong Consistency* for the stored data.

Safety

In order to maintain *Strong Consistency* of data on a set of server nodes, the Raft algorithm ensures that the Leader is storing all the entries related to previous terms,

committed inside its own log.
During Leader election, the request for a vote that a Candidate sends to other nodes, also contains some information regarding the log of the Candidate. This information includes the term number of said node, which can be evaluated by the receiver, to immediately discard requests from not updated nodes.

In addition to this rule, there are also several other design choices, which help avoid breakage of consensus [54]:

* *leader election safety*: there shall be at most one Leader for each term;
* *log matching safety*: if multiple logs contains an entry with the same Index and Term, then those logs are guaranteed to be identical, till the given index;
* *leader completeness*: the log entries which are committed in a given term shall always appear in the log of the Leader;
* *state machine safety*: if a server applied a particular log entry to its state machine, then no other node in the cluster shall apply a different command in its own log;
* *leader is append-only*: a Leader node can only add commands at the end of its log. No other data altering operation is allowed;
* *follower node crash*: when a Follower node crashes, all the requests sent to the crashed node are ignored. Moreover, the crashed node cannot take part to any Leader election. When the node is restarted, it shall synchronize with the Leader.

These characteristics of the Raft algorithm are considered to be sufficient, in order to ensure correct management of operations. Still, the implementation of Strong Consistency is generally expensive in term of computational time, since it requires the system to await for the majority of nodes to synchronize with the Leader, before acknowledging the completion of write operations.
As such, it is possible that the performance of this system may not be ideal in contexts where the stored data is changed frequently.

3.2.2 Interaction with ETCD

ETCD allows direct interactions from the client through specific requests, which generally include functionalities [29] such as:

* *write key*: where the client, communicating with any node of the ETCD cluster, instruct the distributed system to write a new key-value or modify an existing key-value. Following the Raft algorithm, if this type of request is sent to a Follower node, it is redirected to the Leader for the actual processing;
* *read key*: where the client, communicating with any node of the ETCD cluster, asks for the value of a specific key. According to ETCD configurations, this type of request can be carried out by any ETCD cluster member, since they are constantly updated on the current state of the stored data [30].

On a more practical level, these requests can be sent either using a dedicated ETCD CLI$_G$ or through ETCD libraries/APIs$_G$ developed in order to allow for interaction also within applications written in different programming languages. In our work, which focuses on code written in C++, the library of choice was `etcd-cpp-apiv3` [26],

which is considered to be one of the most updated and reliable ones.

Other functionalities [29] provided by ETCD CLI_G and $APIs_G$ are:

* *delete key*: which requests the delete of a specific key-value from the ETCD database;
* *watch key*: which is an interesting functionality, that allows clients to subscribe to value updates of specific keys, in order to be notified in real time. This feature is can be particularly helpful when implementing communication through shared memory, since it removes the need for clients to continuously poll the data, in order to verify the presence of changes;
* *transactions*: used to perform certain operations when one set of conditions is met or perform other operations when the conditions are not met;
* *leases*: mechanism generally used to detect whether a node is alive in the distributed system.

All these remote requests and functionalities, which allow the client to communicate with the ETCD system, are carried out using the $HTTP_G$ protocol. This allows ETCD to leverage all the features provided by such protocol, including *pipelining*: feature that allows the transmission of multiple requests in a single TCP_G packet, possibly improving the performance of the system. [45]

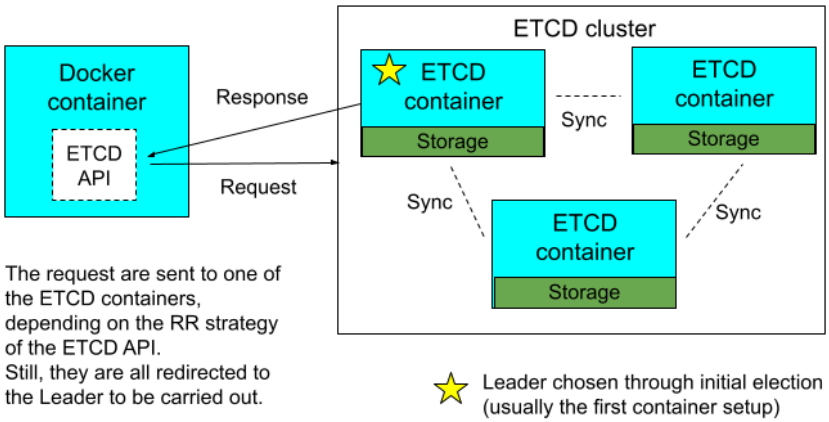


Figure 3.6: Example of ETCD cluster with Docker interaction.

3.2.3 Docker compatibility

Considering the distributed nature of the ETCD system, it is possible to actually implement ETCD nodes as Docker containers. They can be run both in standalone mode and in cluster mode [28] with other ETCD containers, communicating using the Docker networking features.
Moreover, as shown in Figure 5.5, different applications running on other Docker containers can connect and interact with the containerized ETCD system, using language-specific $APIs_G$ or, if installed on the app container, directly the ETCD CLI_G.

3.2.4 System maintenance

ETCD clusters and nodes require periodic maintenance to remain reliable. Depending on the needs of the ETCD application, the maintenance can be automated and performed without downtime or significantly degraded performance [31].

The ETCD system, in fact, keeps an exact history of its keyspace, which should be periodically compacted in order to avoid performance degradation and eventual storage space exhaustion. On a more technical level, compacting the keyspace history drops all information about keys superseded prior to a given keyspace revision (version of the key). This operation can be configured to be performed periodically or after a set number of key revisions.

This freed space should then become available for additional writes to the keyspace. However, in order to completely free the computational resource of the host, an additional operation called *defragmentation* is required. This operation releases the storage space still being consumed by the internal fragmentation of the backend database, and needs to be issued manually for each ETCD member, so that cluster-wide latency spikes may be avoided.

Even if the compaction operation can be performed automatically, maintaining the ETCD system using these tools can be quite expensive, both in computational terms and in practical terms, since the defragmentation can only to be launched manually when needed.

3.3 Redis

Redis is an open source, in-memory data structure store used as database, cache, message broker and streaming engine [58]. Redis provides support for multiple different data structures, such as:

* strings;
* hashes;
* lists;
* sets;
* sorted sets with range queries;
* bitmaps;
* hyperloglogs;
* geospatial indexes;
* streams;

which can all be managed inside the key-value storage provided by the system. It is also possible to run atomic operations on these types (e.g. appending to a string, incrementing the value of an hash, push an element to a list), dedicated to the specific characteristics of the targeted data structure.

Differently from ETCD, which works mainly by operating on the disk, Redis works with an in-memory dataset, which can optionally be periodically dumped on to the disk in the form of snapshots.
The software is written in ANSI C and works with most POSIX systems, without external dependences.

3.3.1 Features and configurations

This system provides many interesting and, sometimes, unique features that can prove useful when managing a distributed system [58]. These include:

* *LUA scripting*: used for writing scripts that interact with the keys stored in the database;

* *eviction policies*: for managing which data should be deleted in case a set memory limit is reached, in term of resource consumption. These policies usually take into consideration the frequency with which a value is accessed (e.g. Least Frequently Used - LFU) or the last time the value was accessed (e.g. Least Recently Used - LRU);

* *transactions*: used for executing multiple commands at the same time, in a serialized manner, or in order to place condition on the execution of specific requests;

* *on-disk persistence*: with different levels, which allow the system to work directly in-memory or to periodically snapshot the database on the disk;

* *creation of distributed systems*: which are highly available thanks to the Eventual Consistency nature of the Redis system;

* *publish/subscribe paradigm*: based on events triggered by multiple type of occurrences, both related to messages (e.g. received, sent) and key values (e.g. changes, deletion, addition);

* *client-side caching*: this functionality be configured client-side, in order to decide which information to query from the Redis database and which to manage locally through a cache. This decision is typically tied to the frequency of changes that specific key is subject to (measured through the publish-subscribe mechanism);

* *pipelining*: implemented for managing multiple read and write requests with a reduced amount of read/write sys calls to the socket methods. This is particularly useful when dealing with multiple clients performing a large number of requests in the same time interval.

Additionally, if required by the context of application, it is also possible to configure *Redis itself as a cache*, making all the stored keys "live" only for a certain period of time, before being deleted.

3.3.2 Replication and synchronization mechanism

At the base of Redis replication there is a Leader-Follower (Master-Follower) consensus mechanism [59]. Similarly to the Raft algorithm, it allows replica Redis instances to be exact copies of Master instances.
Moreover, the replica automatically reconnect to the Master every time the link breaks, and attempt to be an exact copy of it regardless of what happens to the master. On a general level, the Redis mechanism works very similarly to the Raft algorithm implemented in ETCD, with the same roles and principles at its basis (e.g. Leader election, log replication).

There are, however, very important differences to consider, which highlight the different approaches of the two systems to data consistency.
First of all, Redis replicas (Followers) do not accept write requests in their default configuration, differently from ETCD where these requests are redirected by the Followers to the Leader. This is done in order to prevent multiple members to expose view of the distributed DB_G which is different from the one of the Leader.

The second difference is related to the management of members disconnections. In ETCD full snapshots of the database are periodically stored on the disk by the single members, and in case of disconnection they always request full synchronization of the last snapshot to the Leader. In Redis, on the other hand, snapshots of the DB_G are not stored on the disk, in the default non-persistence configuration. As such, when disconnections happen the member requests just partial synchronization with the current log of the Master, only for the commands lost during the downtime. Full synchronization requires, in fact, the creation of a new complete snapshot of the Master's DB, which is a computationally expensive operation. As such, this process is performed only in cases when partial synchronization is not possible.

The third, and most important, difference is related to the management of write requests. In ETCD, as mentioned in section 3.2.1, the Leader waits for the write operation to be *commited*, before sending the positive result to the client which requested it. This is an important aspect, which characterizes a system with Strong Consistency of data as a priority.
However, in Redis, this is considered to be time-inefficient. As such, when managing write requests, Redis immediately sends a positive result to the Client, as soon as the writing has been completed on the Master's log, without waiting for the replicas synchronization.
This might become problematic in situations where the Master node crashes before the synchronization with the replicas is complete, since the Followers will not have inside their logs the write requests that has been declared completed to the Client. As such, this system cannot be defined as Strongly Consistent. However, considering that the replicas will still, in the end, contain the same data inside their databases, this system provides Eventual Consistency of the stored data.

In general, we can assert that the main difference between the ETCD and Redis systems management of data is in their approach to consistency. ETCD focuses heavily on Strong Consistency, even at the expense of system performance. Whereas, Redis focuses on providing good performance in standard situations, while accepting the possibility of temporarily losing consistency in some edge cases (Eventual Consistency).

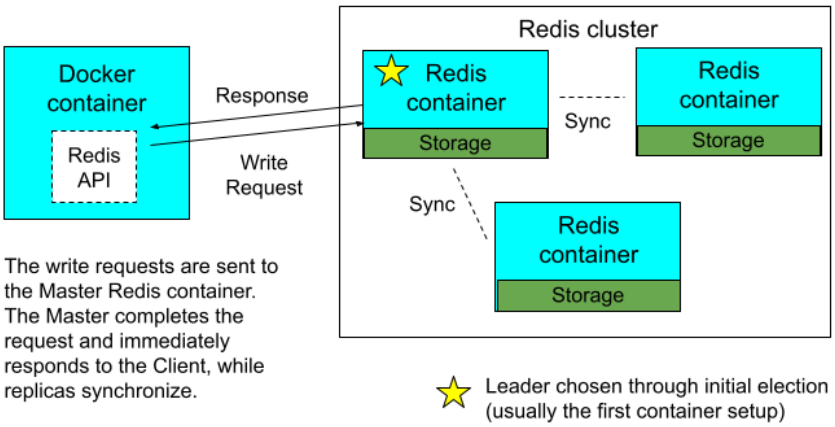


Figure 3.7: Representation of Redis cluster replication with Docker.

3.3.3 Interaction with Redis

Similarly to ETCD, Redis also provides CLI_G commands (`redis-cli`) that can be used to interact with the Redis instance from command line.
Moreover, regarding the possibility to interact with the Redis instance from applications written in specific programming languages (e.g. C++), Redis provides a list of APIs_G that can be used for this purpose [56].
In particular, for our project, the most fitting option is the `redis-plus-plus` C++ API_G [55], which is fairly more complex than the previously mentioned ETCD alternative, but provides a large amount of interesting functionalities.

In term of key-value interactions, the features provided include all the ones mentioned in section 3.2.2 for ETCD, with the addition of:

* *pipelined requests*: to be configured directly in-code, by inserting requests into the pipeline structure and then call a dedicated method for their execution;
* *publish-subscribe mechanism*: which is socket-based and allows the notification of clients through specific messages.

3.3.4 Docker compatibility

Considering the distributed nature of the Redis system, it is possible to actually implement Redis nodes as Docker containers. They can be run both in standalone mode and in cluster mode [57] with other Redis containers, communicating using the Docker networking features.
Moreover, as we can see in Figure 5.6, different applications running on other Docker containers can connect and interact with the containerized Redis system, using language-specific APIs_G or, if installed on the app container, directly the Redis CLI_G.

Chapter 4

Feasibility study

This chapter discusses the scope of the thesis, with a feasibility study conducted on multiple softwares and introducing the main subject of experimentation, which is the ***TORCS game engine****.*

4.1 Preliminary analysis

In order to study the possibility to modularize and distribute a piece of videogame-related software, we need to choose the main subject of research. More specifically, we require a piece of software which is:

* *medium sized*, in terms of quantity of code present in its codebase. Software that is too simple can, in fact, limit the amount of features that can be turned into distributed modules. Whereas very complex software could require an excessive amount of effort in order to decouple intertwined functionalities from the main architecture;

* *composite*, with a clear distinction between the various parts that make up its codebase. This can be particularly helpful when identifying the functionalities to modularize for the distributed alternative;

* *well documented*, particularly in relation to the software architecture. A clear and complete documentation can, in fact, make the process of modularization much more efficient, allowing us to rapidly identify the software components and their interactions.

In addition to these three main criteria, we also try to focus on software which is *not excessively demanding* in terms of resources, since the local hardware limits would be more likely influence the experimental results in that case.
We can now elaborate more on the individual software we considered in our preliminary analysis, in the following sections.

4.1.1 Godot

Godot is a cross-platform Game Engine for the development of 2D and 3D games [42]. It provides a comprehensive set of tools, which allow the development of video games that can be easily exported to multiple platforms.

This characteristic can prove to be quite interesting for research purposes, as it enables multiple testing scenarios for applications, with different configurations [43]. Moreover, this software is described as semi-portable, allowing its executables to be run from any location and never requiring advanced system privileges.

This development software supports multiple programming languages, such as:

* *GDScript*, which is the recommended Godot original programming language;
* *Visual Scripting*;
* *C#* and *C++*;
* *Python*, which is considered to be in an experimental integration phase.

These cover most programming languages considered for our research project.

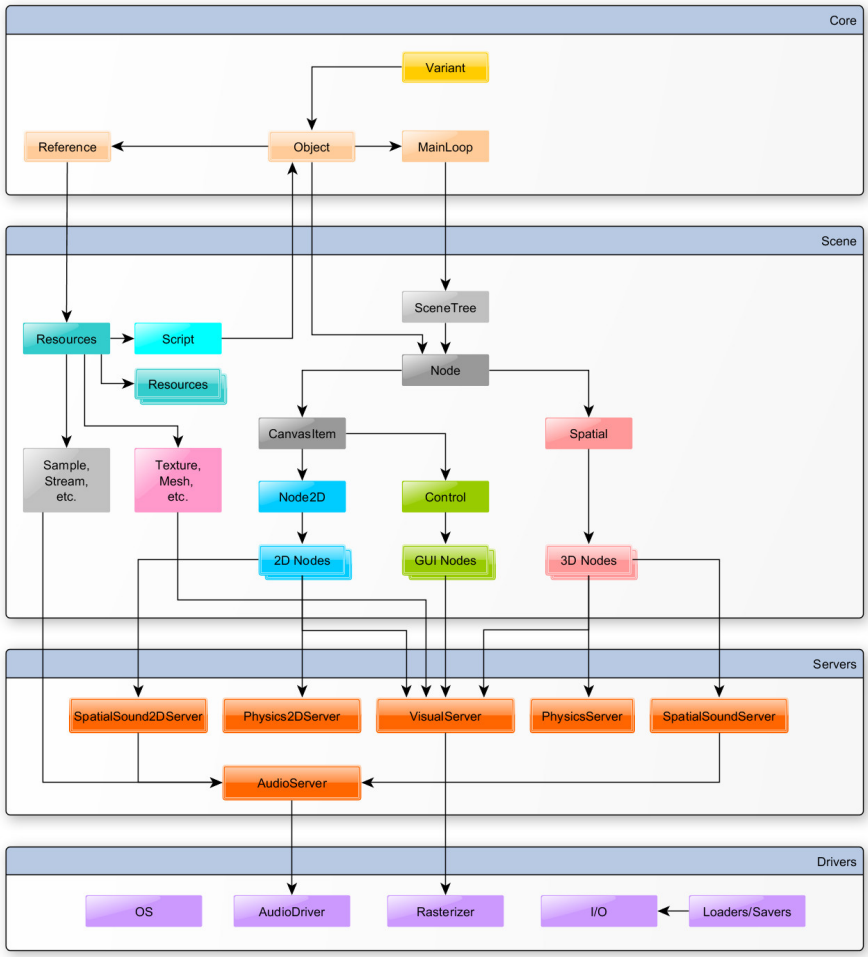


Figure 4.1: Representation of the Godot architecture.

The engine itself is reasonably well documented [42], with also a complete graphical

representation of its whole architecture [40], which we can see in Figure 4.1. However, this architecture is still quite complex, with multiple layers and many modules very integrated with each other. The core components, despite being quite small-sized, are not necessarily a good fit for modularization purposes, as they are the less flexible part of the structure.

Nonetheless, Godot offers flexibility in the form of custom modules [41], which can be developed in C++ and used to add new functionalities to the original Game Engine. They can be both small-sized or large-sized (e.g. libraries) and used both for video games development and execution.

Finally, another feature provided by Godot is the implementation of multi-threading as servers [44]. These structures are daemons which manage data, process it and push the elaborated result to other modules. As such, they can become an interesting component for the purpose of computation offloading in a distributed environment.

4.1.2 Python Minecraft Clone

In the following GitHub repository: https://github.com/obiwac/python-minecraft-clone we are provided with the source code of a Python clone of the video game Minecraft.
Minecraft is a sandbox game, originally developed in the Java programming language, where the players explore a blocky, procedurally generated 3D world, where they may discover and extract raw minerals, craft tools and items, and build structures [46].

Figure 4.2: In-game screenshot of Minecraft.

Despite not being a Game Engine, this piece of software was deemed to be interesting for our purposes, considering the simplicity of its structure, which would make the identification of the core functionalities relatively easy [52]. However, as discussed in the analysis criteria, the limited amount of features currently implemented could reduce the scope of a possible modularization, especially considering that the project is still in development.

In terms of documentation, this piece of software is also quite lacking, with only few comments inside the actual code and no external textual reference. Still, the

related video tutorial series [21], referenced in the repository, is well structured and clearly explains the development of all the software features and components.

4.1.3 Flightgear

Flightgear is an open-source flight simulator written mostly in C++ [36]. This software is quite complex in terms of structure, including a large number of subsystems handling tasks that range from simple features to more complex ones (e.g. multiplayer).

Figure 4.3: In-game screenshot of Flightgear.

While this complexity aspect is certainly problematic for the modularization of its tasks, the software codebase is managed following programming best practices and with clear distinction between the individual functionalities [34].
Moreover, the software is extensively documented, with a rich and well maintained Wiki that defines in clear terms what are its main features and tasks [33].

Being a "free-exploration" game, there are tasks that are computationally hard to manage and which could benefit from modularization. However, the software itself is quite demanding in terms of computational resources [35]. This may cause performance issues that can introduce noise into measurements and evaluations, if the local hardware is not able to satisfy the software requirements.

4.1.4 TORCS - The Open Racing Car Simulator

TORCS is a 3D racing simulator designed to allow AI_G drivers and humans to compete with each other, with the aim of studying autonomous driving [61].

This software includes both a video game and a Game Engine, with functionalities which are clearly separated inside the codebase [62]. This separation is helpful for a possible modularization of the system, more so if we consider that some of its components are explicitly loaded during execution (plugin), essentially defining the runtime component of the Game Engine (e.g. Rendering, Simulation, Track, Robot).

The codebase of this software is medium-sized, with enough functionalities to modularize, but also not excessively complex if the intention is to act on all of its components.

Figure 4.4: In-game screenshot of TORCS.

Additionally, the system requirements for running simulated races are not high, which prevents performance problems related to insufficient resources and the negative effects that would ensue.

The main problem of this software is the lack of maintained documentation, as the provided one is quite outdated and sparse between multiple sources.

4.1.5 FoFiX

FoFiX is a highly customizable rhythm game supporting many modes of guitar, bass, drum, and vocal gameplay for up to four players. It is the continuation of a long succession of modifications to the original Frets on Fire by Unreal Voodoo [37].

Figure 4.5: In-game screenshot of FoFiX.

The codebase itself of this game is medium-sized and written exclusively in the Python programming language. The code contains multiple small tasks, related to specific functionalities that could be benefit from modularization. However, the software was not developed with clear separation between the various functionalities, which are instead quite sparse and disorganized, thus hindering possible modularization operations.

There is also little to no real documentation related to the structure of the software, which could be problematic for understanding its features, also considering that the code is quite complex and largely uncommented.

Finally, even if the system requirements for this game are quite low, we need to consider that generally managing audio-based software in a network environment is not a trivial task.

4.1.6 Conclusion and Final decision

We now provide a summarization of each software evaluation, in the following table:

Table 4.1: Summary of the analysis.

Proposal	Evaluation
Godot	This Game Engine provides a modular structure and interesting features (Godot modules and servers). However, it is unclear if the main components actually fits for the purpose of modularization.
Python Minecraft Clone	The simple structure of the game can be interesting for allowing an easier modularization process. However, in this context, it might be excessive and undermine the purpose of the modularization.
Flightgear	The structure of the code provides many modularization cues, however it exceeds in complexity and resource requirements.
TORCS	The project exposes medium hardware requirements and complexity, while providing a modular codebase structure. The documentation clearly defines the modules actually loaded at runtime, providing an interesting starting point for a further modularization process.
FoFiX	The game is entirely developed in Python, with low requirements and medium size. However the timed audio-based nature of the game, along with the lack of documentation makes it a less interesting option.

At the end of this analysis, considering the evaluation criteria and the features of software option, the final choice was on the *TORCS Open Racing Car Simulator*.

4.2 Technical analysis

After TORCS was chosen, as output of the Preliminary analysis, a more in-depth and technical analysis was performed on the actual components of the software. More

specifically, we aimed to understand: the role of the various TORCS modules, the general Game Engine logic and how the game state is managed inside the system.

4.2.1 Architecture

The TORCS architecture is composed of three main layers:

* *orchestration layer*: which contains the logic used to call the methods defined in the TORCS API_G and libraries, for general functionalities such as main menu management or game engine initialization;
* *API_G layer*: which contains the actual implementation of the TORCS functionalities, separated in different libraries with specific purposes;
* *plugin layer*: which contains the Game Engine modules, related to general functionalities that are loaded and used at runtime (e.g. Rendering, Simulation, Track, Robot).

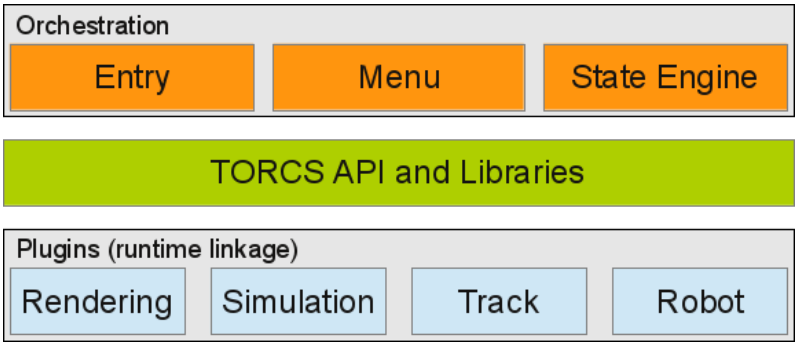


Figure 4.6: Representation of the TORCS architecture.

As we can see in the simplified representation of the architecture, on Figure 4.6, these layers are made of multiple components, which we will now elaborate on in the following sections.

Orchestration layer

As previously mentioned, this layer provides the programming logic for controlling the major program flow using different components, including:

* a *Entry Stage*, which is the first phase of TORCS start-up operations. In this process, the command line is analysed and, depending on the desired operation mode, TORCS is run either: in command-line mode, with no graphic and real time, or with the graphical menu;
* a *Menu* and its components, which are responsible for offering a visual user interface to ease the setup of TORCS. All the settings selected during its usage are persisted in XML_G files and are read later by the respective components. The components also include a *Race Menu*, which is a special menu that allows the creation of custom racing sessions;

* a *State Engine*, which controls the execution of the race specific configurations, including setup, run and shutdown of the simulation. Changes in the TORCS game state can be triggered by data inspection or function calls returned values, allowing the implementation of functionalities which interact with the simulation itself.

API$_G$ layer

This layer defines some interfaces and libraries which are used in multiple parts of the project, such as:

* XML$_G$ parameter file handling;
* common functions for robots;
* features related to portability;
* configurations for game screens;
* music player functionalities.

These include also tools for AI$_G$ development and creation of *adaptive agents*.

Plugin layer

This layer provides multiple plugins which all have specified interfaces and are loaded based on the configurations during runtime. These include Game Engine core functionalities and modules, such as:

* *Rendering*: responsible for rendering the current game situation. The default implementation performs visual and audio 3D rendering, based on OpenGL 1.3 and OpenAL APIs$_G$;
* *Simulation*: responsible for progressing the situation by a given time step;
* *Track*: responsible for loading tracks into the TORCS circuit management structure;
* *Robot*: responsible for driving cars in the simulation. TORCS can load multiple robots at the same time to drive multiple cars, with one robot supporting up to 10 cars at once. This module can also provide a "Human Driver" (human), which takes input from the user to control the car.

4.2.2 Simulation and State Management

In the TORCS simulation there are two defined types of time: *simulation time* and *real time*.
The simulation time is the time reference with which the simulation-related computations are performed, and its completely independent from real time. The real time, on the other hand, is the time observed by the user, and it's used to synchronize simulation time only in some specific game modes.

The TORCS simulation loop starts in the State Engine with an initial update and, if the race is still normally running, the State Engine loop will return into the same

state multiple times, till some condition for a state change is fulfilled (e.g. when the race has ended).

The simulation updates can be configured with four different operating modes:

* *Interactive Mode*: mostly suitable for interactive simulations, taking into account simulation time and real time. In this context, the simulation time is progressed until it has caught up to the real time and then proceeds with rendering one frame;

* *Blind Mode*: used for practice sessions, this mode considers only simulation time and no 3D scenery is rendered. The list in the GUI_G related the current progress is always updated every 2 seconds of simulation time;

* *Frame Capturing Mode*: this mode is used to capture frames at exact points in simulation time (e.g. every 0.03s). The time is not synchronized with real time, so the simulation might be faster or slower than real time, depending on the generated load;

* *Console Mode*: this mode has no GUI_G, it only prints on a terminal some progress information.

Regardless of the chosen mode, the simulation time step is executed by a Race Engine component during the game main loop.

Like most other Game Engines, TORCS manages the information about the current state of the engine with dedicated data structures. More specifically, TORCS game state is divided in three main components:

* *Race state*: which is the state related to the current race that is taking place, whether it is started, running, paused or stopped. This state is managed mainly through user GUI_G or functionalities (e.g. pause game), which are enabled by the core elements of the `raceengineclient` TORCS library;

* *Car state*: different for each car present in a specific race that is taking place, this state contains the information about the current state of the car (e.g. position, damage, speed). This state is updated during each step in the loop of the `simulation` module;

* *Space state*: this component contains the most relevant information about the general game state (e.g. current race time, number of players), even including a reference to the Car state itself. Like the Car state, it is updated during each step in the loop of the `simulation` module.

In general, the game state management is heavily centralized, with all the state-related structures and variables stored in the `raceengineclient` library of TORCS. In order to allow other TORCS components to modify the current state, the architecture makes use of C++ pointers, which are provided as parameter of most TORCS method calls.

4.2.3 Simulated Car Racing Championship Competition (SCR)

The work presented in the paper by Daniele Loiacono *el al.* proposes an interesting server-multiclient architecture for Simulated Car Recing Championship software [11]. The goal of this kind of competition is to design a controller for racing car that compete on many tracks alone and/or against other drivers.

The basic TORCS architecture comes as a stand-alone application, where the robots are compiled as separate modules and loaded into main memory only when a race takes place. This architecture comes with three major drawbacks:

* the races are not in real-time, since the execution of robots is blocking;
* there is no separation between the bots and the simulation engine, hence the bots have full access to all the data structures defining the track and the current status of the race;
* the only programming languages available for robots development are C and C++.

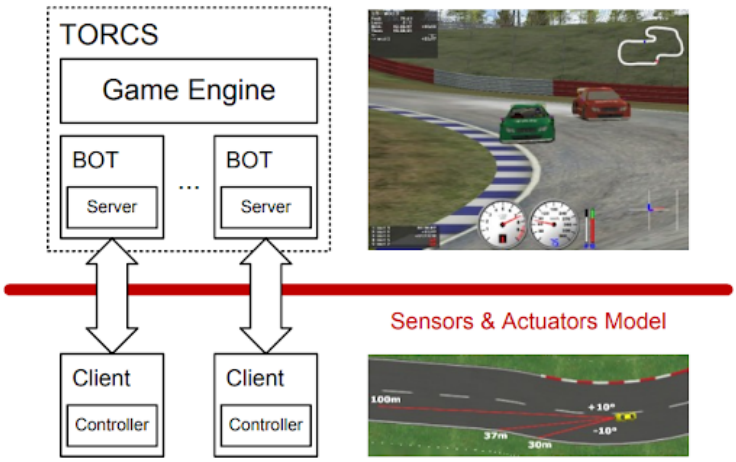


Figure 4.7: The architecture of the SCR software.

This proposal involves an evolution of the original TORCS architecture, with three important extensions:

* it structures TORCS as a client-server application, where the robots are run as external processes and connected to the race server using UDP_G;
* it introduces real-time races, with each game tick corresponding to about 20 ms of simulated time. In this context, the server sends the current sensory input to each bot and waits for 10 ms (real-time) to receive an action from the robot. Then, if no action arrives, the simulation continues with the last performed action, otherwise a new game state is computed with the new action provided;
* the competition software creates a physical separation between the driver code and the race server, with an abstraction layer, thus improving its resilience to cheating.

It also adds *scr-server* to the original architecture, which is a component dedicated to managing the connection between the game and a client robot using UDP_G.

Sensors and actuators

The input provided by the server to the driver robots consist of data about the current Car state, Race state and Space state. This information can be used by the drivers to compute the next action to send to the server, following the logic of its Artificial Intelligence (AI_G).
In order to obtain such data, the robots use a number of sensors, which come as part of the client structure and are able to read information from the environment surrounding the robot during the races (this includes the presence of opponent cars).

Moreover, the robots are able to control the car in the game through a rather typical set of actuators, which includes:

* the steering wheel;
* the gas pedal;
* the brake pedal;
* the gearbox.

Additionally, a *meta-action* is available to request a race restart to the server.

Chapter 5

Software development

*This chapter discusses the development done on the TORCS codebase, including the evaluation of multiple projects as possible starting point. The discussion includes the **containerization** of the main TORCS system components, the introduction of **additional software** and the development of **new components**.*

5.1 Codebase definition

Before beginning the work on the TORCS codebase, we performed an exploration and analysis of some TORCS related projects. In the following sections we discuss the two most relevant proposals we identified.

5.1.1 Patched TORCS 1.3.7

This project takes inspiration from the previously mentioned SCR architecture, introducing significant enhancements to the original TORCS software. It presents a codebase which is mostly congruent with the original 1.3.7 (latest) version of TORCS [51].

Original TORCS 1.3.7

The original codebase, in particular, includes the following high-level directories:

* *data*: containing useful data information about car models, tracks and UI_G elements;
* *doc*: containing general documentation related to the project functionalities and structure;
* *src*: containing the actual source code for the software, with its libraries and modules.

Inside the source code directory, we can find multiple lower-level directories related to the Game Engine modules and libraries, such as:

* *libs*: containing multiple TORCS utility libraries, as well as core components such as the `raceengineclient`, which is used to manage the main loop of the application;

- *drivers*: containing various AI_G implementations for handling robots (cars) behaviour. It also contains an "human" driver, with code related to user manual control of a car;
- *graphic*: containing code related to `ssgraph`, with elements to manage special effects, sound and general rendering of graphical elements;
- *simu*: containing code related to the management of the two possible types of simulation provided. This includes physic elements (e.g. collisions), but also data about the actuators functionalities used to drive the car (as described in section 4.2.3);
- *track*: containing code for managing the build, structure and status of the racing tracks.

The two main enhancements provided by this project are: an implementation of the *SCR-server* for managing the client-server communication with the robots, and a patch which allows to send the game image to another application using *IPC shared memory*.

SCR-server

The SCR-server implementation is composed of multiple files related to the management of the robot's sensors (e.g. mathematical computation of the readings) and a single file dedicated to define the actual behaviour of the server in the context of state-action communication. In particular, the server provides three main functions to: *initialize* a new race, *drive* during the race and *end* the current race. In order to start a new race, the component operates as follows:

1. binds a listen socket to a specific port (e.g. 3001);
2. waits for clients to connect and identifies them;
3. initializes the sensors for sending information to the client robots.

The function dedicated to driving the robot is more complex, since in manages a communication loop with the various remote drivers, operating as follows:

1. sends an identification message to each client using a socket;
2. updates the related sensors;
3. builds a string representing the current game state;
4. sends the state string to the remote clients;
5. waits (with a timeout) for a response action from the clients;
6. sets the control commands to the input action received and computes the next game state;
7. if no input is received and the timeout is reached, the old commands are used instead, to compute the next game state.

This configuration is particularly interesting for the purpose of our work, since it manages to successfully decouple the driver component from the main TORCS executable and also to implement the communication of the game state without the usage of pointers.

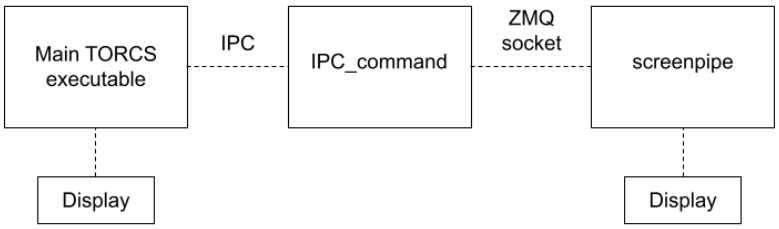


Figure 5.1: Proposed streaming architecture.

screenpipe component

As previously mentioned, this project tries to introduce the possibility of remotely streaming the game image to a different application, using a dedicated patch. This patch introduces two components, one for obtaining the game image data and serialize it (`IPC_command.cpp`), and one for remotely receiving and processing such data (`screenpipe_client.py`). As suggested by the files extensions, the components are written in C++ ad Python respectively.

In particular, the software and libraries used to accomplish such task are the following ones:

* *ZeroMQ*: which is a messaging library for distributed systems, used in both screenpipes's streaming components implement a socket-based communication. The peculiarity of the socket objects created using this library is the possibility to handle asynchronous messages;
* *OpenCV*: used as a real time artificial vision library, it is able to both obtain the image from the main game display (in the form of data) and also to reconstruct this same game image, based on the data transferred using the socket-based communication;
* *System V IPC*: which is a package able to provide shared memory between multiple processes on the same system, allowing them to share parts of their virtual space. In this context of application, it is used to allow the game image obtained from the main TORCS executable to be transferred to the local streaming component, run in a different executable;
* *Google protocol buffers*: used as a data format for serialization and exchange of streaming-related data.

While providing an reasonable starting point for implementing streaming communication, this configuration is incomplete in the version provided by the original project. As such, in our work, we elaborate more on this proposal, actually implementing a distributed version of game image streaming.

5.1.2 PyTorcs-docker

This projects is a further development, conducted starting from the previously discusses Patched TORCS 1.3.7 project [53]. More specifically, this proposal aims to implement a Python wrapper of the TORCS software, instantiable on Docker and using System V IPC to manage the game state.

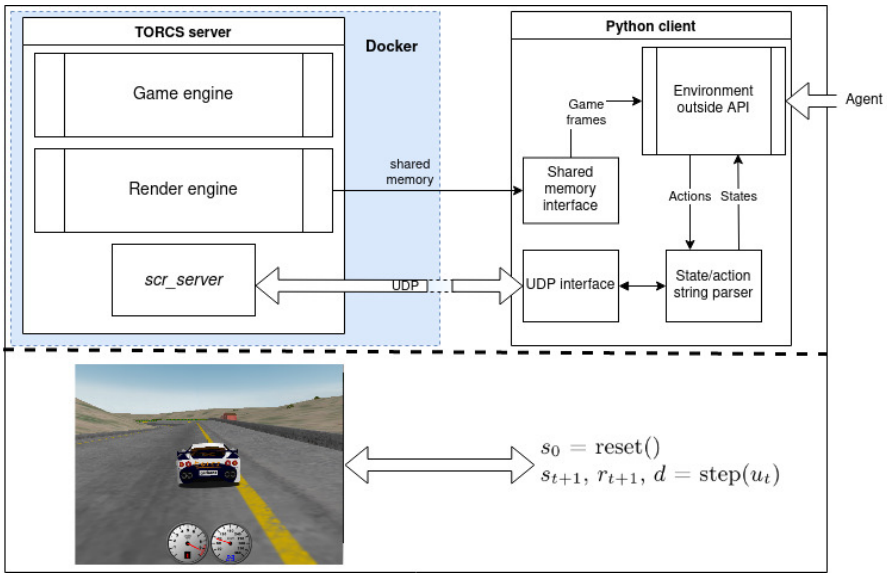


Figure 5.2: Representation of the PyTorcs architecture.

In order to efficiently run graphic components on Docker containers, this project proposes the usage of *NVIDIA Docker containers*, which have the peculiar characteristic of being GPU accelerated. They are, however, not simple to implement on any system, since they require access to the GPU hardware. Moreover, as of now, they are not compatible with *docker-compose*, which is also an interesting tool for implementing systems composed of multiple Docker containers, like in our work.

Similarly to the SCR project, this proposal presents an architecture where the driver is decoupled from the main TORCS executable. However, the PyTorcs architecture also manages to instantiate the executable into a dedicated Docker container, thus taking a step further in the same direction of our work.

As we can see in Figure 5.2, in order to share with the clients the information of the Render engine, the system makes use of *System V IPC* shared memory in a read-only manner (attach-detach, no semaphores).
Using the *SnakeOil3* library, developed for interfacing with TORCS using server extensions, this architecture is able to setup an UPD connection on port 3001 to the SCR-server and perform a constant state-action communication.

5.1.3 Codebase evaluation

The PyTorcs-docker project can be considered an evolution of the Patched TORCS 1.3.7 project, with some interesting developments that make use of approaches similar to what we intend to accomplish in our work (e.g. decoupling of modules, usage of Docker).
There are, however, some important aspects of this project that discourage its choice as a starting point for our work:

* introducing a Python wrapper on top of the original TORCS architecture can be problematic for modularization operations, since it could introduce an additional level of complexity when trying to translate the original Game Engine functionalities to a containerized virtual environment;
* the usage of System V IPC shared memory and sockets for game state/image is not necessarily the ideal approach in a distributed virtual environment with multiple containers;
* the implementation of NVIDIA Docker containers conflicts with the docker-compose tool, which is particularly interesting for the purpose of our work;
* this project removed multiple TORCS core functionalities (e.g. main menu) for the sake of simplicity. This can limit the scope of our project, which aims to modularize and containerize the TORCS Game Engine as a complete software, with all its single components.

As such, considering the interesting SCR approach to TORCS drivers decoupling, the final decision is to reference the Patched TORCS 1.3.7 codebase as a starting point for our project development.

5.2 Containerization of TORCS

Starting the development from the Patched TORCS 1.3.7 codebase, while the driver module is already decoupled from the main executable, no Docker image is included in the proposed architecture.
As such, we develop a new Docker image for the TORCS executable, with a dedicated Dockerfile that can be referenced to build a TORCS Docker container using the command `docker build`. This Docker image is based on the code provided at the project repository, which we call "TORCS_multi_docker".

The Dockerfile alone, however, is not able to directly import environment variables during the launch of the container. In order to circumvent this limitation, we introduce the `docker-compose` tool mentioned in section 3.1.2, which also allows us to run multiple containers at the same time and set cross-container configurations.

The possibility to import custom environment variables into the application container is particularly important for the implementation of *X11 (X Window System)*. This architecture-independent system can be used for remote graphical user interfaces and input device capabilities, which are both paramount features required by a virtual container running an interactable video game. More specifically, launching the TORCS Docker container will make use of the local display in order to render the video game image for the user, provided that the required permissions have been granted using the command: `sudo xhost local:root`.

As previously mentioned, in order to make use of the SCR architecture we require a different executable for the client running the driver module. Additionally, the client must be run on an environment that is separated from the main executable. As such, we develop a Docker image dedicated to this SCR client, referencing the code provided by the SCR project for the C++ Client.

Different Docker containers, however, cannot by default communicate with each other, as they are completely separate virtual environments. As such, using `docker-compose`, we setup a *Docker network*, which creates a bridge between the connected containers, allowing them to exchange data.

Finally, the Distributed TORCS architecture presented in Figure 5.3 is able to launch the TORCS software application, render its game image on the local display and allow a second container to connect to a "quick race" instance of TORCS.

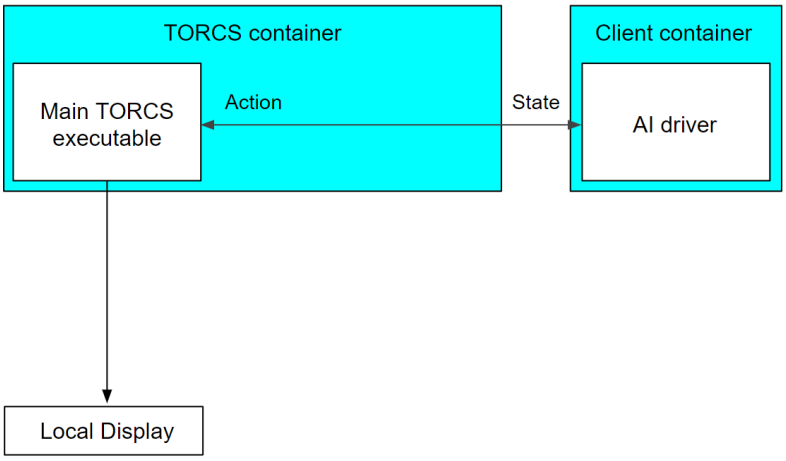


Figure 5.3: Distributed TORCS architecture - phase 1.

5.3 Distributed databases implementation

In order to allow the Game Engine modules located in different containers to communicate with each other, a dedicated mean of communication is required. In the implementation proposed by the Patched TORCS 1.3.7 project, the main TORCS executable and the SCR client communicate through a socket-based connection. This is a reasonable choice for simple action-state exchanges, however when multiple modules are accessing and modifying the same shared game state this approach might not be ideal. Inconsistency in the data read by different modules can, in fact, impact the video game functionalities. As such, we decide to implement data communication between the modules using a *shared storage distributed database*. In this context, we evaluate two different solutions, considered to both to be fitting for the purpose of our work, namely: *ETCD* and *Redis*.

Both solutions are provided with Docker images (ETCD image and Redis image) that can be freely implemented into applications, as such the Docker integration is simple and straightforward, provided that said images are configured to be connected to the same Docker network as the other containers.

Nonetheless, in order to interact with the distributed databases instances from the TORCS application, we also require $APIs_G$ and libraries dedicated to such purpose. As such, we decide to compile and link into TORCS both: etcd-cpp-apiv3 and redis-

plus-plus. In order to separate the two distributed databases implementations, we have two different and dedicated TORCS applications in our project repository.

The results of initial experiments exposed some performance issues with ETCD, mostly caused by the local storage of the history of all the keys that have been changed during execution, alongside multiple expensive DB_G snapshots. As such, we have applied some changes to the original ETCD application, in order to remove said expensive operations and correct some memory management issues arisen in our application context. The new ETCD Docker image referenced in the project is stored in our repository.

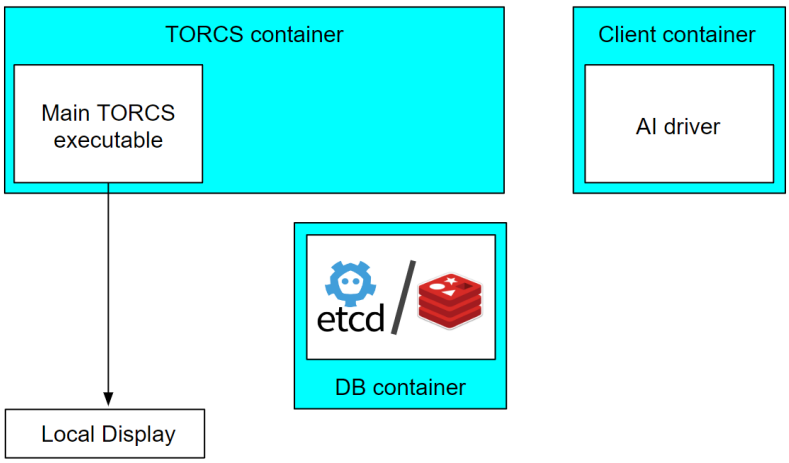


Figure 5.4: Distributed TORCS architecture - phase 2.

5.3.1 OrbitDB

An additional distributed DB_G we considered for implementation is OrbitDB [50]. This system is a decentralized database that uses the InterPlanetary File System (IPFS) as its storage layer. It is designed to be a lightweight distributed database, which requires no central authority to operate.

OrbitDB stores data in a database structure that is similar to a key-value store or a document store. Each database is associated with a unique address, and data is stored in a series of append-only logs that are replicated across the IPFS network. This allows multiple users to access and modify the database concurrently, without the need for a central server.

OrbitDB is implemented in JavaScript and designed to be integrated into applications developed using this language (either using Node.js or native). This was considered to be strong limitation, in our context of application, since TORCS is strictly developed in C++, with no way to interact with such system.

As such, this solution was discarded, in favour of ETCD and Redis distributed databases.

5.3.2 Distributed database clusters

The benefits of Redis Eventual Consistency mechanism, when compared with ETCD Strong Consistency mechanism, are not quite evident in a stand-alone configuration of these distributed databases. The lack of node replicas, in fact, prevents the need for replication operations, which generally slow down the performance of Strong Consistency mechanisms.
As such, since this core difference in functionality is not present, and the two systems end up working in a similar manner.

In order to more clearly highlight the benefits of the Redis distributed database, we implemented a 3-members cluster version of both Redis and ETCD, integrating it with our TORCS distributed architecture.

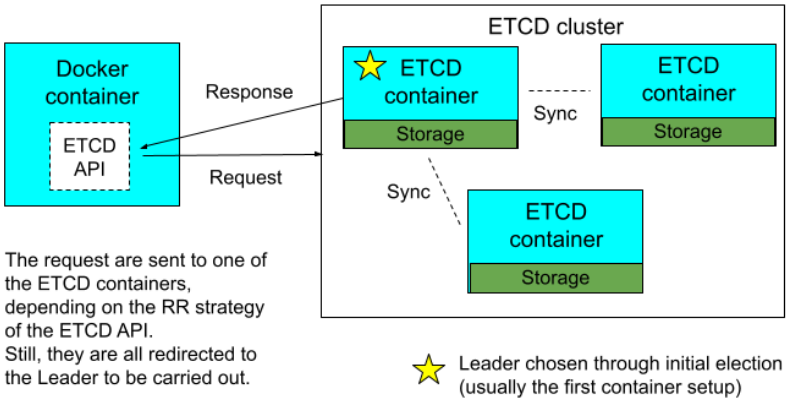


Figure 5.5: ETCD 3-members cluster configuration.

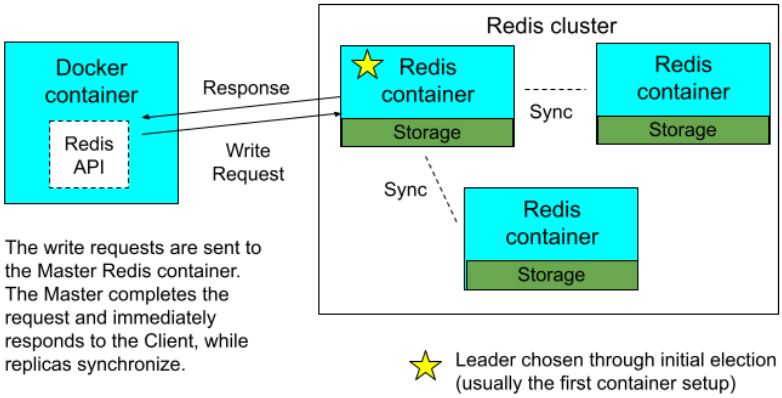


Figure 5.6: Redis 3-members cluster configuration.

As we can see in Figure 5.5, the requests in the ETCD cluster implementation can be sent to any of the ETCD containers, which directly respond to read requests and redirect write requests to the Leader. The choice of which container to send the request is delegated to the ETCD C++ API_G, which uses a Round Robin (RR_G) strategy to balance out the workload between the instances.

In Figure 5.6, on the other hand, we see how all the requests are directed towards the Redis Master node, which immediately responds to the client upon completion.

5.4 Game image streaming

As previously mentioned in section 5.1.1, another interesting functionality implemented in the Patched TORCS 1.3.7 project is the possibility to remotely stream the game image to another display.

This functionality is interesting for testing the performance and capabilities of our means of communication (a.k.a. the distributed databases). As such, we proceed with turning the previously socket-based communication into a data exchange performed using the shared memory provided by ETCD/Redis. In this context, the serialized data of the game image is stored into a specific key of the database, while the remote screenpipe component is notified upon each key change and is able to get such information through a read request.

Considering the weight of the data information used to describe a complete game image (multiple MBs), it is to be expected that the storage operation is quite expensive for a distributed database.
In fact, while the refactoring operation was successful, the streaming performance of the resulting system are not satisfactory, with a very low display framerate (<1 fps).

The reason for this low performance lies in the remote client component responsible for rendering the received image, which is not fit for efficiently perform such task. The component is, in fact, incomplete and developed using Python, which is not the best choice for applications required to process large amounts of data swiftly. Moreover, the sending component also performs unnecessary expensive operations, such as serialization and transfers of data between executables using System V IPC shared memory.

As such, we perform a complete refactoring of both streaming components (sender and receiver), developing them in C++ and significantly reducing the amount of elaboration operations. The result of this process is a much more efficient streaming functionality, which, despite exchanging data using a distributed database, is able to provide reasonable performance (about 30 fps).

Still, considering the excessively large amount of network communication and bandwidth required to implement such functionality at its full potential, we limit the image streaming framerate to 10 frames-per-second.

5.4.1 ETCD changes for resource management

One of the main problems arisen during the implementation of ETCD is the amount of resources needed for the storage of the history of all keys changed during the system execution. This data is stored into the hard drive and memory of the system where ETCD is run. As such, in situations where changes are frequent and the key history data rapidly increases in volume, the resources available can prove not to be sufficient for handling the system operations.

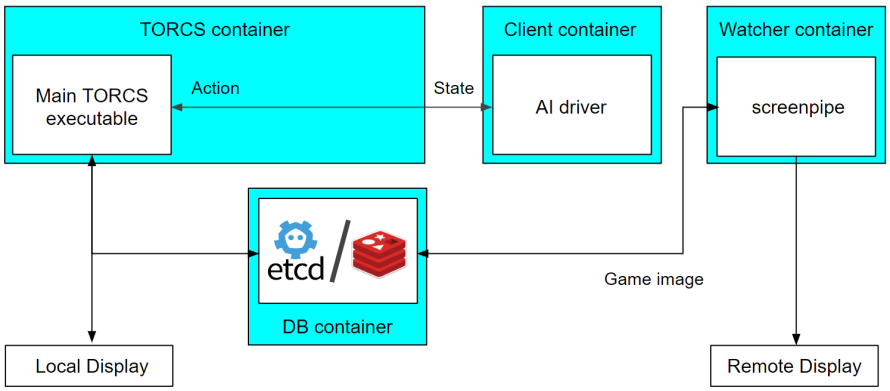


Figure 5.7: Distributed TORCS architecture - phase 3.

This is particularly evident in our context of execution, where the game image streaming, alongside other functionalities later discussed, cause a constant flux of large-sized requests.
As such, we operated some changes to the original ETCD software, by:

* removing the functionality which caused the saving and update of WAL_G files, used for snapshotting the database;
* instruct ETCD to create a new WAL_G folder at each database startup, restarting the database from a standard empty condition.

This was considered to be in line with the needs of our project, considering that long term storage of information was not required in this context. While this change greatly reduced the size of the data stored on the disk, the memory resources were still not managed efficiently enough. As such, we made use of two additional built-in ETCD functionalities: *automatic periodic compaction* and *defragmentation*.
Since the ETCD defragmentation can only be requested manually, we set up a custom script to perform it at certain intervals of time.

Finally, after noticing a delay between the end of the defragmentation operations and the actual freeing of memory resources, we assumed that the system memory management was actually delegated to the Go Garbage Collector. As such, in order to see an immediate effect on the available resources after a defragmentation, we introduced an additional explicit call to the Go Garbage Collector, at the end of the related function.

In the end, we managed to keep the actual DB_G in-memory size within 300 MB, during TORCS execution, which was deemed reasonable to guarantee not to reach resource saturation.

5.5 Distributed state-action communication

As discussed in the previous section, we managed to verify the possibility of replacing socket-based communication between TORCS system components, with a data exchange carried out through the shared memory of a distributed database.

We now proceed with performing a similar replacement operation also in the context of the SCR state-action communication that happens between the TORCS main executable and the remote driver.

In this context, as described in section 4.2.3, the SCR-server broadcasts a string representing the game state to all connected peers, awaits their response action and then processes the new game state based on the responses. This whole communication process can be implemented by writing both the game state string and the response action string on different keys in the distributed database, and have the components continuously check for key changes.

The result of this implementation is a new SCR client-server architecture, providing the same remote driving functionalities as the original system, but leveraging the characteristics of a distributed database.

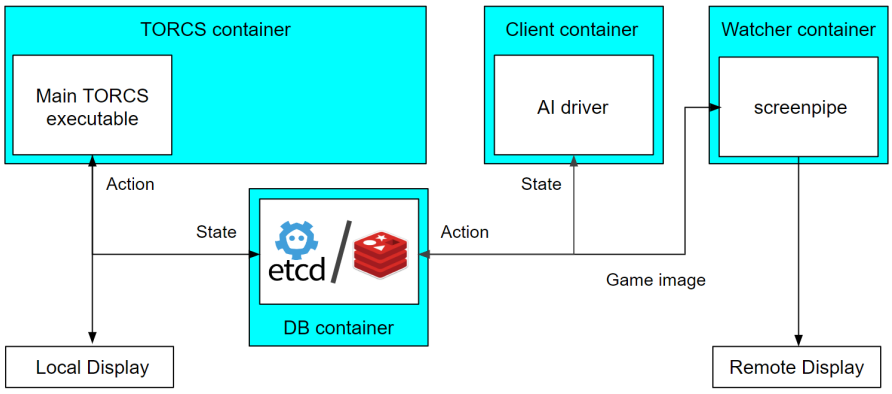


Figure 5.8: Distributed TORCS architecture - phase 4.

5.6 Music Player library

Up until this point, in the development process, we focused on containerizing Game Engine modules (e.g. AI_G driver) or introduce new functionalities (e.g. distributed databases, remote streaming). The TORCS software, however, is also composed of multiple libraries which may benefit from containerization.
These libraries, which we previously mentioned in section 5.1.1, are smaller sized with respect to complete TORCS modules. However, they are still responsible for managing core functionalities of the system (e.g. music).

Before beginning the decoupling operations, we perform a preliminary analysis on the quantity of external references and dependencies present in some of the libraries. In fact, as the number of references increases, the complexity of the decoupling operation increases as well, considering the side effects on the external referencing components that must also be adapted.

The result of this analysis highlights the *musicplayer* library as a reasonable candidate for containerization. This component is dedicated to managing the enabling and disabling of the music in the game menu, and presents no dependency towards

other components, making it quite simple to implement as an independent executable.

Considering the basic logic exposed by this component's methods, the implementation was realized based on the original socket-based communication between SCR client and server, where:

1. the main TORCS executable is launched and listen for messages from the musicplayer on a specific socket port;
2. the musicplayer component is launched and sends an identification message on a specific socket port, allowing the TORCS executable to identify it and proceed;
3. the musicplayer then processes any requests for enabling/disabling the menu music, coming from the TORCS executable on the same socket used for identification.

Using this configuration is possible to implement the same original functionalities of the TORCS musicplayer library, as a distributed component.

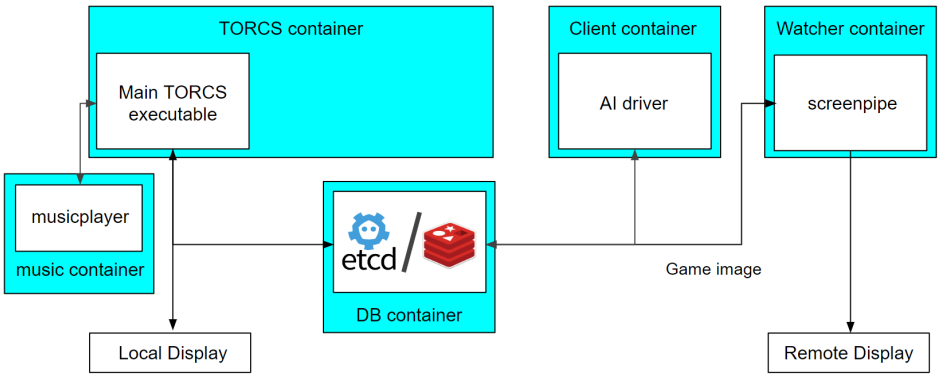


Figure 5.9: Distributed TORCS architecture - phase 5.

5.7 State Manager Middleware

One of the main goals of our development is to be able to effectively distributed information about the TORCS game state among its different decoupled components. However, during our experiments, which we will discuss in the following chapters, we identified a problem in implementing this game state distribution with the same approach as in other development phases (e.g. game image streaming, state-action communication).

In previous contexts the processing time required by the distributed databases to elaborate the requests was considered to be reasonable, with a low impact on the system performance. However, this is not the case in the context of TORCS internal game state management, where certain system components (e.g. graphic module, simulation module) present the requirement for rapid access to such data. In fact, the original implementation with C++ pointers, while centralized, is designed to allow for efficient game state data access, even between different system components.

In order to both allow for distribution of game state data and prevent significant impact on the system performance, we consider a different approach.
We develop a new component, dedicated to managing the storage of the TORCS game state into the distributed database with a set update frequency, independent from the original game loop. This component is also meant to be able to distinguish the nature of the specific game state fields to be stored, depending on the inconsistency tolerance of the user and the frequency of specific values changes, managing the storage operations accordingly.

The final goal is to have a background-running component, which is able to independently store or update the current game state, and to provide a reasonable degree of synchronization between multiple distributed TORCS Game Engine states.

5.7.1 Technical choices

This component, which we will call *State Manager middleware*, is run as a different thread, detached from the main TORCS executable, but running in its same environment (Docker container).
It makes use of `etcd-cpp-apiv3` (or `redis-plus-plus`) to be able to connect to the distributed database container, and perform read/write operations.
Moreover, it also includes the `chrono` and `thread` libraries, in order to introduce thread sleep lasting less than a second.

The update frequency has been set to 100 ms, considering the balance between inconsistency tolerance and the impact on the distributed database performance of frequent requests. Additional tests, with different frequency levels, can be conducted to find the best fitting value for specific system implementations.

The component is initialized through its *StartStateManager* method, which is called during the instantiation of a dedicated thread at the start of a new race. This is done in order to allow for background execution, without influencing the TORCS system performance.

We also distinguish between *static game state fields*, which are variables that compose the game state that are not updated after a race has been started, and *dynamic game state fields*, which are variable of the game state that are frequently updated during race execution.

5.7.2 Component methods

The State Manager middleware provides three methods:

* *StartStateManager*: accessible from other modules and taking `tRmInfo` (full game state data) as input parameter, this method performs the initialization of the `statemanager` component and its update loop;

* *SaveState*: only accessible inside the `statemanager` component and taking the current game state (`tSituation`) as input parameter, this method performs the storage on the distributed DB_G of the relevant game state fields and car state fields (for all cars);

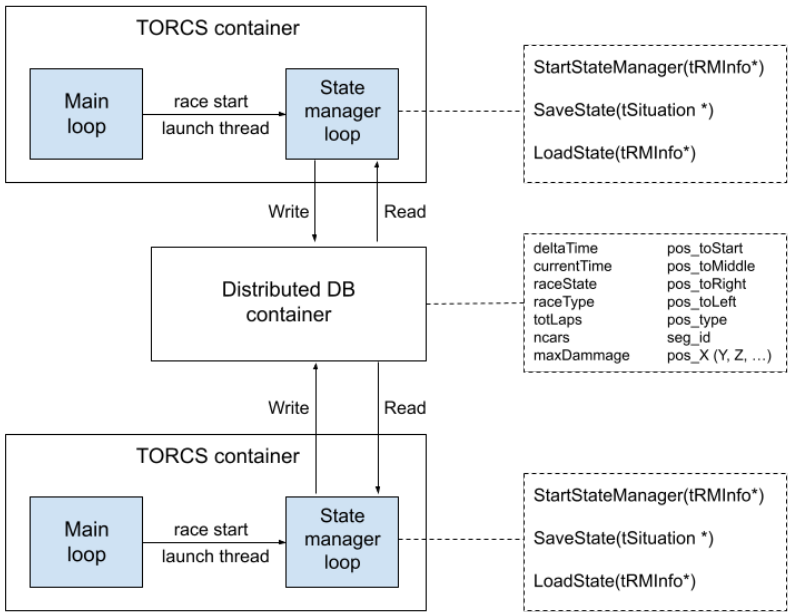


Figure 5.10: State Manager middleware architecture.

* *LoadState*: only accessible inside the `statemanager` component and taking `tRmInfo` as input parameter, this method reads from the distributed DB_G the relevant game state fields and car state fields (for all cars), updating the local state accordingly.

We can now elaborate more on each method's elaboration logic.

StartStateManager

This method operates with the following logic:

1. it checks whether a race is running by verifying if the `/state/currentTime` key is set in the distributed DB_G;
   (a) if it is, the following static game state fields are read from the distributed DB_G and set in the local game state:
      * `/state/raceType`
      * `/state/totLaps`
      * `/state/ncars`
      * `/state/maxDammage`
   (b) otherwise, the following static game state fields are written into the distributed DB_G:
      * `/state/raceType`
      * `/state/totLaps`
      * `/state/ncars`

   - `/state/maxDammage`

   The following dynamic game state fields are written into the distributed DB_G, to allow for later comparisons:

   - `/state/deltaTime`
   - `/state/currentTime`
   - `/state/raceState`

   The following dynamic car state fields are written into the distributed DB_G, for each car *"i"* to allow for later comparisons:

   - `/carstate/car[i]/pos_toStart`
   - `/carstate/car[i]/pos_toMiddle`
   - `/carstate/car[i]/pos_toRight`
   - `/carstate/car[i]/pos_toLeft`
   - `/carstate/car[i]/seg_id`
   - `/carstate/car[i]/pos_X`
   - `/carstate/car[i]/pos_Y`
   - `/carstate/car[i]/pos_Z`
   - `/carstate/car[i]/pos_AX`
   - `/carstate/car[i]/pos_AY`
   - `/carstate/car[i]/pos_AZ`

2. the game state update loop is started, performing a cycle each 100 ms;
3. if the race is paused, the `statemanager` waits without performing requests. Otherwise, it checks whether the distributed DB_G `/state/currentTime` is different from the local value;
   (a) if the remote value is greater than the local one, this means that the local GE_G values are outdated and the *LoadState* method is called;
   (b) if the remote value is lower than the local one, this means that the local GE_G is able to provide updated state values to the distributed DB_G, so the *SaveState* method is called.
4. after the Load/Save operations have been performed, the thread sleeps for 100 ms and then starts a new cycle.

Once the race has ended, all the distributed DB_G keys are removed as a clean up operation.

StartStateManager

This method operates with the following logic:

1. for each dynamic game state field, the remote value is checked and, if different from the local one, it is updated through a write request;
2. for each dynamic car state field of each car, the remote value is checked and, if different from the local one, it is updated through a write request;
   - the following values are related to the position of the car with respect to the specific track segment:

  - `/carstate/car[i]/pos_toStart`
  - `/carstate/car[i]/pos_toMiddle`
  - `/carstate/car[i]/pos_toRight`
  - `/carstate/car[i]/pos_toLeft`
  - `/carstate/car[i]/seg_id`
* the following values are related to the position of the car with respect to the global position:
  - `/carstate/car[i]/pos_X`
  - `/carstate/car[i]/pos_Y`
  - `/carstate/car[i]/pos_Z`
  - `/carstate/car[i]/pos_AX`
  - `/carstate/car[i]/pos_AY`
  - `/carstate/car[i]/pos_AZ`

LoadState

This method operates with the following logic:

1. for each dynamic game state field, the local value is checked and, if different from the one in remote value, it is updated accordingly;
2. for each dynamic car state field of each car, the local value is checked and, if different from the remote one, it is updated accordingly;
   * the following values are related to the position of the car with respect to the specific track segment:
     - `/carstate/car[i]/pos_toStart`
     - `/carstate/car[i]/pos_toMiddle`
     - `/carstate/car[i]/pos_toRight`
     - `/carstate/car[i]/pos_toLeft`
   * the value related to segment id (`seg_id`) is used to iterate on the double linked list present in the local car state, setting the current track segment depending on the id read from the distributed DB_G.
   * the following values are related to the position of the car with respect to the global position:
     - `/carstate/car[i]/pos_X`
     - `/carstate/car[i]/pos_Y`
     - `/carstate/car[i]/pos_Z`
     - `/carstate/car[i]/pos_AX`
     - `/carstate/car[i]/pos_AY`
     - `/carstate/car[i]/pos_AZ`
3. in order for the car state updates to be reflected in the game simulation, the *_reSimItf.config()* method is called.

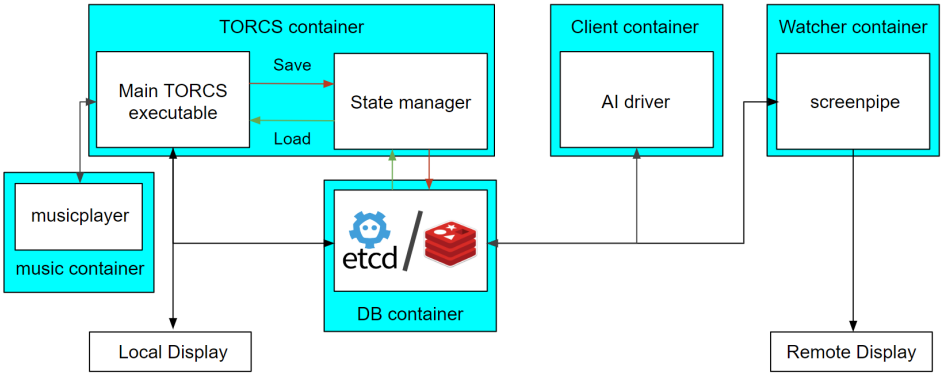


Figure 5.11: Distributed TORCS architecture - phase 6.

5.7.3 Final remarks

With the introduction of this component into TORCS, we can see at execution time how if we:

1. launch an instance of distributed DB_G.
2. run a race in a TORCS executable for a certain amount of time.
3. pause/stop the race.
4. launch a race on a second TORCS executable.

The `statemanager` on the second instance immediately synchronizes with the values stored into the distributed DB_G, updating the game time and teleporting the car to the position read from distributed DB_G. With some additional enhancements (e.g. add velocity and acceleration car state fields to distributed DB_G), it could be possible to implement a synchronization mechanism able to work in a distributed Game Engine environment.

5.7.4 Additional development

In order to verify the possibility to implement constant game state distribution, between multiple TORCS instances, we perform some changes to the original State Manager configuration. More specifically, we introduce the following modifications:

* we increase the update frequency from 100 ms to 1 ms (sleep time);
* we allow only the first TORCS instance launched to write on the distributed database, while the TORCS following instances are allowed only to read from the database;
* the read-only TORCS instances do not perform local simulation updates, as their state is updated by the State Manager.

We also add new car state fields for the storage operations, including speed-related ones:

* `/carstate/car[i]/vel_X`

* `/carstate/car[i]/vel_Y`
* `/carstate/car[i]/vel_Z`
* `/carstate/car[i]/vel_AZ`

And acceleration-related ones:

* `/carstate/car[i]/acc_X`
* `/carstate/car[i]/acc_Y`
* `/carstate/car[i]/acc_Z`

With this configuration, we are able to have multiple distributed instances of TORCS reproducing the in-game situation of the first TORCS instance, without any computation being performed by the game engines themselves. Moreover, the system is able to update the remote state of up to 10 cars in the same race.

Chapter 6

Experimental methodology

*This chapter discusses the single experiments conducted during the project, providing their **purpose** and **expected theoretical outcome**. Moreover, this chapter introduces the research methods and design that were used to conduct the studies, both for the **qualitative** and the **quantitative** experiments.*

6.1 Qualitative experiments

In the following sections we present the qualitative experiments, performed during the development of our project, in order to obtain a general idea about the effects on the system of specific software integrations or configurations. More specifically, we discuss the various data collection and analysis methods we used for these experiments and their expected theoretical outcome.

In the end, these experiments can provide us general numerical data, which is often not supported by multiple measurements or graphical representations, but meaningful enough to gain an insight on the positive/negative effects of technical decisions.

The software development direction in our project heavily relies on the outcomes of these experiments.

6.1.1 X11 forwarding performance assessment

Considering the initial status of the project, with a single image running TORCS in command line mode, one of the first steps of the development is to allow the display of the game image even when running it on a Docker container.

The most common approach to this task involves the use of X11 (X Window System) with *X11 forwarding*. This functionality, in particular, allows users to remotely access graphical user interface (GUI_G) applications on a remote server. On a practical level, the users are able to run GUI_G application on a remote machine, while displaying the application window on the local machine.
This technology, however, presents some drawbacks in terms of performance when managing remote image streaming over networks. As such, we decide to verify whether X11 can lead to bottlenecks, in our system.

Taking into account the network latency present between two remote machines, we expect to be able to identify a noticeable degradation on terms of graphics framerate, between the local and the remote game image, using this technology.

Methods & Configurations

In order to verify whether X11 can lead to bottlenecks, we prepare two machines connected to the same local network: a *Desktop PC* and a *Notebook PC*. The specifications of both machines are described as follows:

Table 6.1: Desktop machine specifications.

Component	Description
Operating system	Ubuntu 20.04.4 LTS - 64 bit
CPU	AMD Ryzen 5 2600 six-core processor x12
GPU	NVIDIA GeForce GTX 1060 6 GB
RAM	16 GB DDR4
Storage Memory (HDD)	1.3 TB

Table 6.2: Notebook machine specifications.

Component	Description
Operating system	Ubuntu 20.04.4 LTS - 64 bit
CPU	Intel Core i3-6006U (2.0 GHz)
GPU	NVIDIA GeForce 940MX 2 GB
RAM	4 GB DDR4
Storage Memory (HDD)	500 GB

This experiment is first conducted using an SSH_G *based connection* which, as per design, encrypts all data transferred using the channel. To allow such connection, with the X11 forwarding feature, we perform the following preliminary configuration operations:

1. install `openssh-server` on the Notebook PC, which allows remote clients to connect to the local machine and render on their display the applications GUI_G;
2. install `openssh-client` on the Desktop PC, which allows the local machine to connect to other machines, provided that they have `openssh-server` installed;
3. enable the usage of *Indirect GLX*, from the Window Manager configurations of both machines. This is an OpenGL extension to the X Window System, which is required for most GUI_G applications to be run through remote connections;

4. enable the X11 Forwarding option in the `/etc/ssh/sshd_config` file, present in the Notebook PC.

After these steps have been performed, it is possible to connect from the Desktop PC to the Notebook PC, using an SSH_G connection, requested through the following command: `ssh -Y <remote_user>@<notebook_IP>`. This command specifies that a *Trusted X11 Forwarding* should be used, with the `-Y` flag.

As previously mentioned, however, this configuration makes use of an SSH_G connection, which performs encryption/decryption of all transferred data. These operations can be expensive for the machines and possibly have a significant impact on the performance of the application. As such, we perform some additional operations to allow for a non-SSH_G connection, which completely rely on X11:

1. remove the `-nolisten tcp` flag present in the Window Manager configuration file, in order for the X11-Server to be reachable through local network connections;

2. use the command `xhost +` on both machines, to allow all connections to the X11-Server. This configuration is a security risk, and should only be used for local network experimental purposes;

3. set the `DISPLAY` environment variable on the Notebook PC to the Desktop display, using the command: `export DISPLAY=<desktop_IP>:0.0`;

4. using the `xauth generate $DISPLAY` command on the Desktop, generate a new token for accessing the X11-Server. Then, add this token to the Notebook using the `xauth add $DISPLAY . <token_hex>` command in the Notebook terminal.

Using these configurations, launching the TORCS executable on the Notebook PC actually renders its game image on the Desktop display, allowing for user input directly from the Desktop PC.

With both the SSH_G and the direct X11 configurations, we are able to see the **graphics framerate** of TORCS directly from the game image and the **latency between the two machines**, using the `hping` command.
These two values are deemed significant, in order to qualitatively evaluate whether the usage of X11 for this type of operations introduces significant performance degradation and is source of bottlenecks, taking into consideration the network latency present in a realistic environment.
Additionally, these values allow us to compare the two configurations and verify whether the SSH_G encrypting operations significantly impacts the application performance.

6.1.2 ETCD for SCR state-action communication

As discussed in section 5.5, one of the fist steps in the development of our system is the introduction of distributed databases, starting with ETCD, as a means to realize state-action communication between server and client in the SCR configuration.

In particular, we want to verify the possibility to implement such a configuration without a significant impact on the game performance (framerate) and without altering the functionalities of the AI_G drivers, which are managed with this communication.

Methods & Configuration

The configuration of the system, which acts as System Under Test (SUT_G), is the one presented on Figure 5.8. The game image streaming functionality is disabled, as not to introduce workload not correlated with the purpose of the test.
In terms of machine specifications, we reference the ones presented in Table 6.1.

The system, with this configuration, handles the state-action communication via the following procedure:

1. the TORCS executable (server) writes the current game state information as a `string` in the key `/gamestate/<port>`, where the port is identified by the one used by the client to connect;
2. the AI_G driver (client), which is watching the game state key for any changes, is notified when a new game state is written, and proceeds with computing the action to send as a response;
3. the AI_G driver writes the current game state information as a `string` in the key `/driver_action/<port>`.

Considering the time needed to process write/read requests in ETCD, it is possible that the game framerate and the system functionalities are negatively impacted, with respect to the original socket-based configuration. As such, we proceed to verify the variation of the framerate with respect to the original configuration, by looking at the in-game value fluctuation, and the movement of the car on the track.

Additionally, we also sample the average round-trip-time of the state-action communication in the original and the current configuration, to obtain a general idea of the difference between the two. In this context we do not compute the confidence intervals related to the measurements, as such we cannot make any assertion on the best performing configuration, out of the two.

6.1.3 Game image streaming solutions

After the development and integration of the game image streaming functionality, we aim to measure the performance of the *screenpipe* component of the distributed TORCS system. Moreover, our intention is to verify the difference between multiple configurations, with different environments and means of communication.
In particular, we experiment with two versions of the ETCD system, the original and our updated version, as described in section 5.4.1.
In terms of machine specifications, we reference the ones presented in Table 6.1.

Methods & Design

The measurements are conducted during a TORCS race, lasting 90 seconds, with the SCR client-server communication being active and running.
The variables we deemed to be relevant for this performance evaluation are:

* *graphics framerate* of the remote display;
* *average CPU usage* of the system-related processes or Docker containers;
* *average RAM usage* of the system-related processes or Docker containers.

The measurements are then conducted on the following system configurations:

* local environment using ZeroMQ as means of communication;
* local environment using the original version of ETCD as means of communication;
* local environment using out updated version of ETCD as means of communication;
* local environment using Redis as means of communication;
* Docker environment using ZeroMQ as means of communication;
* Docker environment using the original version of ETCD as means of communication;
* Docker environment using out updated version of ETCD as means of communication;
* Docker environment using Redis as means of communication.

For the Docker environments we do not bind the containers to specific CPU cores, as the distributed TORCS containers often make use of multiple cores. As such, any binding introduced would limit the resources available to the application and have an impact on its performance.
Moreover, the "updated ETCD" version referenced during this experiment is an intermediate version, with respect to the improvement discussed in section 5.4.1. This intermediate version includes the removal of DB_G snapshotting and the introduction of maintenance operations (compaction and defragmentation), but not the additional call to the Go Garbage Collection and the in-memory configuration.

6.1.4 3-members clusters benchmarking

As mentioned in section 5.3.2, the cluster implementation of both Redis and ETCD are significantly different from their stand-alone version, in terms of actual behaviour. While the stand-alone versions of the two systems present a similar request processing method, in the cluster version the Eventual Consistency mechanism of Redis should provide relatively better performance than ETCD.

In order to verify this assumption, we perform benchmarking operations on cluster configurations of these two systems. The number of nodes chosen for this tests is 3, since it is considered to be reasonably balanced both in terms of replication-related traffic and resources required to actually instance the cluster on a local environment.

It is important to remark that Redis, by default, implements the *sharding* of the key space. This means that the management of all the keys (e.g. 9000 key spaces), in the key-value database, should be split among at least three Master nodes (e.g. 3000 key spaces each), to have a stable configuration. However, such a requirement is problematic for our tests, since we intend to only instantiate three nodes: a Master and two replicas; in order to obtain exactly the same configuration between Redis and ETCD.
As such, we forced Redis to instantiate all the key space on a single Master node, which is not a recommended setting, but fits our testing purposes.

Considering the technical characteristics and the replication mechanisms of the two systems, we expect to be able notice better performance in the Redis cluster. Still, we also consider the impact of network latency on the two systems. As such, in situations with network latency, ETCD could be able to provide comparable performance with respect to Redis, thanks to its HTTP_G pipelining implementation.

Methods & Design

To perform significant performance evaluations and comparisons between Redis and ETCD, we first aim to identify the saturation point of one of the two systems. According to its documentation, Redis is hard to saturate, thanks to its high performance and in-memory data persistence. As such, we focus on identifying ETCD saturation point.

In the process of reaching this objective, we perform the following operations:

1. fix a large *amount of requests* (e.g. 300000);
2. gradually increase the number of *concurrent connections* making requests to the distributed database, until the saturation point is reached;
3. collect the *number of failed requests*, which, if different from 0, indicates that the saturation point has been reached.

Specifically, we set the following configurations:

* *number of requests*: 300000;
* *type of requests*: read and write;
* *size of the request*: 256 bytes;
* *number of clients*: 100 to 1000 with a step of 100.

Then, a pre-saturation value, for *concurrent connections*, is chosen for performing additional tests on both systems. These following tests focus on measuring the system *throughput*, for both read and write requests, with a gradually increasing amount of latency: from 0 to 20 ms, with a 2 ms step.

In order to carry out these tests, we make use of benchmarking tools provided by ETCD and Redis.
ETCD, in particular, provides a dedicated tool in its `etcd/tools/benchmark` directory, which can be used through specific commands, which allow multiple parameters. For our tests, the command used is the following:

```
go run main.go put -endpoints=172.20.0.2:2379 -conns=100 -clients=100 put
-total=300000 -val-size=256
```

Where we specify:

* the *Leader endpoint*, to which to send the requests;
* the *number of connections and clients*, which is the same for both settings and identifies the number of clients sending the requests to the distributed database;
* the *type of requests*, either GET or SET;

- the *number of total requests*, set to 300000 in order to provide a reasonable benchmarking interval;
- the *size of the request*, which is 256 bytes for both reads and writes;

On the other hand, Redis provides a benchmarking tool in its CLI_G interface: `redis-cli`. The command used for our test is the following:

```
redis-benchmark -h 172.20.0.2 -p 6379 -c 10 -t set -n 300000 -d 256
```

Where we specify:

- the *Master endpoint*, to which to send the requests;
- the *number of clients*, which identifies the number of clients sending the requests to the distributed database;
- the *type of requests*, either GET or SET;
- the *size of the request*, which is 256 bytes for both reads and writes;
- the *number of total requests*, set to 300000 in order to provide a reasonable benchmarking interval.

6.2 Quantitative experiments

In the following sections we present the quantitative experiments, performed during the development of our project, in order to obtain precise and numerical data about specific phenomena, effects on the system performance of specific configurations or correlation between system components. More specifically, we will discuss the various data collection and analysis methods we used for these experiments and their expected theoretical outcome.

The data provided by these experiments is supported by multiple measurements and graphical representations, which allow for clear numerical comparisons between different system configurations.

All the following experiments are conducted in a Docker environment, in a situation of *steady-state simulation*, which starts at the beginning of a race and terminates after 90 seconds.
In order to obtain numerical data, useful to perform accurate comparisons, we compute the *arithmetic mean* of the values of the collected samples, using the following formula:

$$\bar{x} = \frac{1}{N} \sum_{j=1}^{N} x_j$$

Then, we compute the *standard deviation*, using the formula:

$$s = \sqrt{\frac{\sum_{j=1}^{N} (x_j - \bar{x})^2}{N - 1}}$$

From which we can compute the *standard error*, using the formula:

$$se = \frac{s}{\sqrt{\text{sample size}}}$$

Finally, we compute the *confidence intervals*, using 95% as *confidence level* and the following formulas:

$$\text{lower bound} = \bar{x} - (1.96 * se)$$

$$\text{upper bound} = \bar{x} + (1.96 * se)$$

6.2.1 Network traffic analysis

One of the biggest limitations when developing systems based on network communication is the bandwidth available, since intense traffic can easily saturate the channel which is being used. Moreover, distributed databases, even if instanced on virtual environments, are often dependant on the physical hardware (e.g. hard drive, memory) which constitutes the hosting machine.

As such, when comparing different system solutions, we also need to take into consideration the amount of data transferred in input/output, both on the network and on the disk/memory. In particular, according to documentation, ETCD is considered to be heavily dependant on the performance of the hard drive used to store the database-related data.

Redis, on the other hand, is defined as an in-memory database. As such, its performance are mostly dependant on the local memory (RAM) and the network bandwidth available.

In general, we expect ETCD to be more demanding in terms of resources, if compared with Redis.

Methods & Design

The distributed system in this test is configured such that:

* SCR client-server communication is active and using a distributed database as means of communication;
* the game image streaming is active and using a distributed database as means of communication.

Thus, we perform the experiment in TORCS "quick game" mode (which makes use of the SCR communication) and expect to be able to correlate the data we collect with the two main functionalities we previously mentioned. It is important to remark that the screenpipe component has been bounded to perform image streaming up to 10 fps, in order to reasonably limit the network traffic generated by the streaming functionality.

In order to evaluate and compare the resource required by configurations of our distributed system, we focus on the following variables:

* *Network I/O*: indicating the amount of data sent and received over the network;
* *Block I/O*: indicating the amount of data written and read from the disk.

These data is provided by the command `docker stats` and written into a text file at regular intervals of 2 seconds, using a custom bash script. More specifically, we collect the data from the three main Docker containers interested by the elaborations: the TORCS executable container, the distributed database container and the client (AI_G driver) container.

Considering that these variables values are *monotonically increasing* with the number of samples, we compute the confidence intervals based on the following elaborations:

* the average *increase in Network Input*;
* the average *increase in Network Output*;
* the average *increase in Block Input*;
* the average *increase in Block Output*.

Finally, the two distributed databases implementations we evaluate and compare are: our updated version of ETCD, the standard version of Redis.

6.2.2 SCR client-server responsiveness

As described in section 4.2.3, the communication between client and server in this distributed TORCS implementation is a core functionality, which allows to decouple the AI_G driver from the main executable.
Still, different means of communication provide different performance, when implementing such functionality. As such, with this experiment, we aim to quantify and compare the performance of different distributed databases implementations. In particular, we consider:

* ETCD in its stand-alone implementation;
* ETCD in its 3-members cluster implementation;
* Redis in its stand-alone implementation;
* Redis in its 3-members cluster implementation.

Considering the design and characteristics of these two distributed databases, we expect the stand-alone versions to provide a similar performance, since no latency is involved and there is no theoretical benefit provided to Redis by its Eventual Consistency mechanism (as discussed in section 5.3.2). In the cluster versions, on the other hand, we expect Redis to provide better performance than ETCD, since it consensus mechanism should allow for a swifter elaboration of write requests.

Methods & Design

Focusing on the SCR communication, we configure the distributed system to have the game image streaming functionality disabled, since it introduced additional traffic which could impact the measurements. Moreover, we perform the experiment in TORCS "quick race" mode, where the SCR communication is present.

In order to obtain significant and comparable data for the different configurations, we focused on the measurement of the *round-trip-time*, for each round of communication

between the client and the server.
To collect such data, we set up a real-time counter in the SCR server, which measure the time elapsed between the writing of the game state onto the distributed DB_G and the reading of the next action. All the values obtained are then written on a dedicated text file.

Considering the large sample size, we perform a preliminary computation of the average between groups of 10 samples, which we call *partials*. Then we proceed with computing the average value from the partials, and the confidence intervals following the procedure discussed in section 6.2.

6.2.3 Network latency impact assessment

As we previously mentioned, a distributed system which bases its components communication on a network, is likely to be impacted negatively in terms of performance by the presence of network latency.
Thus, in order to quantify this impact on our TORCS distributed system, we artificially introduce an increasing amount of network latency into the system, while measuring the performance degradation experienced by the application.

Considering that the distributed database container acts as a means of communication between the various distributed TORCS modules, it is considered to be a suitable point for introducing the artificial delay.

In particular, taking into account the dependency of the main TORCS executable from the SCR communication, while operating in "quick game" default configuration, we expect the game framerate to be noticeably impacted by network delays. Additionally, since the network data transfer performed to implement game image streaming is very intense, we also expect to be able to notice a significant reduction in terms of screenpipe's graphics framerate.

In this experiment, we aim to compare the behaviour of ETCD and Redis, in both its stand-alone and 3-members cluster version. In general, in the stand-alone version, we expect ETCD to guarantee an higher degree of resistance to performance degradation caused by network latency, thanks to its $HTTP_G$ pipelining feature. Still, in the cluster version, Redis could provide benefits in terms of performance, thanks to it Eventual Consistency mechanism.

Methods & Design

The distributed system configuration is similar to the one presented in section 6.2.1, with both SCR communication and game image streaming, being active and running. As such, we still perform the experiment in TORCS "quick game" mode.

In order to reasonably evaluate and compare the application performance, we measure the values related to the following variables:

* the average *framerate* of the main TORCS display, in frames-per-second (fps);
* the average *framerate* of the remote screenpipe display, in frames-per-second (fps).

The measurements are conducted with a sampling interval of 1 second, on both displays. It is important to remark that the screenpipe component has been bounded to perform image streaming up to 10 fps, in order to reasonably limit the network traffic generated by the streaming functionality.

As previously mentioned, the network latency is artificially introduced only on the distributed database (`ddb`) container, using the command:

```
docker exec <ddb_container> tc qdisc add dev eth0 root netem delay <x>ms
```

Where `<x>` is the quantity of latency to be introduced. More specifically, with this command, we only introduce latency in the container's output communications.

The experiment involves the introduction of network latency with values that range from 0 to 20 ms, with a step of 4 ms, and from 20 to 100 ms, with a step of 20 ms. These values are deemed to be reasonable in order to most clearly represent the impact of both low, moderate and high latency on the system performance.

The four distributed databases implementations we evaluate and compare are: ETCD in its stand-alone version, ETCD in its 3-members cluster version, Redis in its stand-alone version, Redis in its 3-members cluster version.

6.2.4 Distribution of dynamic game state data

One of the core objectives of our project is to verify the possibility to replace the C++ pointers-based approach to game state distribution between TORCS components (described in section 4.2.2), with a solution based on shared memory and a distributed database.

Our general idea is to operate on each single game state related data field, as defined by TORCS codebase, replacing each reference to it with a write/read requests to the distributed database.
For instance, operating on a data field called `deltaTime`, we create a related key in the key-value storage of the distributed database, called `/state/deltaTime`. Then we replace each local value assignment operation with a write request sent to the distributed database, targeting the value key, and each local read of the value with a read request sent to the distributed database.

Through this logic we can completely remove the need for C++ pointers in the management of the game state. However, the processing of requests performed by the distributed database is not as fast as a read/write of the local value, and this can impact the performance of the TORCS application.
Moreover, the amount of requests sent to the distributed database could prove to be overbearing, and slow down its request throughput.

In general, we expect the TORCS performance (particularly the framerate) to be significantly degraded by this change, with an increasing impact as more data fields are stored into the database. In terms of requests, we expect to find a balanced situation, where introducing the management of additional data fields increases the number of requests being made. However, the database could likely reach a saturation point, after which the number of requests actually served cannot increase any more.

Lastly, we also consider the impact network latency can have in this context, experimenting with it both on stand-alone and cluster versions of ETCD/Redis. We expect network latency to have a very significant and negative impact on the system performance, considering the large number of requests being made. As such, ETCD should greatly benefit from its HTTP_G pipelining implementation.

Methods & Design

In order to understand the impact on the TORCS performance of storing information about the game state, we experiment on a system configuration where game image streaming and SCR client-server communication are both disabled.

We then focus on the following variables, deemed relevant for providing significant comparison between different configurations:

* the average *graphics framerate* of the TORCS main display;
* the average *number of write requests* processed by the distributed database, obtained through network sniffing, performed using the Wireshark software;
* the average *number of read requests* processed by the distributed database, obtained through network sniffing, performed using the Wireshark software.

The testing configuration change based on the following variables:

* the *number of game state data fields* stored on the distributed database, which varies from 0 to 3, including:
  - `deltaTime`;
  - `currentTime`;
  - `raceState`;
* the *network latency* introduced, which varies from 0 to 20 ms, with a step of 4 ms.

The game state data fields differ in nature and in the number of read/write requests being performed to manage their operations. In fact, we consider only data fields which are read or changed after their configuration at the start of a new race, which we call *dynamic data fields*.
Still, despite their differences, they are a significant factor for comparing the multiple system configurations. As such, we experiment both on stand-alone and cluster versions of ETCD/Redis.

6.2.5 Distribution of static game state data

With a logic similar to the one presented in the experiment discussed in the previous section, we operate a similar analysis on the *static game state data fields*. These data fields are written just once at the beginning of each race and then read multiple times through TORCS execution.

The aim of this experiment is to understand the impact on the game performance of storing game state data fields that generate almost only read requests, and verify the

specific source of the performance degradation. In particular, we first experiments with configurations with only write requests and then proceed to add interactions based on read requests.

In general, we expect to notice only a marginal performance reduction, regardless of the amount of static game state data fields stored into the distributed database. Moreover, this experiment is conducted only on a system configuration with a stand-alone version of ETCD.

Methods & Design

In order to understand the impact on the TORCS performance of storing information about the game state, we experiment on a system configuration where game image streaming and SCR client-server communication are both disabled.

We then focus on the following variables, deemed relevant for providing significant comparison between different configurations:

* the average *graphics framerate* of the TORCS main display;
* the average *number of write requests* processed by the distributed database, obtained through network sniffing, performed using the Wireshark software;
* the average *number of read requests* processed by the distributed database, obtained through network sniffing, performed using the Wireshark software.

The testing configuration change depending on the static game state data field stored, experimenting with just one different field each time, including the following:

* `totLaps`;
* `maxDamage`;
* `raceType`;
* `ncars`.

The game state data fields differ in nature and in the number of read requests being performed to manage their operations. These differences, however, are important to identify the actual source of the game performance degradation generated by the storage of different game state data fields.

6.2.6 Graphics and physics engine correlation

During our experiments we identified a possible correlation between the TORCS graphics engine and the physical engine (simulation module). In fact, we noticed that the delay introduced by managing the game state data through a distributed database, instead of using local C++ pointers, have a significant impact on the graphics framerate of the game, when the requests are made from the simulation module.

In the actual TORCS code, this correlation is even more clear, since both the function for managing the simulation and the graphics updates are located inside the same Game Engine loop. As such, considering the synchronous and sequential management of instructions in TORCS, delays in one component certainly end up impacting also

the other component operations.

In order to experimentally verify this correlation and quantify the impact on the system performance, we proceed with incrementally introduce artificial delay into the simulation module. Considering that the simulation is updated multiple times for each Game Engine step, we expect even small delays to have a significant impact on the system performance.

Additionally, introducing these delays on each frame elaboration prevents the system from being able to process more than a certain amount of frames-per-second (e.g. if each frame is experiments 20 ms of delay, no more than $1000/20 = 50$ fps can be processed). As such, we also consider the theoretical framerate upper bounds introduced by the delays and expect the actual measured framerate to be limited in these bounds.

Methods & Design

To numerically quantify the impact on the TORCS system performance of introducing delays into its simulation module, we experiment on a system configuration where no communication with the distributed database is performed. This is done in order to prevent any additional, non relevant, side effects.

We proceed with introducing an increasing amount of delay into the simulation update function, which is called inside the Game Engine step processing function. In particular, the method used to introduce the artificial delay is the following one:

```
std::this_thread::sleep_for(std::chrono::milliseconds(<x>));
```

Where `<x>` is the value in milliseconds of the delay we want to introduce, which ranges from 0 to 10 ms, with a step of 1 ms. This method does not perform *busy wait*, but instead *blocks* the thread for the specified amount of milliseconds.

The variables we consider to be significative in this experiment are:

* the average *graphics framerate*, which represents the performance of the TORCS system;
* the average *number of Game Engine steps*, which reflects the times a delay has been introduced into the simulation module.

The number of Game Engine steps computed for each graphics frame is variable and dependant on a condition which is tied to the *current game time*. As such, using the value in milliseconds of the delay introduced and the average number of Game Engine steps processed, we can compute the average delay introduced for each graphics frame, using the formula:

$$\text{avg. frame delay} = \text{delay} * \text{avg. num. of calls}$$

Then, we can compute the maximum number of frames that can be computed in a second, taking into account this delay, using the formula:

$$\text{fps up. bound} = 1000/\text{avg. frame delay}$$

Finally, we can compare the graphics framerate we measured with the bound we computed, as well as evaluating the impact on the performance of the delays introduced.

6.2.7 Graphics and game engine framerate correlation

As mentioned in the previous experiment, the TORCS graphics framerate is closely tied to the TORCS Game Engine steps computation, which we call *Game Engine framerate*.
To contextualize, the Game Engine framerate refers to the frequency at which the game simulation is updated, whereas the graphics framerate refers to the frequency at which the game is able to render and display new frames on the screen.

In order to quantify the ratio between these two values, we perform an experiment following a design which is similar to the one used for the one described in the previous section. We introduce an increasing amount of delay into the simulation module and then verify impact on both graphics and Game Engine framerate.
Additionally, we compute the actual *net operational time* of the system, considering the delays we are introducing.

Similarly to the experiment described in the previous section, we expect to notice a significant decrease in both framerates values as the delay increases. As well as seeing the actual operational time drop to a much lower value.

Methods & Design

The system configuration we used for this experiment is the same as described in section 6.2.6, with no communication performed with the distributed database. The delays are introduced into the simulation module with the same approach. However, the variables we consider for our measurements are:

* the average *graphics framerate*;
* the average *Game Engine framerate*;
* the *net operational time*.

The net operational time, in particular, is computed directly from the actual delay introduced and printed on a text file by the TORCS executable.

6.2.8 Temporal State Manager inconsistency

After the additional development performed on the State Manager middleware, as described in section 5.7.4, we aim to quantify the temporal inconsistency between the first TORCS instance launched (which only writes to the DB_G) and the following TORCS instances (which only read from the DB_G).

Despite being mostly successful, in fact, this implementation presents the following problems:

* the rendering of cars in the reading TORCS instances generates graphical glitches, which intensifies as the number of cars to be managed increases. They are also more noticeable in the ETCD configuration, rather than the Redis configuration;

* various data related to the general state of the game (e.g. UI_G) is slightly inconsistent.

Experimenting with both ETCD/Redis stand-alone/cluster configurations, we expect the temporal delay to be more intense in cluster configurations, particularly ETCD which should generally perform worse than Redis when processing large amounts of data in a short period of time.

Methods & Design

In this experiment, we configure the TORCS system to focus only on game state storage, without any game image streaming or SCR client-server functionalities. Then, we instantiate a "quick race" with 1 *human* driver and 9 other *local* AI_G controlled cars.

In order to specifically quantify the difference between the state of the local TORCS instance and the remote TORCS instance, we measure and compare the time *before writing the state* and *after reading the state* from the distributed database. Both these timestamps are printed on different text files by the two instances and then parsed for comparison.

In the end, we obtain the actual variable we are interested in, which is the *average temporal inconsistency* between the two instances.

Additionally, we also explore the variation of the inconsistency value with respect to the temporal segment of each sample. This is done in order to identify potential patterns of increments/decrements in state inconsistency value during the course of the race.

6.2.9 Positional State Manager inconsistency

In the same context of the experiment described in section 6.2.8, we also assume the presence of inconsistency in terms of the position of the cars which are represented in the read-only instances of TORCS. As such, we perform an additional experiment on both ETCD/Redis stand-alone/cluster configurations.

This analysis focuses on values related to the *global position*, *speed* and *acceleration* of the human player's car, comparing them between the first TORCS instance and a second one. Additionally, we also aim to understand the impact of network latency on the positional inconsistency of the cars. As such, we introduce an increasing amount of artificial latency on the distributed database and measure the values related to the factors we previously mentioned.

In general, we expect ETCD to perform worse than Redis in both its stand-alone and cluster versions, when no latency is present. However, thanks to $HTTP_G$ pipelining, it is possible for ETCD to present performance similar to Redis, when latency is introduced.

Methods & Design

In this experiment, we configure the TORCS system to focus only on game state storage, without any game image streaming or SCR client-server functionalities. Then, we instantiate a "quick race" with 1 *human* driver and 9 other *local* AI_G controlled cars.

The variables we consider to be relevant for this experiment measurements are:

* the *global position of the car*, with its coordinates. The inconsistency for this value is measured in *meters (m)*, as it is the unit of measure used by the TORCS software;
* the *speed of the car*, with its vectorial coordinates. The inconsistency for this value is measured in *km/h*, as it is the unit of measure used by the TORCS software;
* the *acceleration of the car*, with its vectorial coordinates. The inconsistency for this value is measured in *km/h*, as it is the unit of measure used by the TORCS software;

Considering the need to sample multiple local game state values at similar intervals of time in two different TORCS instances, we introduce an additional thread, launched at the beginning of a new race, which is responsible for printing data to a text file at a regular interval of time (1 ms).

The data printed include also a timestamp with nanosecond precision, in order to be able to compare the values of the two different TORCS instances. Considering the infeasibility of obtaining values at the exact same moment from both instances, we choose to compare the each game state of the first TORCS instance, with the game state of the second TORCS instance which is identified by the *closest higher timestamp*.

Considering the multi-dimension vectorial nature of the data we are comparing, in order to compute the difference between the local and remote game state, we operate using the following formula, for each factor:

$$\text{diff} = \sqrt{\sum_{n=1}^{N} (x_i - y_i)^2}$$

Where we have:

* N: as the number of dimensions for the single factor;
* x_i: as the value for the dimension i of the factor, in the second TORCS instance;
* y_i: as the value for the dimension i of the factor, in the first TORCS instance.

In the end, we are able to compare the positional inconsistency between the two instances with both ETCD and Redis in their stand-alone/cluster versions.

In addition to this comparison, we also explore the variation of the inconsistency value with respect to the temporal segment of each sample. This is done in order to identify potential patterns of increments/decrements in state inconsistency value during the course of the race.

Chapter 7

Experiments results & discussion

In this chapter provides the results of the experiments described in the previous chapter, with a clear representation of the data and highlighting the most relevant elements. These results are discussed then in order to provide a correlation with respect to the ***theoretical expectations*** *and the* ***research questions****.*

7.1 Qualitative experiments

In the following sections we discuss the results of the qualitative experiments, performed during the development of our project, in order to obtain a general idea about the effects on the system of specific software integrations or configurations. More specifically, we discuss the numerical and non-numerical results we obtained as general indicators of the behaviour of the system with respect to specific approaches.

The software development direction in our project heavily relies on the outcomes of these experiments, as such we also discuss the technical choices we make as a consequence of these results.

7.1.1 X11 forwarding performance assessment

The experiment on the remote TORCS game execution between two different PCs, making use of X11 Forwarding, provides us with the following results:

Table 7.1: X11 forwarding performance assessment.

Configuration	Average framerate	Average latency
Remote SSH_G with X11 Forwarding	6.200 fps	54.000 ms
Direct X11 Forwarding	7.100 fps	52.000 ms

As discussed in section 6.1.1, we performed our experiment with two different

configurations: with *remote SSH*$_G$ and with *direct X11 Forwarding* (no-SSH$_G$).
Still, both these configurations are set up to work on a local network, which inevitably introduces a certain amount of network latency. We consider this delay to be a significant factor in our experiment, for the configuration of a realistic testing environment. As we can see in table 7.1, for both configurations the average latency is measured to be around 50 ms, as such it is possible for this delay to have an impact on the resulting system performance.

The results we obtained, in terms of graphics framerate of the game, indicate generally low performance with both configurations, with around 6-7 fps for the remote game display. Considering these results, we can assert that X11 could, in fact, be a possible source for performance bottlenecks, when implementing systems which base their remote displaying functionalities on it.

Moreover, we also expected to observe a significant improvement in performance, when switching from an SSH$_G$-based connection to a non-SSH$_G$ connection, due to the lack of data encryption in the data transfer. As we can see in the table, however, this is not necessarily the case. Even if there seems to be an improvement in the system performance, the difference between the two configurations is only marginal and generally lower than expected.

7.1.2 ETCD for SCR state-action communication

After the introduction of on ETCD as a means for implementing SCR state-action communication, in place of the original socket-based connection, we observe the following system behaviour:

* there is little no difference in the behaviour of the remote AI$_G$ driver. The cars all drive correctly during the race, without crashing with other cars or running off the racing circuit, even with multiple different AI$_G$ controlled cars;
* there is occasional stuttering in the game image rendering window, with drops of about 30-40 fps, possibly caused by delays in the processing of certain state-action operations by the distributed database.

These results indicates that the ETCD implementation was successful, as the core driving functionalities are not significantly impacted by the change. The stuttering effect is generally not problematic for the correct management of the race, as the AI$_G$ driver are not impacted by this occasional phenomenon.

To provide a numerical representation of the time taken to manage each communication cycle, we present the round-trip-time (RTT$_G$) of each state-action exchange, in the two different configurations:

Table 7.2: SCR state-action communication RTT$_G$ - ETCD comparison.

Configuration	Average RTT$_G$
Socket-based communication	22.593 ms
ETCD-based communication	22.183 ms

From the results presented in table 7.2, we can see how both configurations, on average, manage to carry out each communication round in about 22 ms. Considering the radical change we performed in the management of the SCR client-server communication, we deem this result to be indicative of a successful distributed database implementation, for this specific purpose.

7.1.3 Game image streaming solutions

We now present the resource usage of multiple different system configurations, considering the purpose of implementing remote game image streaming.

Local environment with ZeroMQ

Table 7.3: Local environment with ZeroMQ.

Process	Average CPU usage	Average RAM usage
screenpipe	101.1% (2 cores)	0.96 GB
IPC_command	5.3% (1 core)	0.96 GB
Torcs	5.4% (1 core)	0.08 GB

Average framerate = 2.2 fps

In this context, the remote game streaming image is affected by significant latency for each frame, with causes it to get increasingly farther from the source image, as time goes on.

Local environment with original ETCD

Table 7.4: Local environment with original ETCD.

Process	Average CPU usage	Average RAM usage
screenpipe	150.0% (2 cores)	0.11 GB
IPC_command	105.0% (2 cores)	0.19 GB
Torcs	4.0% (1 core)	0.08 GB
ETCD	26.5% (1 core)	4.80 GB

Average framerate = 0.7 fps

In a local environment, ETCD is not significantly impacted by the increasing storage space required. However, the memory requirement, which also increases over time, has an impact on performance and can lead to resource saturation.
As we can see from table 7.8, the amount of RAM usage in this configuration is considerable. This is caused by the presence of a back-end database, which keeps the history of all previous key versions. As such, we identified the need for a new and updated ETCD version, fitting for our purposes.

Local environment with updated ETCD

Table 7.5: Local environment with updated ETCD.

Process	Average CPU usage	Average RAM usage
screenpipe	145.0% (2 cores)	0.11 GB
IPC_command	105.0% (2 cores)	0.22 GB
Torcs	4.0% (1 core)	0.10 GB
ETCD	15.5% (1 core)	4.80 GB

Average framerate = 0.7 fps

As in the previous configuration, in a local environment, ETCD is not significantly impacted by the increasing storage space required. However, the memory requirement, which also increases over time, has an impact on performance and can lead to resource saturation.
Even with our changes to ETCD, the amount of memory required is still quite significant, since the back-end database memory management is still reliant on the Go Garbage Collector to free the resources allocated during execution. As discussed in section 5.4.1, this result highlights the need for a direct call to the Garbage Collector and the introduction of an in-memory configuration, which removes any dependency from the disk performance.

Local environment with Redis

Table 7.6: Local environment with Redis.

Process	Average CPU usage	Average RAM usage
screenpipe	35.7% (1 core)	0.08 GB
TORCS + IPC_Command	4.0% (1 core)	0.10 GB
Redis	25.8% (1 core)	0.01 GB

Average framerate = 46.2 fps

In the local environment Redis performs very well, with low resource requirements and a stable high framerate for the remotely streamed image. For this experiment we temporarily removed the framerate limitations to the streaming functionality, and as such we can see framerate values over 10 fps.

Docker environment with ZeroMQ

Table 7.7: Docker environment with ZeroMQ.

Container	Average CPU usage	Average RAM usage
screenpipe	102.1% (2 cores)	0.54 GB
TORCS + IPC_Command	587.2% (6 cores)	0.19 GB

Average framerate = 1.9 fps

As in the local environment, the remote game streaming image is affected by significant latency for each frame, with causes it to get increasingly farther from the source image, as time goes on.

Docker environment with original ETCD

Table 7.8: Docker environment with original ETCD.

Container	Average CPU usage	Average RAM usage
screenpipe	101.2% (2 cores)	0.12 GB
TORCS + IPC_Command	582.6% (6 cores)	1.60 GB
ETCD	20.5% (1 core)	10.01 GB

Average framerate = 0.7 fps

In a Docker virtual environment, where the resources are limited, ETCD is significantly impacted by the increasing storage space required and can quickly reach the disk storage limit. Moreover, the memory requirement also increases over time, making the impact on performance even worse, considering the limited amount of resources available to the container.
The RAM usage of the ETCD container is even worse than what was measured in its local environment configuration, reaching up to more than 10 GB. This further validates the need for a more the need for a direct call to the Garbage Collector and the introduction of an in-memory configuration, which can remove any dependency from the disk performance.

Docker environment with updated ETCD

Table 7.9: Docker environment with updated ETCD.

Container	Average CPU usage	Average RAM usage
screenpipe	101.0% (2 cores)	0.11 GB
TORCS + IPC_Command	603.3% (7 cores)	1.71 GB
ETCD	9.4% (1 core)	10.08 GB

Average framerate = 0.7 fps

As in the previous configuration, in a Docker virtual environment, where the resources are limited, ETCD is significantly impacted by the increasing storage space required and can quickly reach the disk storage limit. Moreover, the memory requirement also increases over time, making the impact on performance even worse, considering the limited amount of resources available to the container.
The RAM usage of the ETCD container is even worse than what was measured in its local environment configuration, reaching up to more than 10 GB. This further validates the need for a more the need for a direct call to the Garbage Collector and the introduction of an in-memory configuration, which can remove any dependency from the disk performance.

Docker environment with Redis

Table 7.10: Docker environment with Redis.

Container	Average CPU usage	Average RAM usage
screenpipe	37.9% (1 core)	0.03 GB
TORCS + IPC_Command	355.4% (4 cores)	0.18 GB
Redis	25.7% (1 core)	0.01 GB

Average framerate = 45.4 fps

In the Docker environment Redis performs very well, with low resource requirements and a stable high framerate for the remotely streamed image. For this experiment we temporarily removed the framerate limitations to the streaming functionality, and as such we can see framerate values over 10 fps.

Conclusion

From the results we just presented, we can see that the configuration with a socket-based communication, using ZeroMQ, presents lower resource requirements and higher framerates than ETCD. This holds true both in local and Docker environments, even if ZeroMQ still presents very low graphics framerates in the remote display. Additionally, the ZeroMQ configuration introduces an increasing delay in the game image representation, while the ETCD-based version only presents a low framerate.

The main problem in the ETCD implementation is related to the memory usage, which keeps increasing over time, even if the storage space requirements are more limited in our updated ETCD version. As previously mentioned, the cause for this high RAM usage lies in the back-end database memory management, which is mostly left to the Go Garbage Collector.
These results are the reason behind the ETCD final software improvements described in section 5.4.1.

Between the option we considered, Redis is certainly the best performing solution, with low resource requirements in its local and Docker environment configurations, while providing high streaming framerate values.

7.1.4 3-members clusters benchmarking

We now provide a representation of the result we obtained from the analysis aimed towards identifying the saturation point of the ETCD distributed database.

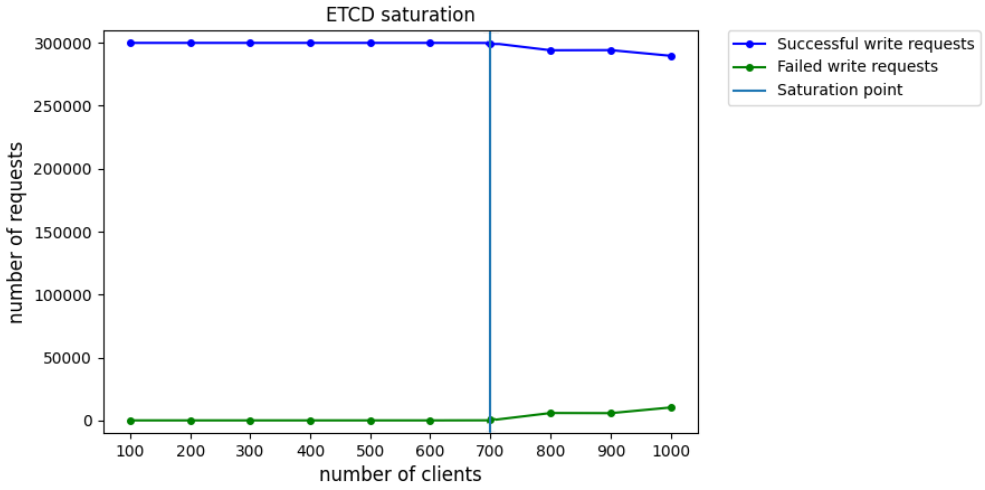


Figure 7.1: ETCD saturation point representation.

As we can see from Figure 7.1, the saturation point is reached when about 700 connected clients are creating workload through requests sent to the system, according to the parameters described in section 6.1.4.
Using the pre-saturation value (600 clients), we then proceeded with the second part of the experiment and the introduction of an increasing amount of latency into the system, while measuring its read/write requests throughput.
On the ETCD system, the results we obtained are as follows:

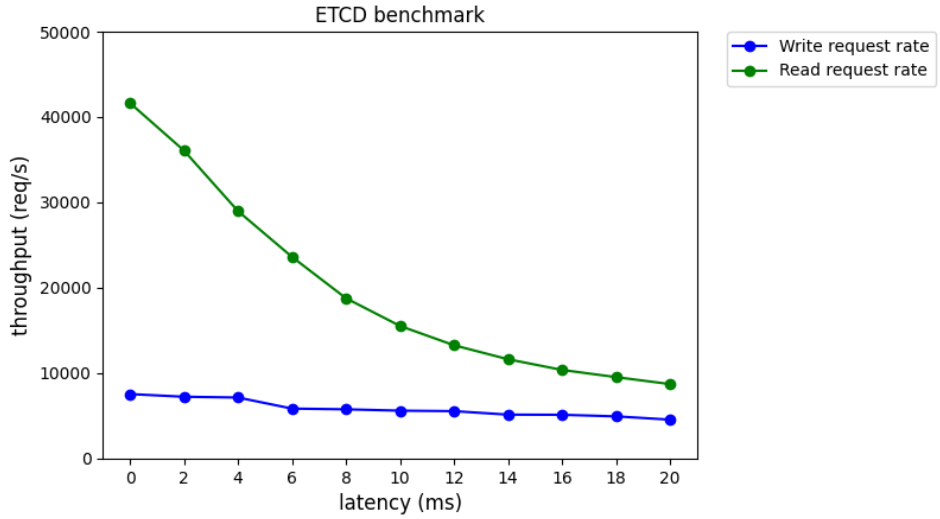


Figure 7.2: ETCD benchmarking.

As we can see from Figure 7.2, both read and write requests throughput is significantly

impacted by the latency we introduced. The write requests are relatively less impacted by the delay, thanks to the ETCD implementation of HTTP_G pipelining, which interests this type of requests.
On the Redis system, the results we obtained are as follows:

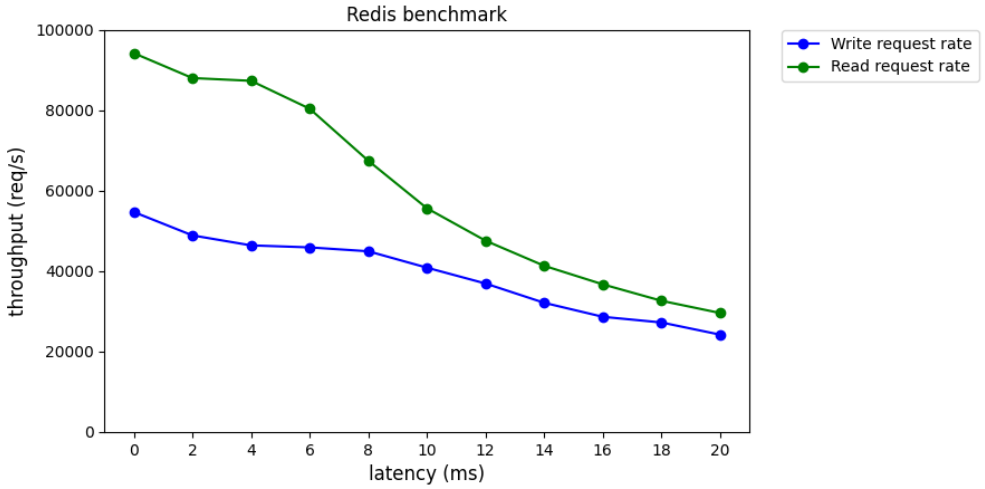


Figure 7.3: Redis benchmarking.

As we can see from Figure 7.3, both read and write requests throughput is significantly impacted by the latency we introduced, with a the read requests being slightly more affected than the write requests.

Systems performance comparison

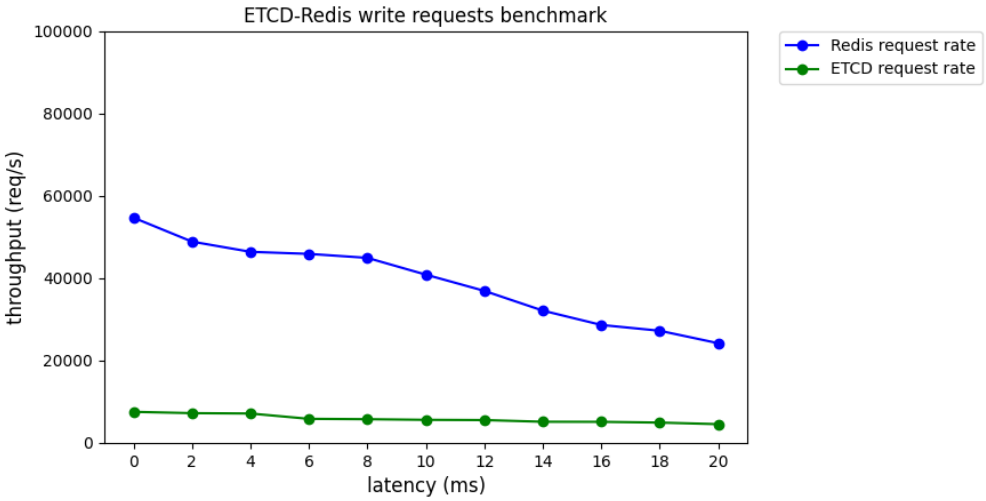


Figure 7.4: ETCD-Redis write benchmark comparison.

As we can see in Figure 7.4, Redis always performs much better than ETCD in terms of write requests throughput. However, the impact of latency in this context is

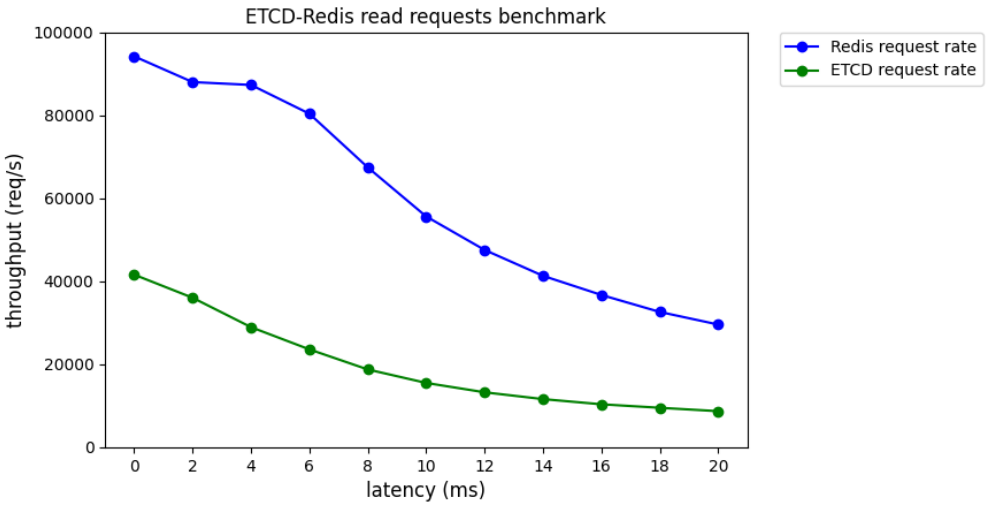


Figure 7.5: ETCD-Redis reads benchmark.

numerically greater on Redis than ETCD, and even proportionally the performance degradation in Redis ($\sim 55\%$) is greater than in ETCD ($\sim 39\%$).
This is an expected behaviour, since ETCD implements HTTP_G pipelining, which decreases the number of packets used to manage the same amount of requests, thus reducing number of elements affected by network latency.

In the context of read requests, as we can see in Figure 7.5, Redis still always performs better than ETCD. Additionally, while numerically the impact of latency on Redis is greater than on ETCD, proportionally the decrease in performance is greater on ETCD ($\sim 79\%$) than it is on Redis ($\sim 68\%$).

As such, we can conclude that, in terms of performance of their cluster version, Redis is generally superior to ETCD. This holds true both in contexts with no network latency and in contexts with network latency introduced. The result we obtained is in line with our expectations, considering that, in situations with Redis and ETCD clusters of 3 nodes, Redis can leverage its Eventual Consistency mechanism to guarantee better performance, with respect to systems based on Strong Consistency, such as ETCD.

7.2 Quantitative experiments

The following sections present quantitative experiments, performed in order to obtain precise and numerical data about specific phenomena, effects on the system performance of specific configurations or correlation between system components.

The data provided by these experiments is supported by multiple measurements and graphical representations, which allow us to compare and discuss different approaches and solutions to achieve the objectives of project.

7.2.1 Network traffic analysis

We now present the graphical representation of the network traffic measured for the three main container in our distributed TORCS architecture, including: main TORCS container, distributed database container, client container. The following representations will make use of a logarithmic scale, which facilitates the representation of values with different orders of magnitude.

ETCD system configuration

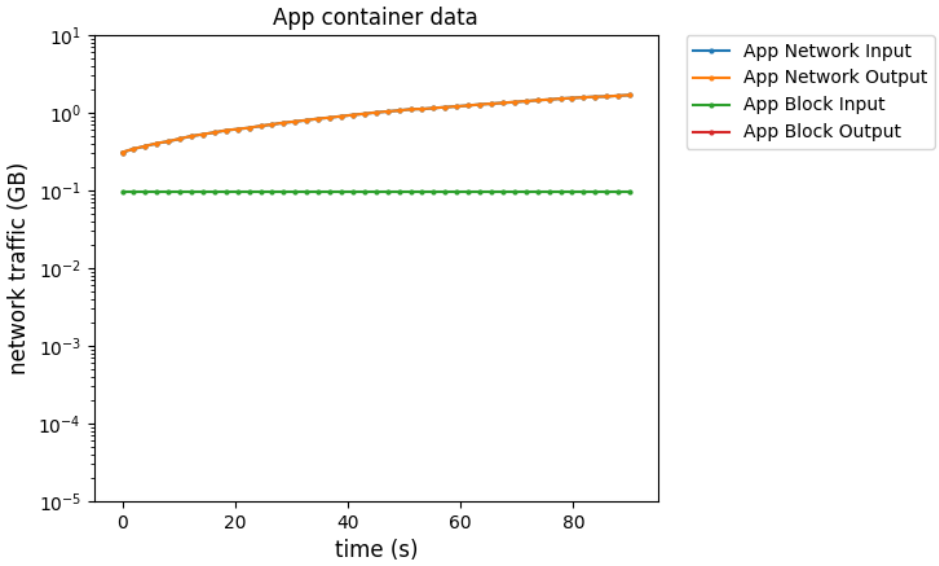


Figure 7.6: Network traffic in the TORCS application container (App).

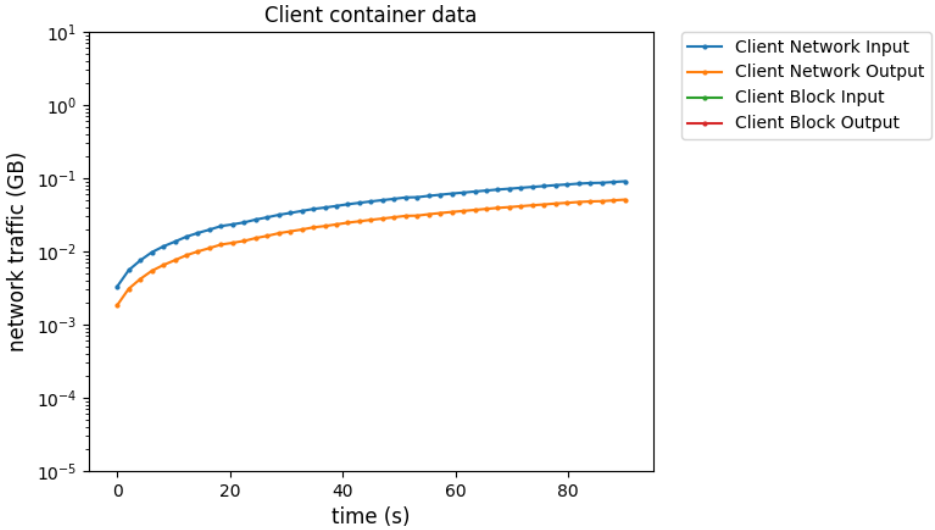


Figure 7.7: Network traffic in the remote AI_G driver container (client).

As we can see from Figure 7.7, no data is written to/read from the disk by the container, as its Block I/O values never increase.
As for the Network I/O, we can see in Figure 7.6 the amount of traffic generated by the remote streaming functionality and the game state writing component of the SCR state-action communication, which is particularly intense: in the order of multiple GBs, within 90 seconds of execution time. The AI_G driver, on the other hand, just presents network traffic values that can be related to the writing of the next action into the distributed database, as part of the SCR state-action communication.

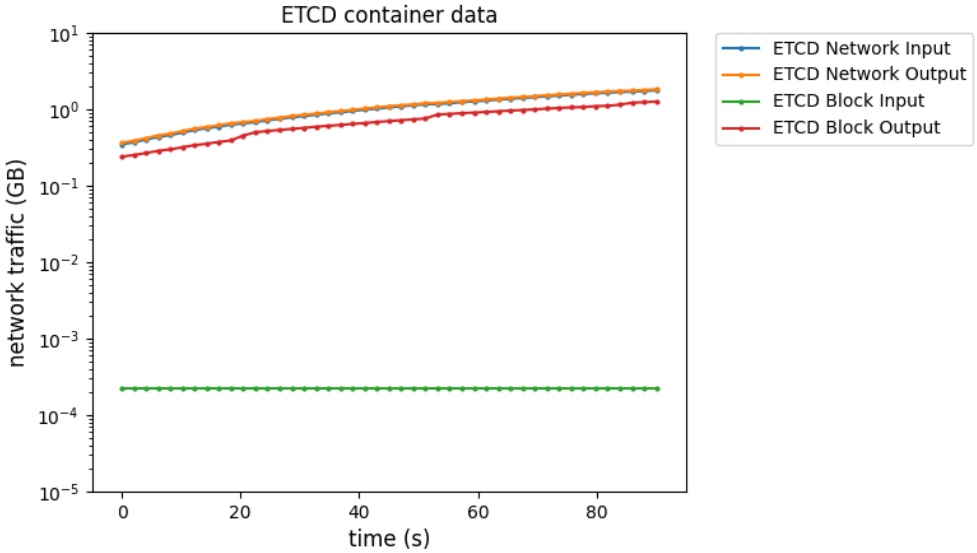


Figure 7.8: Network traffic in the ETCD container - ETCD.

In Figure 7.8, we see that a noticeable amount of data is written to the disk by the ETCD system, mainly related to the local snapshotting and storage of the database. Additionally, we notice how the Network I/O values can be directly correlated with the ones measured in the other two containers. These ETCD values, in fact, are constituted by the sum of the other Network I/O values, which satisfies our theoretical expectations, considering that ETCD acts as a means of communication between all other TORCS distributed system components.

Redis system configuration

As we can see from Figure 7.10, no data is written to/read from the disk by the container, as its Block I/O values never increase.
As for the Network I/O, we can see in Figure 7.9 the amount of traffic generated by the remote streaming functionality and the game state writing component of the SCR state-action communication, which is particularly intense: in the order of multiple GBs, within 90 seconds of execution time. In particular, we notice how the input network traffic is relatively low, since it only reflects the effects of the SCR state-action communication.
The output network traffic, on the other hand, reflects the much more intense communication caused by the remote game image streaming. This value results to be even larger than the one measured in the ETCD configuration, which is to be expected considering

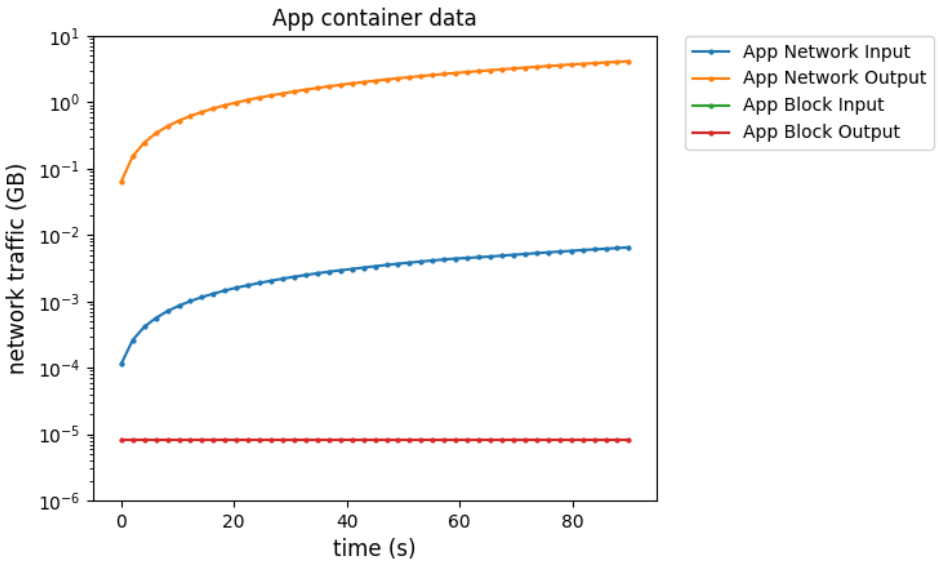


Figure 7.9: Network traffic in the TORCS main container - Redis.

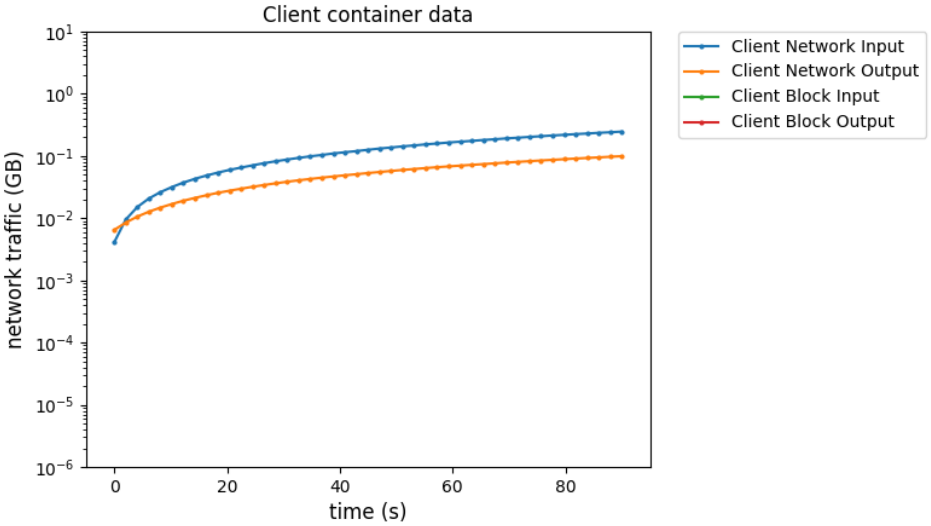


Figure 7.10: Network traffic in the remote AI_G driver container - Redis.

the lack of streaming framerate limitations, which were temporarily removed for this experiment.
In Figure 7.11, we see that no data is written to/read from te disk by the container, as its Block I/O values never increase. This is an expected behaviour, considering that Redis is an in-memory database, which should not persist any data on the disk, if not configured otherwise.
Additionally, we notice how the Network I/O values can be directly correlated with the ones measured in the other two containers. These Redis values, in fact, are constituted by the sum of the other Network I/O values, which satisfies our theoretical expectations, considering that Redis acts as a means of communication between all other TORCS

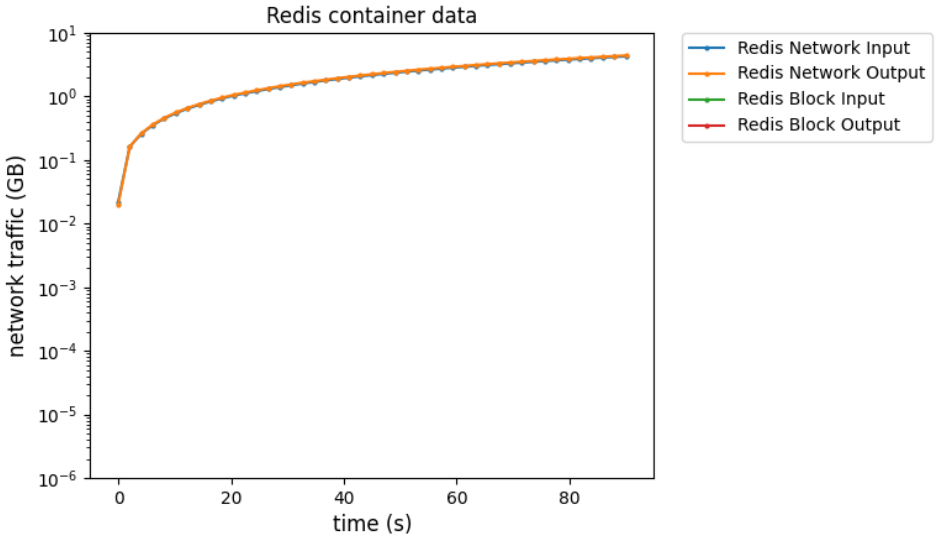


Figure 7.11: Network traffic in the Redis container - Redis.

distributed system components.

Conclusions

From the results we obtained, we can see how the behaviour of the system is generally similar between the two system configurations, with nearly the same amount of data transferred during SCR client-server communication. The main difference lies in the network traffic generated by the remote streaming functionality, which is greater in the Redis configuration, where there is not limit to the streaming framerate and the distributed database is able to handle the traffic introduced, as we saw in section 7.1.3.

Additionally, from the Block I/O values, we can see how the amount of data written/read from the disk by ETCD is quite significant, which introduces an dependency from the local hardware performance. This is not the case for Redis, as seen from its related measurements, where its in-memory nature allows it to operate without any local data persistence.

Finally, these results make us reason on whether a game image streaming implementation using a distributed database is actually a fitting solution. The large amount of data transferred, which can congest the distributed database, and the performance measured during the experiment discussed in section 7.1.3, indicate that this approach may not be ideal.

7.2.2 SCR client-server responsiveness

We present the results obtained from the experiment conducted with the aim of measuring the average round-trip-time (RTT_G) in the SCR client-server communication, on the four system configurations we mentioned in section 6.2.2.

Table 7.11: SCR client-server responsiveness.

Configuration	Average RTT_G	CI low. bound	CI up. bound	Sample size
Redis solo	22.600 ms	19.300 ms	25.900 ms	363
ETCD solo	23.700 ms	22.100 ms	25.400 ms	345
Redis cluster	22.800 ms	19.200 ms	26.300 ms	361
ETCD cluster	29.700 ms	25.000 ms	34.400 ms	278

Considering the stand-alone (solo) configurations of the system, we can see that both provide stable and comparable values. Moreover, the confidence intervals of the Redis configuration are completely included into the ones of the ETCD configuration, as such we can assert that the Redis implementation performs in the same range as the ETCD implementation, in our context of execution.

From the results obtained in the 3-members cluster configurations, we can see how the Redis configuration performs very similarly to its stand-alone version. This is due to its Eventual Consistency mechanism, which allows it not to be heavily influenced by the presence of replication.
ETCD, on the other hand, performs much worse in its cluster version due to the low scalability provided by its Strong Consistency mechanism.

If we compare the two systems cluster configurations, we can see a clear difference in terms of average RTT_G. Even if the Redis confidence interval upper bound still crosses with the ETCD lower bound, this overlap is much less significant with respect to their stand-alone versions. Moreover, considering the technical characteristics of the two systems, introducing additional nodes into the cluster would most likely make this difference even more evident.
As such, we conclude that in their cluster configuration, the Redis distributed database performs better than ETCD in terms of average RTT_G, in our context of execution.

7.2.3 Network latency impact assessment

We present the results obtained from our experiment with the introduction of an increasing amount of network latency on the distributed databases Leader node, measuring the framerate on both the main TORCS display and the remote streaming display.

ETCD stand-alone configuration

As we can see from Figure 7.12, as the amount of network introduced increases, the framerate rapidly decreases up to less than half of the original value, at the 20 ms mark. Then it further decreases, with a generally slower rate, as the delay introduced reaches 100 ms.
The remote streaming display framerate is also impacted by the introduction of network latency, however its decrease is generally limited, also taking into account the already low original no-latency value.

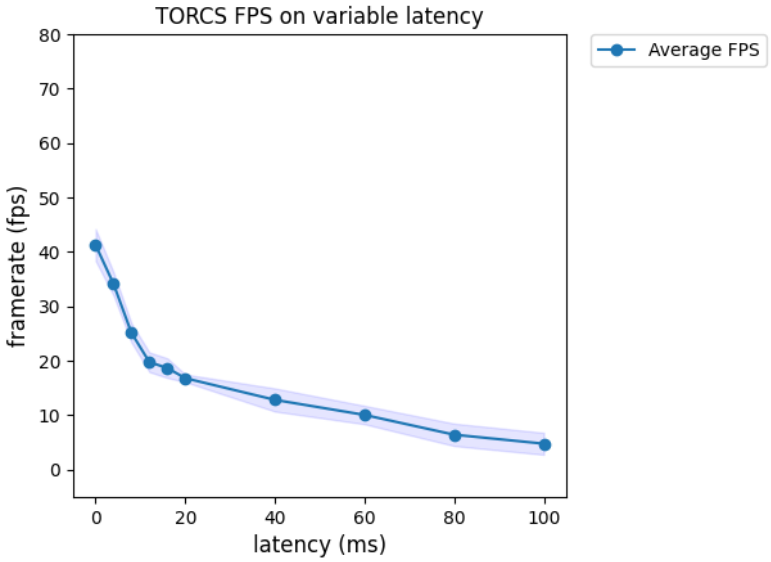


Figure 7.12: TORCS main display framerate variation - ETCD stand-alone.

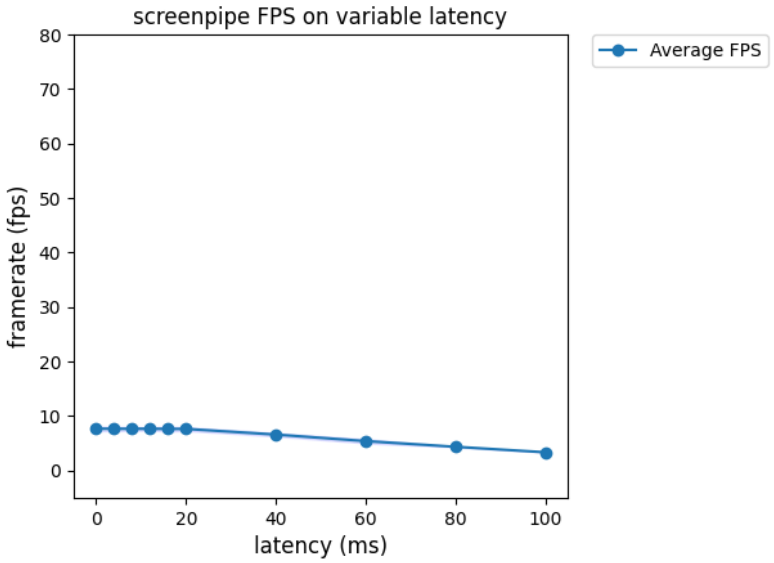


Figure 7.13: Remote streaming display framerate variation - ETCD stand-alone.

Considering this behaviour, we can assert that network latency heavily impacts the performance of the system in its ETCD stand-alone configuration, with particular reference to the game image streaming functionality and the SCR client-server communication.

Redis stand-alone configuration

As we can see from Figure 7.14, when no network latency is present, the performance in terms of framerate is very good, with more than 80 fps on average. However, as the amount of network introduced increases, the framerate rapidly decreases to

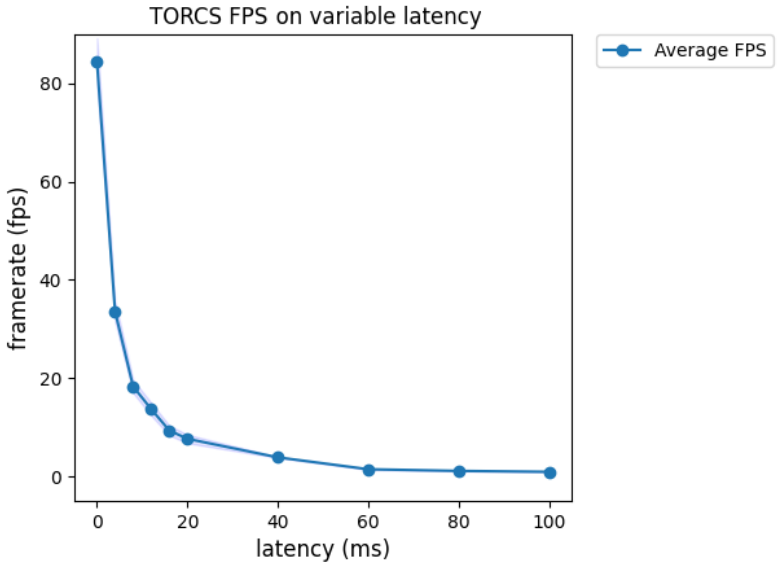


Figure 7.14: TORCS main display framerate variation - Redis stand-alone.

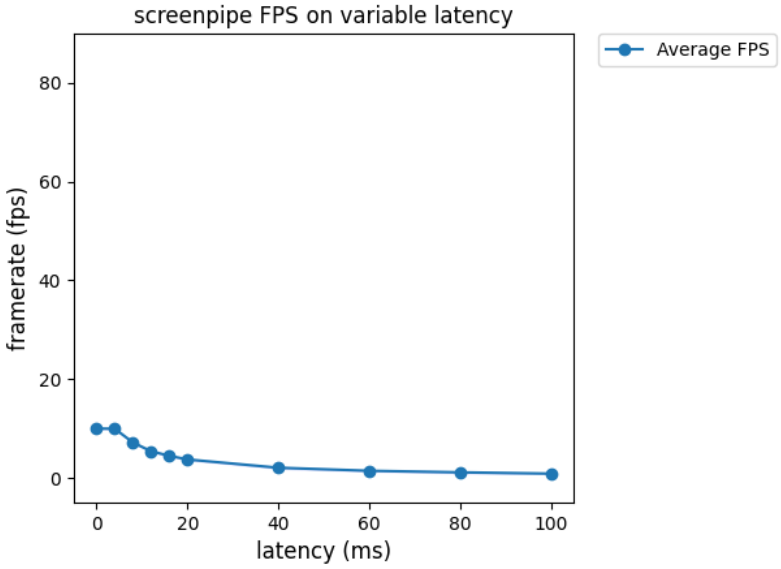


Figure 7.15: Remote streaming display framerate variation - Redis stand-alone.

less than 10 fps, at the 20 ms mark. Then it further decreases, with a generally slower rate, as the delay introduced reaches 100 ms. The remote streaming display framerate is also impacted by the introduction of network latency in a similar manner, with an initial sharper decrease and a more gradual one, from the 20 ms mark to 100 ms.

Considering this behaviour, we can assert that network latency heavily impacts the performance of the system in its Redis stand-alone configuration, with particular reference to the game image streaming functionality and the SCR client-server communication.

ETCD cluster configuration

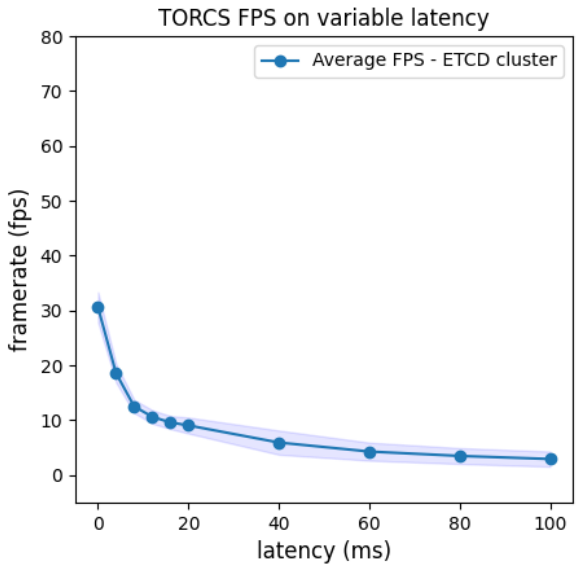


Figure 7.16: TORCS main display framerate variation - ETCD cluster.

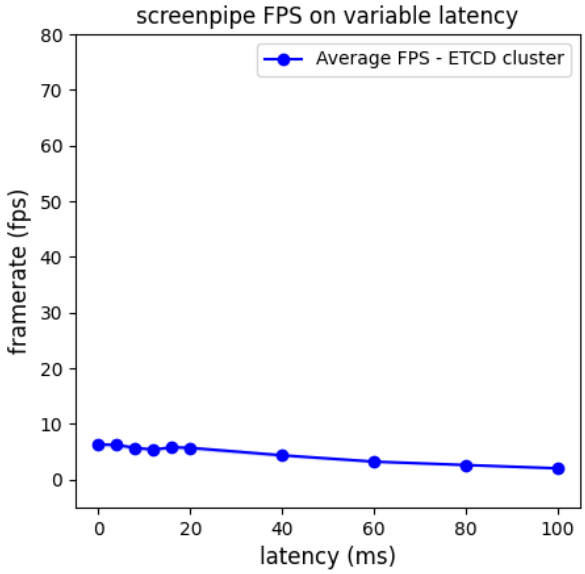


Figure 7.17: Remote streaming display framerate variation - ETCD cluster.

As we can see from Figure 7.16, as the amount of network introduced increases, the framerate rapidly decreases up to less than half of the original value, at the 20 ms mark. Then it further decreases, with a generally slower rate, as the delay introduced reaches 100 ms.
The remote streaming display framerate is also impacted by the introduction of network latency, however its decrease is generally limited, also taking into account the already

low original no-latency value.

Considering this behaviour, we can assert that network latency heavily impacts the performance of the system in its ETCD cluster configuration, with particular reference to the game image streaming functionality and the SCR client-server communication.

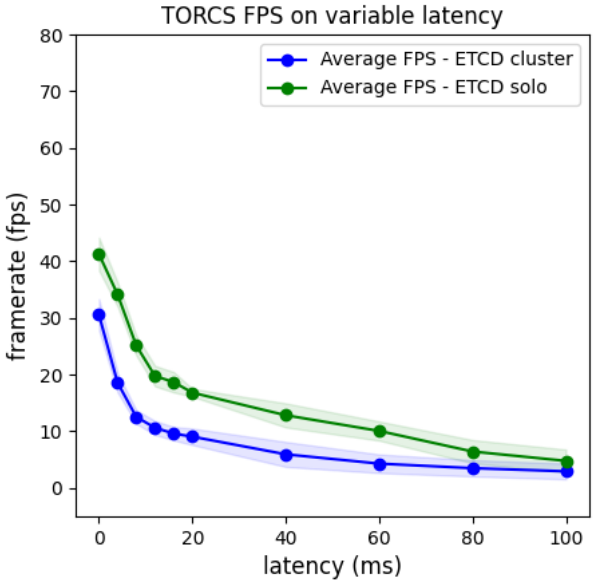


Figure 7.18: TORCS main display framerate comparison - ETCD stand-alone vs. cluster.

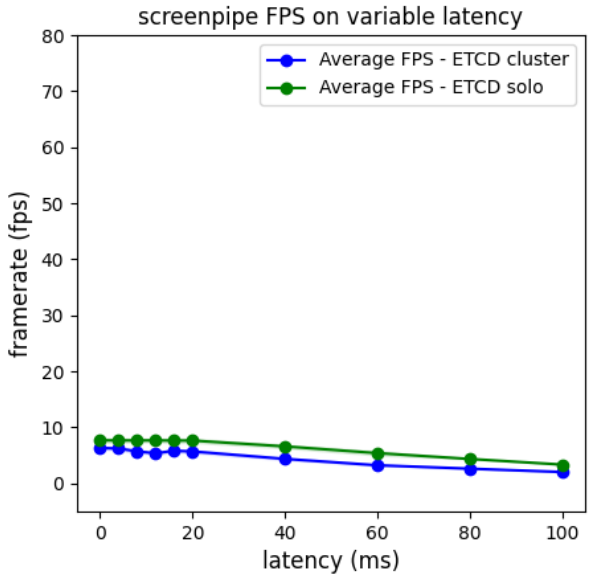


Figure 7.19: Remote display framerate comparison - ETCD stand-alone vs. cluster.

If we then compare the main TORCS display performance of the system in its ETCD

cluster version against the performance in its stand-alone version, as we can see in Figure 7.18, the stand-alone performs better overall. This is an expected behaviour, considering the Strong Consistency mechanism implemented by ETCD, which requires the Leader to wait for all the replicas to synchronize before proceeding with processing further requests. As such, if more replicas are introduced they delay is likely to increase.

Redis cluster configuration

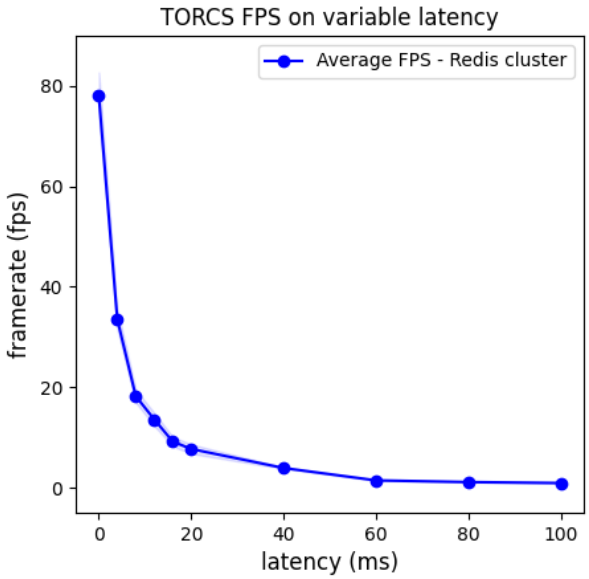


Figure 7.20: TORCS main display framerate variation - Redis cluster.

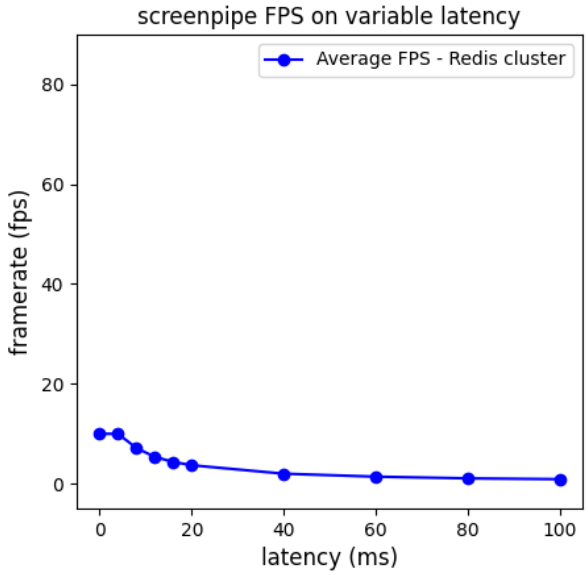


Figure 7.21: Remote streaming display framerate variation - Redis cluster.

As we can see from Figure 7.20, when no network latency is present, the performance in terms of framerate is very good, with almost 80 fps on average. However, as the amount of network introduced increases, the framerate rapidly decreases to less than 10 fps, at the 20 ms mark. Then it further decreases, with a generally slower rate, as the delay introduced reaches 100 ms.
The remote streaming display framerate is also impacted by the introduction of network latency in a similar manner, with an initial sharper decrease and a more gradual one, from the 20 ms mark to 100 ms.

Considering this behaviour, we can assert that network latency heavily impacts the performance of the system in its Redis stand-alone configuration, with particular reference to the game image streaming functionality and the SCR client-server communication.

If we then compare the main TORCS display performance of the system in its Redis cluster version against the performance in its stand-alone version, as we can see in figures 7.22 and 7.23, they are mostly congruent. This is an expected behaviour, considering that there is no functional difference in Redis between having replicas or not, as its Eventual Consistency mechanism allows the system to proceed with elaborating requests without awaiting for replica synchronization to finish.

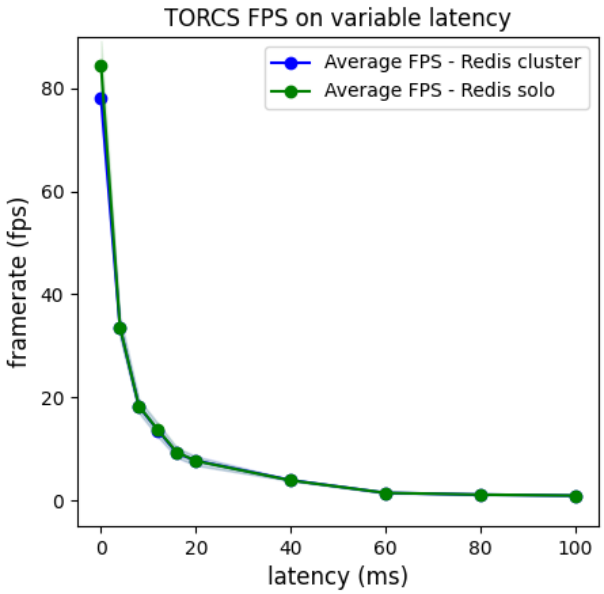


Figure 7.22: TORCS main display framerate comparison - Redis stand-alone vs. cluster.

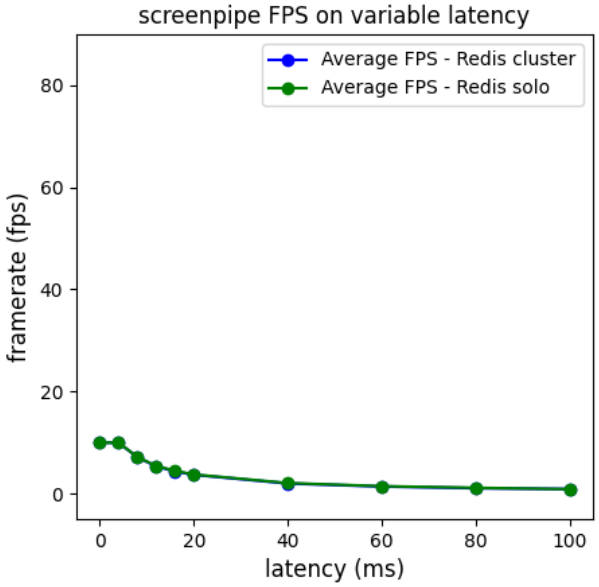


Figure 7.23: Remote display framerate comparison - Redis stand-alone vs. cluster.

Stand-alone configurations comparison

After discussing the behaviour of the single system configurations in relation to the introduction of network latency, we can now proceed with comparing their stand-alone configurations with each other.

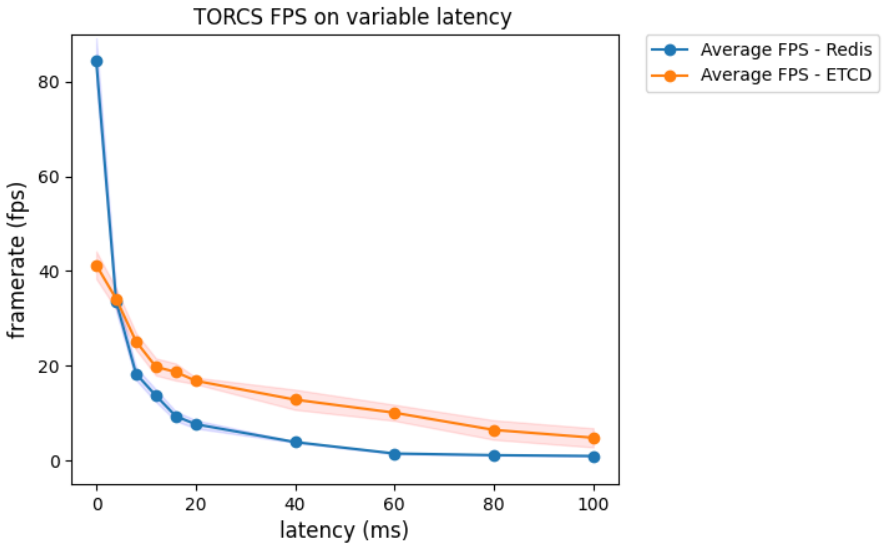


Figure 7.24: TORCS main display framerate comparison - stand-alone.

As we can see from Figure 7.24, the TORCS display framerate is much higher in the Redis configuration, when no latency is introduced. However, the introduction of even low amounts (~ 4 ms) of latency generate a shard drop in the Redis system

performance, decreasing it to the same level or lower than ETCD.

This behaviour is likely tied to the ETCD implementation of HTTP_G pipelining, which helps it mitigating the effects of latency on its operations. Redis, without a dedicated in-code implementation, does not provide the pipelining functionality, as such it is to be expected that its performance gets outclassed by ETCD in situations with moderate to high latency.

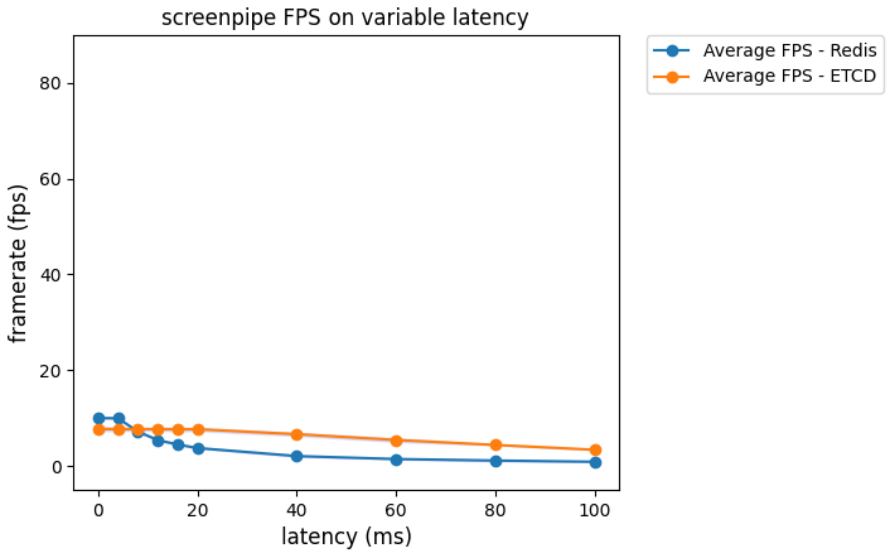


Figure 7.25: Remote display framerate comparison - stand-alone.

The situation in the remote display is quite similar, as we can see from Figure 7.25, with Redis providing better performance in situations with little to no network latency and ETCD providing better performance in situations where latency is introduced.

Cluster configurations comparison

After discussing the behaviour of the single system configurations in relation to the introduction of network latency, we can now proceed with comparing their cluster configurations with each other. As we can see from Figure 7.26, the TORCS display framerate is much higher in the Redis configuration, when no latency is introduced. However, the introduction of even low amounts (~ 4 ms) of latency generate a shard drop in the Redis system performance, decreasing it to the same level or lower than ETCD. This behaviour is likely tied to the ETCD implementation of HTTP_G pipelining, which helps it mitigating the effects of latency on its operations. Redis, without a dedicated in-code implementation, does not provide the pipelining functionality, as such it is to be expected that its performance gets outclassed by ETCD in situations with moderate to high latency.

Still, with the cluster configuration, Redis is able to provide additional benefits related to the replication mechanism. Leveraging the characteristics of its Eventual Consistency mechanism, Redis is able to provide generally the same performance as its stand-alone version, differently from ETCD which does not perform as well in terms of

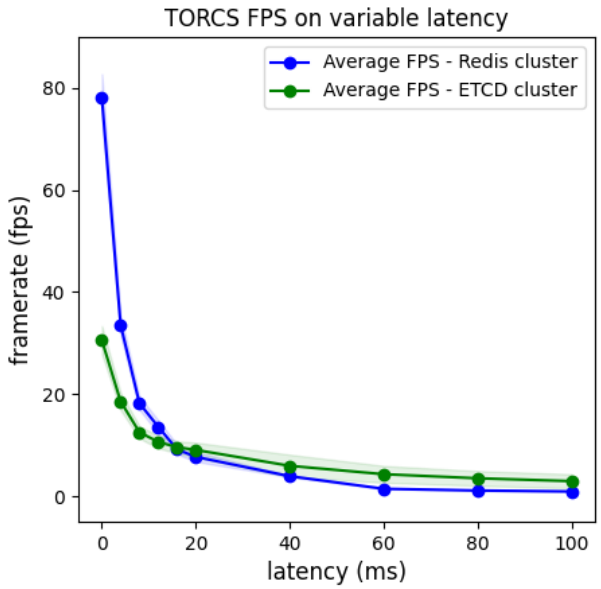


Figure 7.26: TORCS main display framerate comparison - cluster.

scalability in the number of cluster nodes.
This behaviour can be seen in Figure 7.26 where, if compared with Figure 7.24, the amount of latency required to have the Redis performance become worse than ETCD is much larger ($\sim$ 16 ms against the previous $\sim$ 4 ms). This is an interesting result, which highlights a trade-off between the benefits provided by the ETCD implementation of HTTP_G pipelining and the benefits provided by Redis Eventual Consistency replication mechanism.

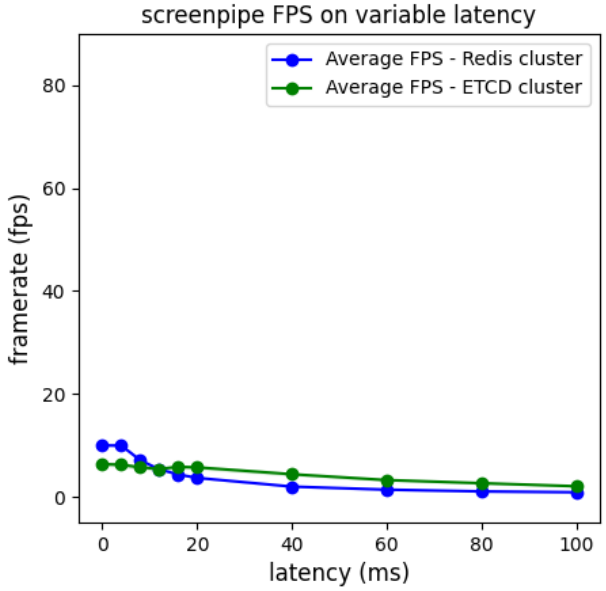


Figure 7.27: Remote display framerate comparison - cluster.

7.2.4 Distribution of dynamic game state data

We present the results obtained by introducing an increasing number of game state related fields into the distributed storage, while also introducing an increasing amount of network latency into each configuration.

ETCD stand-alone configuration

For the system configuration with 0 game state fields stored we measure an average framerate of 77.170 fps, which is not influenced by any network latency introduced into the distributed database, as there is no communication between TORCS and the storage.

In this particular configuration, we only consider write requests as they were the only sampled value at this point of the development. Still, they are meaningful enough to provide us with an indication of the trend in the processing of the requests by the system, in addition as being an element of comparison with other configurations.

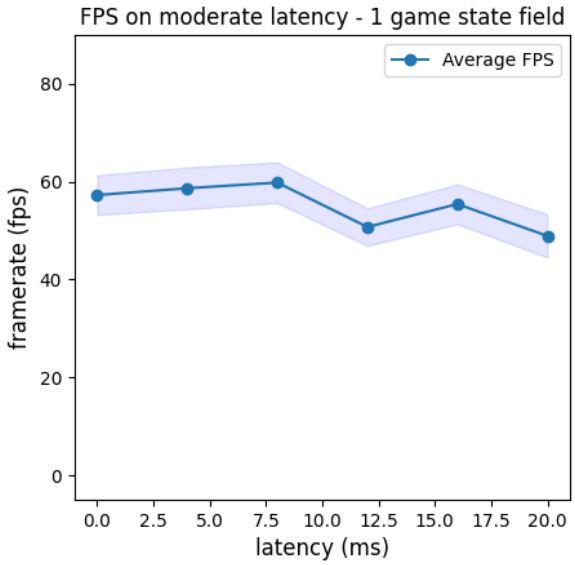


Figure 7.28: Graphics framerate - ETCD stand-alone - 1 game state field.

As we can see from Figure 7.28, managing 1 game state field (`deltaTime`) using the distributed database influences the graphics framerate, decreasing it from about 77 fps to about 60 fps. While the decrease is not monotonic, we can notice that the value tends to become lower as the network latency increases.

In terms of processed requests, we can see in Figure 7.29 that the behaviour is generally similar to the framerate, with a starting point slightly above 50 req/s and a slow decrease to below 50 req/s, when the 20 ms latency mark is reached.

As we can see from Figure 7.30, managing 2 game state fields (`deltaTime`, `currentTime`) using the distributed database, heavily influences the graphics framerate, decreasing it up to about 10 fps even when no latency is introduced.

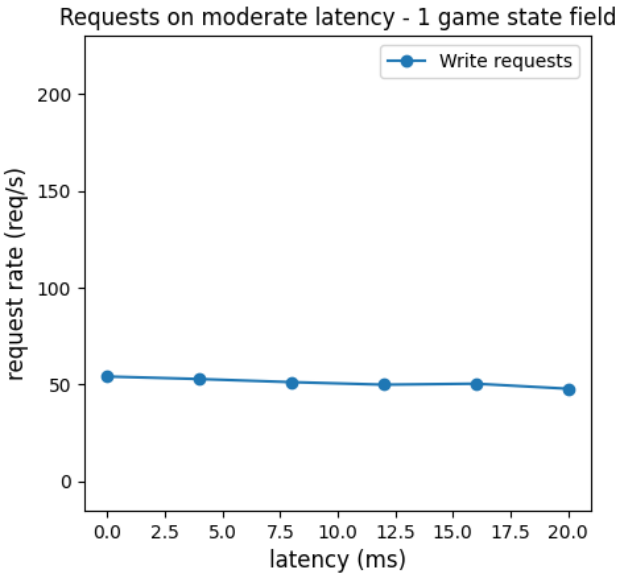


Figure 7.29: Requests processed - ETCD stand-alone - 1 game state field.

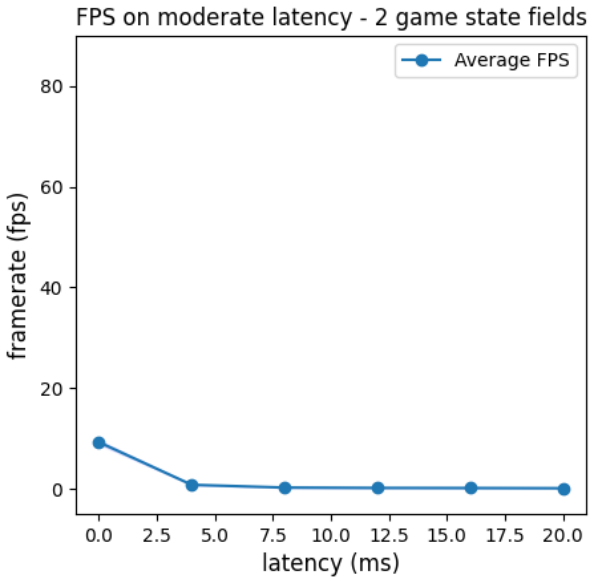


Figure 7.30: Graphics framerate - ETCD stand-alone - 2 game state fields.

The introduction of network latency in this context further affects the system performance, decreasing the framerate up to near 0 fps, thus making the game completely unplayable. This particularly heavy effect on the performance is likely tied also to the characteristics of the `currentTime` game state field we introduced. Considering that this field is both written and read a multitude of times during code execution.

In terms of processed requests, we can see in Figure 7.31 that the request rate follows a trend which is similar to the framerate, with an initially high value, in a situation

with no latency. However, this value sharply drops after even low amounts of network latency are introduced, proceeding then to decrease gradually as latency increases.

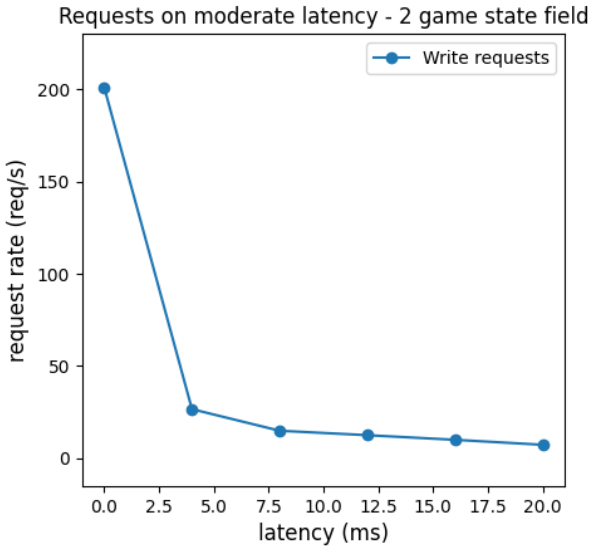


Figure 7.31: Requests processed - ETCD stand-alone - 2 game state fields.

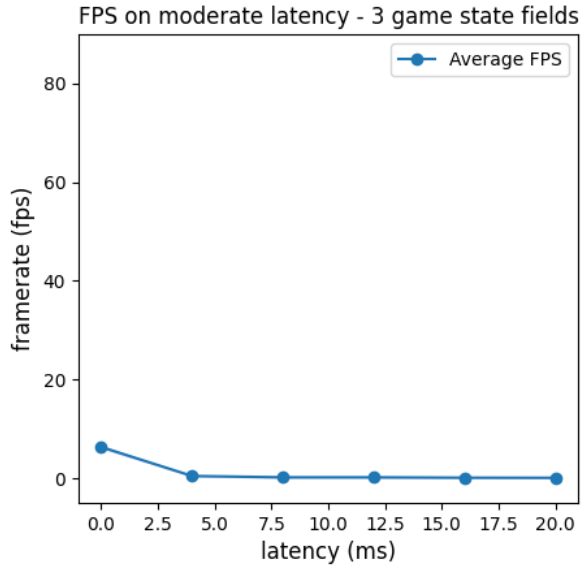


Figure 7.32: Graphics framerate - ETCD stand-alone - 3 game state fields.

As we can see from Figure 7.32, managing 3 game state fields (`deltaTime`, `currentTime`, `raceState`) using the distributed database, heavily influences the graphics framerate, decreasing it up to less than 10 fps even when no latency is introduced.

The introduction of network latency in this context further affects the system perfor-

mance, decreasing the framerate up to near 0 fps, thus making the game completely unplayable. This behaviour is similar to what we experimented in the previous configuration, for both the framerate and for the request rate, as we can see in Figure 7.33, with similar root causes. Finally, we if compare the results of the multiple measurements

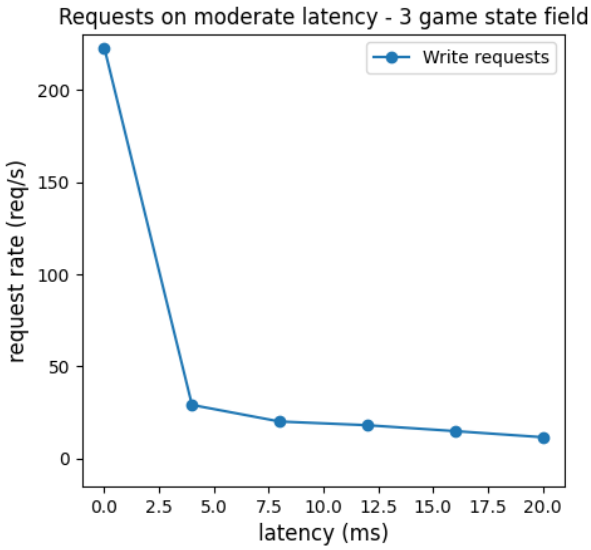


Figure 7.33: Requests processed - ETCD stand-alone - 3 game state fields.

we just discussed, as presented in Figure 7.34, we can see that the increasing amount of game state fields stored into ETCD greatly impacts its performance in a negative way. Additionally, the introduction of network latency also has a significant impact on performance, making the game completely unplayable in the context where 2-3 game state are being managed with this approach.

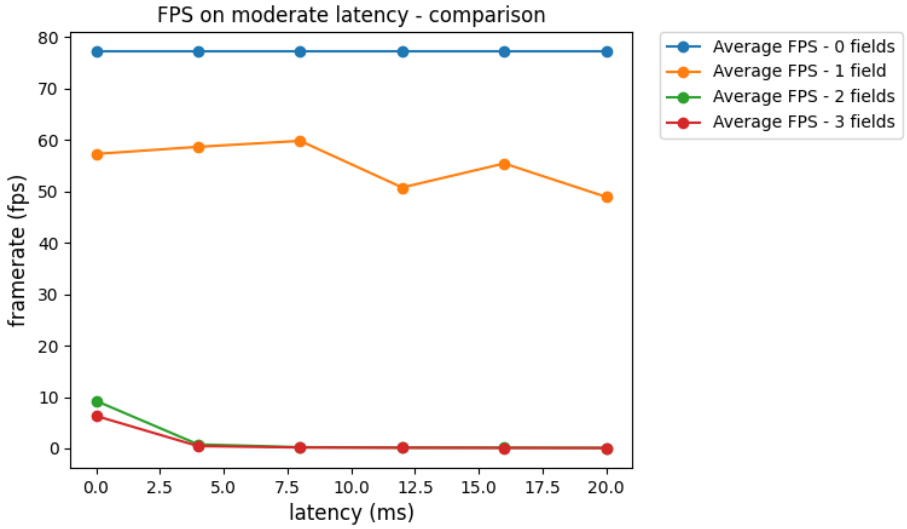


Figure 7.34: Graphics framerate comparison - ETCD stand-alone.

As such, we conclude that using ETCD for this type of approach is not a feasible solution, as the impact on system performance of the delay introduced by the distributed DB_G processing of requests, in addition to network latency, is excessively heavy.

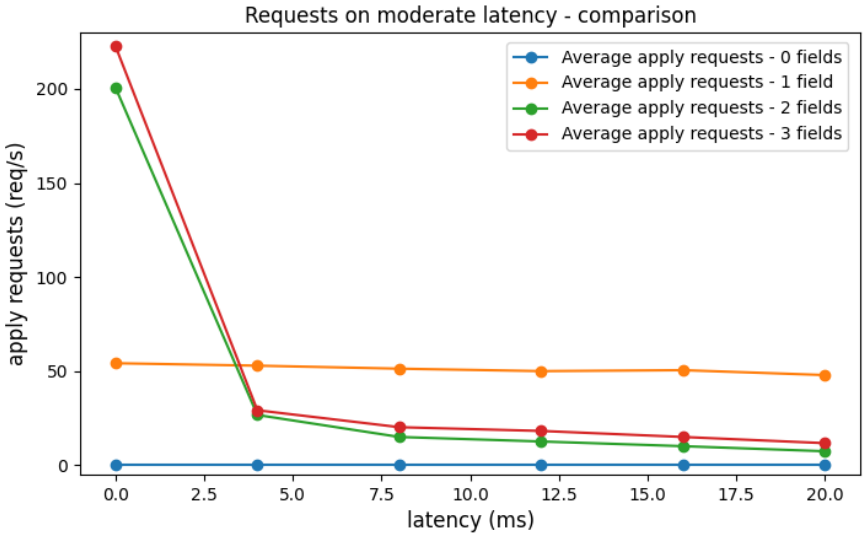


Figure 7.35: Requests processed comparison - ETCD stand-alone.

Redis stand-alone configuration

For the system configuration with 0 game state fields stored we measure an average framerate of 83.789 fps, which is not influenced by any network latency introduced into the distributed database, as there is no communication between TORCS and the storage. In this configuration, we consider both write and read requests rates.
As we can see from Figure 7.36, managing 1 game state field (`deltaTime`) using the distributed database does not significantly influences the graphics framerate, when no network latency is introduced. However, when even low amounts of latency are introduced, the graphics framerate rapidly drops, decreasing it from about 80 fps to about 20 fps as latency increases. This decrease is gradual and monotonic, highlighting weaker performance in the context of slow connections.

In terms of processed requests, we can see in Figure 7.37 that the behaviour is generally similar to what we experienced on ETCD, with the request rate slowly decreasing as more latency is introduced.

As we can see from Figure 7.38, managing 2 game state fields (`deltaTime`, `currentTime`) using the distributed database, heavily influences the graphics framerate, decreasing it up to about 60 fps even when no latency is introduced. Still, considering the characteristics of the `currentTime` game state field we introduced, the system performance with this configuration a relatively good.

However, the introduction of network latency in this context greatly affects the system performance, decreasing the framerate up to near 0 fps, thus making the game completely unplayable.

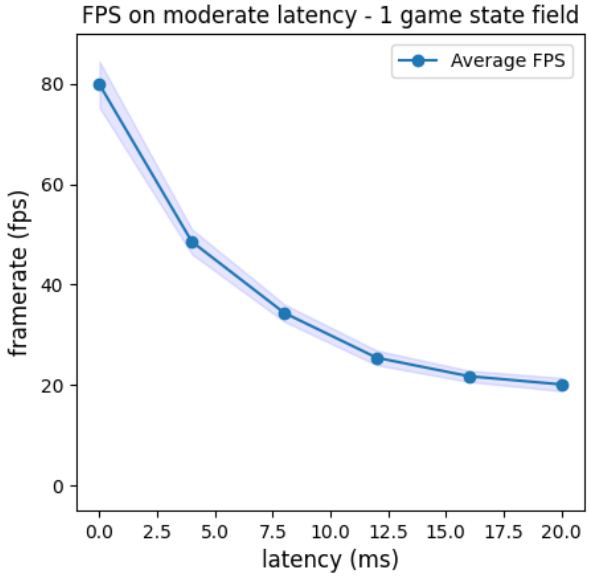


Figure 7.36: Graphics framerate - Redis stand-alone - 1 game state field.

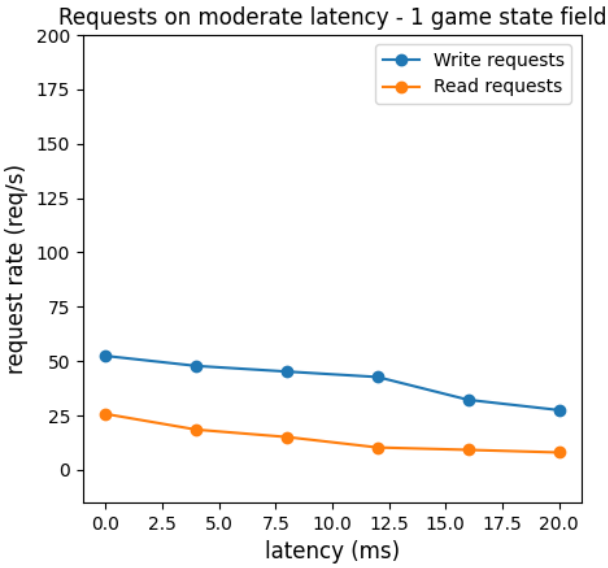


Figure 7.37: Requests processed - Redis stand-alone - 1 game state field.

In terms of processed requests, we can see in Figure 7.39 that the request rate follows a trend which is similar to the framerate, with an initially high value, in a situation with no latency. However, this value sharply drops after even low amounts of network latency are introduced, proceeding then to decrease gradually as latency increases.

As we can see from Figure 7.40, managing 3 game state fields (`deltaTime`, `currentTime`, `raceState`) using the distributed database, heavily influences the graphics framerate,

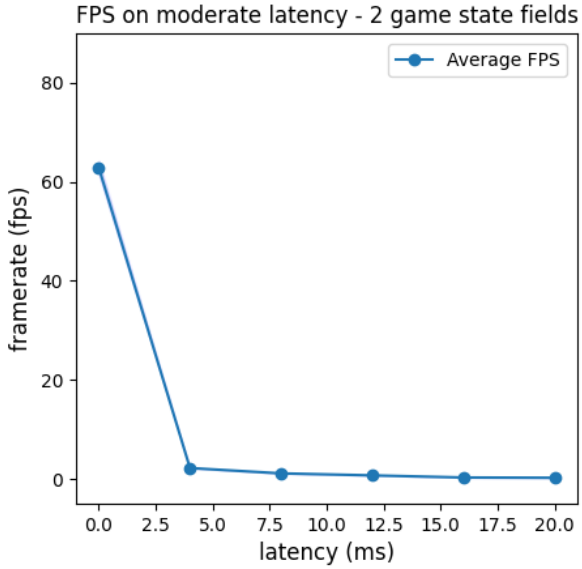


Figure 7.38: Graphics framerate - Redis stand-alone - 2 game state field.

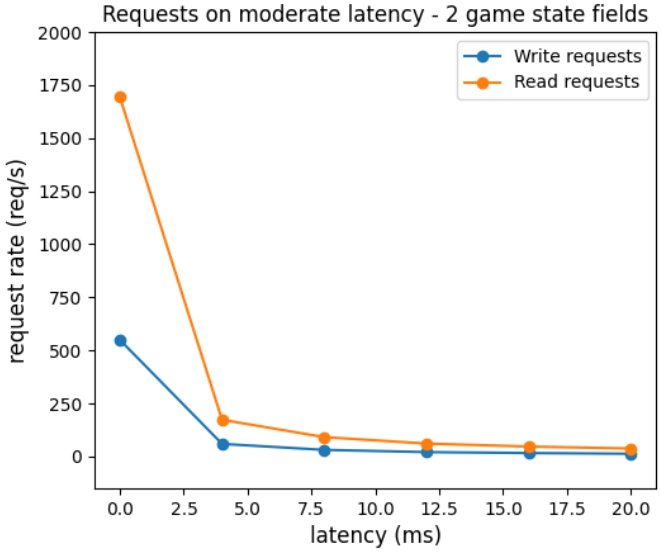


Figure 7.39: Requests processed - Redis stand-alone - 2 game state field.

decreasing it up to less than 60 fps even when no latency is introduced. Still, considering the characteristics of the `currentTime` game state field we previously introduced, the system performance with this configuration a relatively good.

The introduction of network latency in this context further affects the system performance, decreasing the framerate up to near 0 fps, thus making the game completely unplayable. This behaviour is similar to what we experimented in the previous configuration, for both the framerate and for the request rate, as we can see in Figure 7.41, with similar root causes.

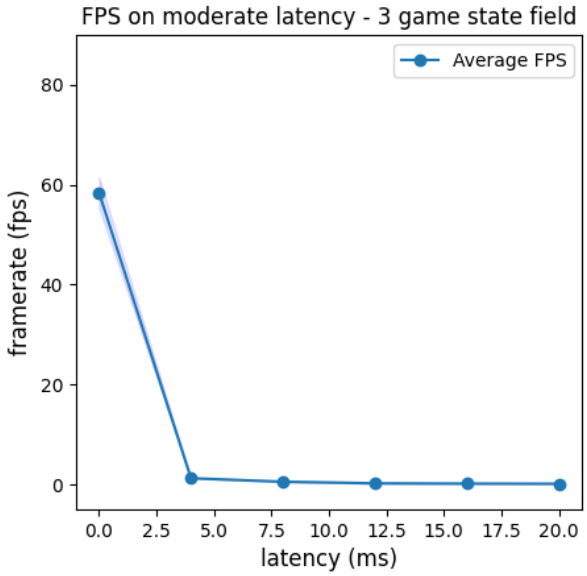


Figure 7.40: Graphics framerate - Redis stand-alone - 3 game state fields.

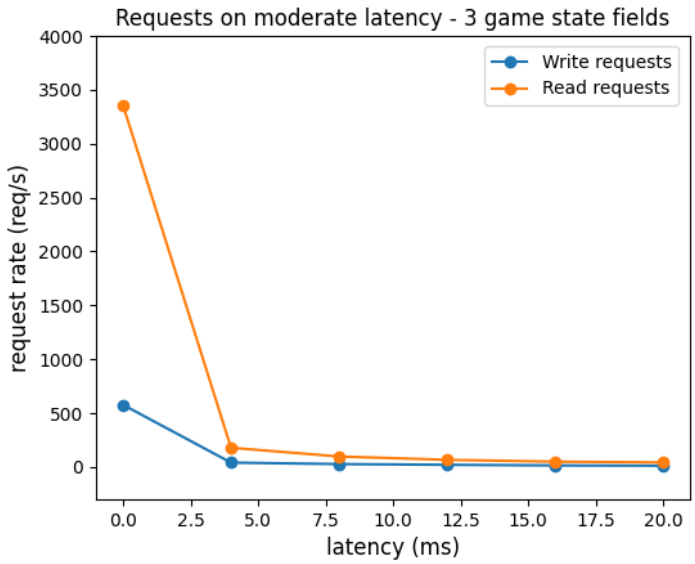


Figure 7.41: Requests processed - Redis stand-alone - 3 game state fields.

Finally, if we compare the results of the multiple measurements we just discussed, as presented in Figure 7.42, we can see that the increasing amount of game state fields stored into Redis greatly impacts its performance in a negative way.
Additionally, the introduction of network latency also has a significant impact on performance, making the game completely unplayable in the context where 2-3 game state are being managed with this approach.

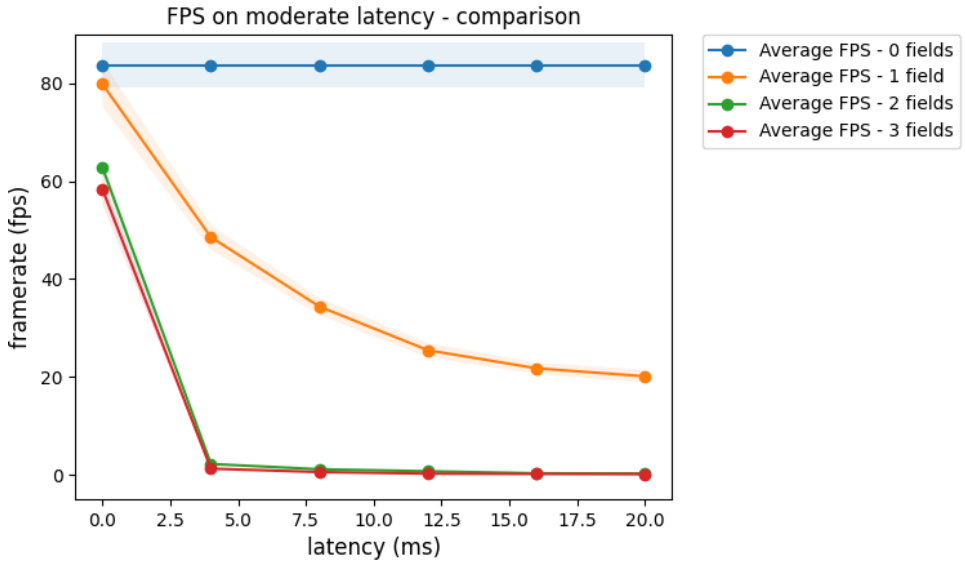


Figure 7.42: Graphics framerate comparison - Redis stand-alone.

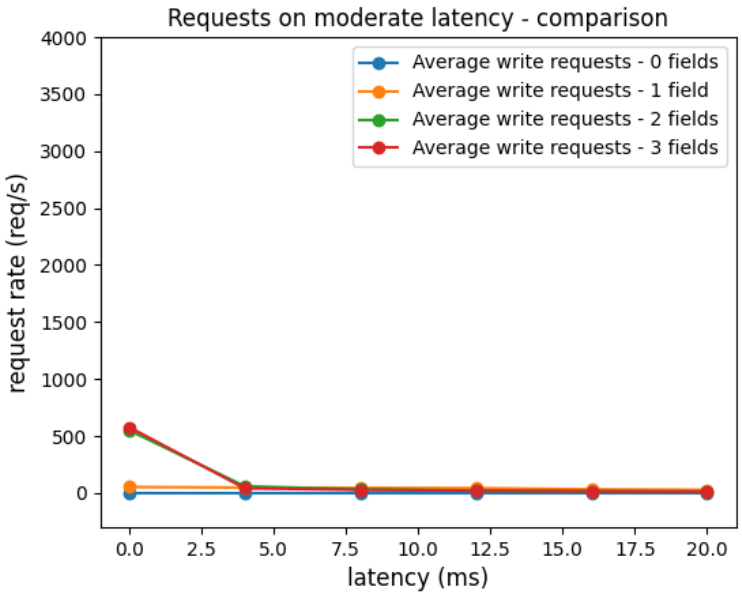


Figure 7.43: Write requests processed comparison - Redis stand-alone.

As such, we conclude that using Redis for this type of approach is not a feasible solution, as the impact on system performance of the delay introduced by the distributed DB_G processing of requests, in addition to network latency, is excessively heavy.

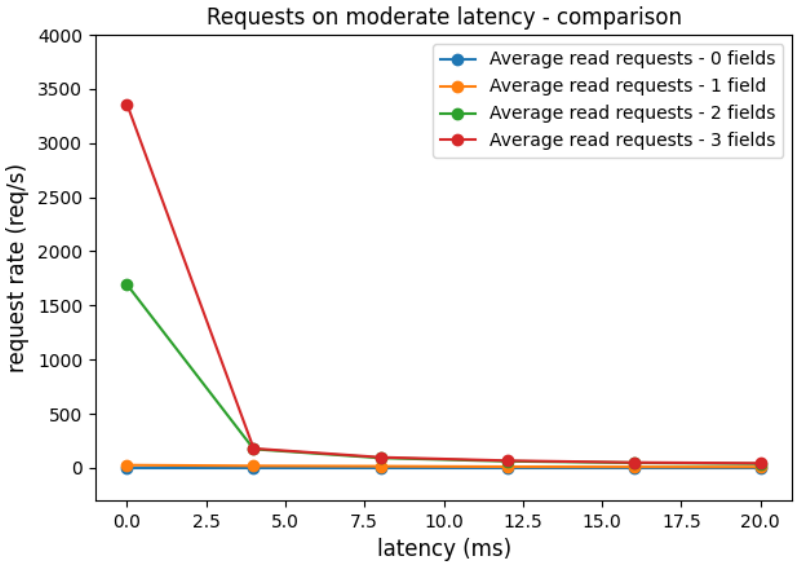


Figure 7.44: Read requests processed comparison - Redis stand-alone.

ETCD 3-members cluster configuration

For the system configuration with 0 game state fields stored we measure an average framerate of 77.170 fps, which is not influenced by any network latency introduced into the distributed database, as there is no communication between TORCS and the storage. In this configuration, we consider both write and read requests rates.

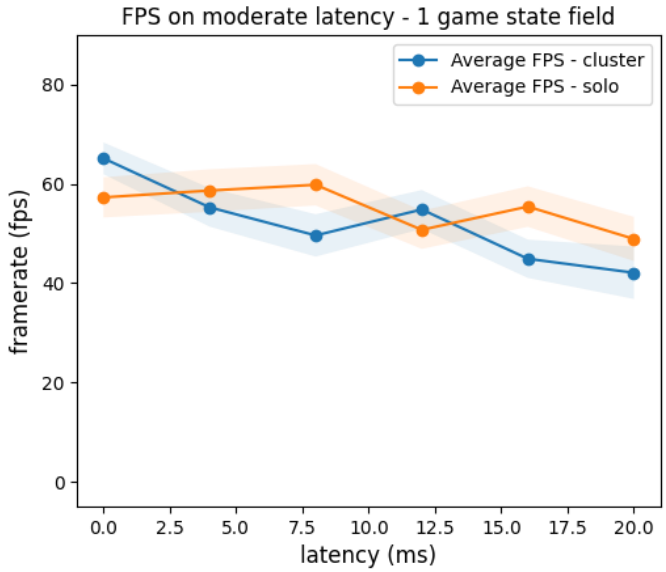


Figure 7.45: Graphics framerate - ETCD cluster - 1 game state field.

As we can see from Figure 7.45, managing 1 game state field (`deltaTime`) using the

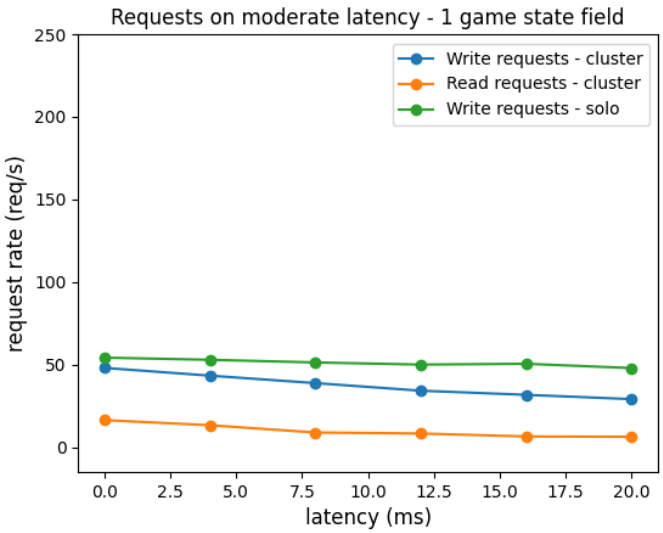


Figure 7.46: Requests processed - ETCD cluster - 1 game state field.

distributed database influences the graphics framerate, decreasing it from about 65 fps to about 55 fps. While the decrease is not monotonic, we can notice that the value tends to become lower as the network latency increases. This behaviour is similar to what we experienced in the stand-alone configuration, as we can see from the crossing confidence intervals.

In terms of processed requests, we can see in Figure 7.46 that the behaviour is generally similar to the framerate, with a starting point slightly above 50 req/s and a slow decrease to below 50 req/s, when the 20 ms latency mark is reached. If we compare the write requests with the values obtained for the stand-alone ETCD version, the cluster version performs generally worse, as expected considering the presence of additional replication operations that may slow down the system.

As we can see from Figure 7.47, managing 2 game state fields (`deltaTime`, `currentTime`) using the distributed database, heavily influences the graphics framerate, decreasing it up to about 5 fps even when no latency is introduced. This behaviour is similar to what we experienced in the stand-alone configuration, with generally lower values in the cluster version.

The introduction of network latency in this context further affects the system performance, decreasing the framerate up to near 0 fps, thus making the game completely unplayable. This particularly heavy effect on the performance is likely tied also to the characteristics of the `currentTime` game state field we introduced. Considering that this field is both written and read a multitude of times during code execution.

In terms of processed requests, we can see in Figure 7.48 that the request rate follows a trend which is similar to the framerate, with an initially high value, in a situation with no latency. However, this value sharply drops after even low amounts of network latency are introduced, proceeding then to decrease gradually as latency increases. If we compare the write requests with the values obtained for the stand-alone ETCD

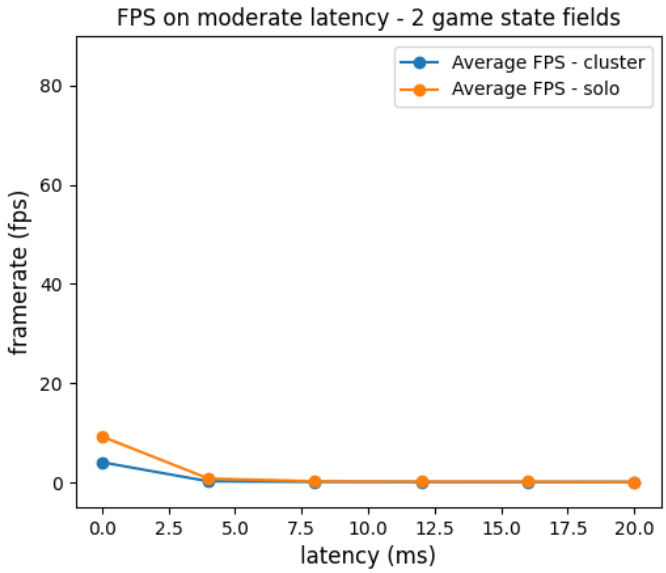


Figure 7.47: Graphics framerate - ETCD cluster - 2 game state fields.

version, the cluster version performs generally worse, for the same reason we previously discussed.

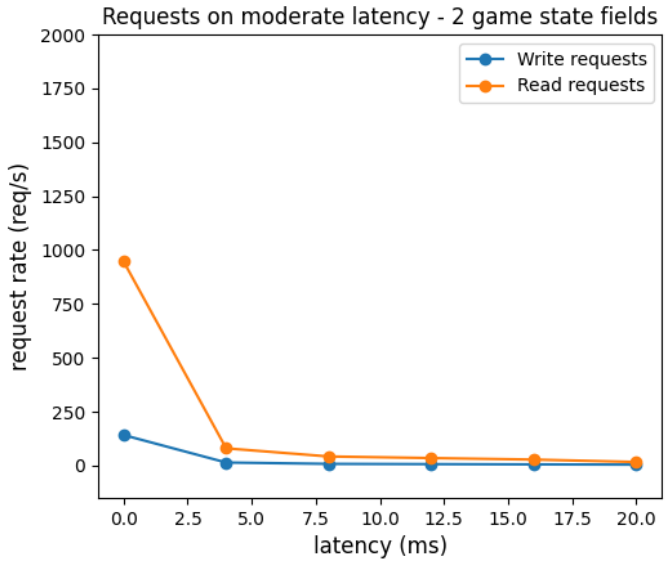


Figure 7.48: Requests processed - ETCD cluster - 2 game state fields.

As we can see from Figure 7.49, managing 3 game state fields (`deltaTime`, `currentTime`, `raceState`) using the distributed database, heavily influences the graphics framerate, decreasing it up to less than 5 fps even when no latency is introduced. This behaviour is similar to what we experienced in the stand-alone configuration, with generally lower values in the cluster version.

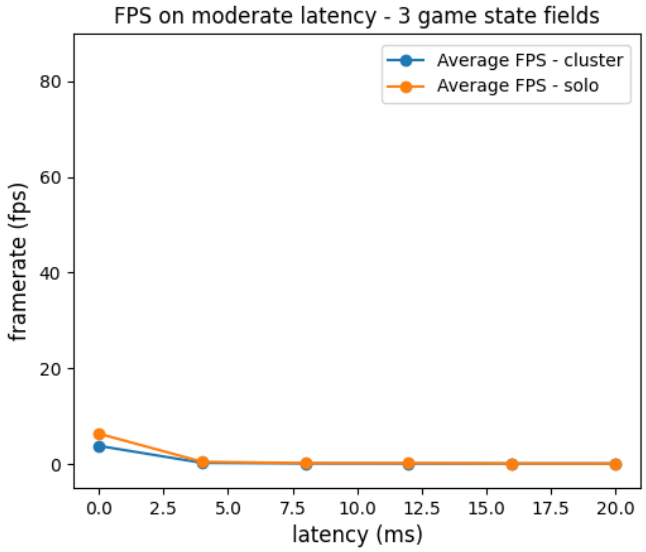


Figure 7.49: Graphics framerate - ETCD cluster - 3 game state fields.

The introduction of network latency in this context further affects the system performance, decreasing the framerate up to near 0 fps, thus making the game completely unplayable. This behaviour is similar to what we experimented in the previous configuration, for both the framerate and for the request rate, as we can see in Figure 7.50, with similar root causes. If we compare the write requests with the values obtained for the stand-alone ETCD version, the cluster version performs generally worse, for the same reason we previously discussed.

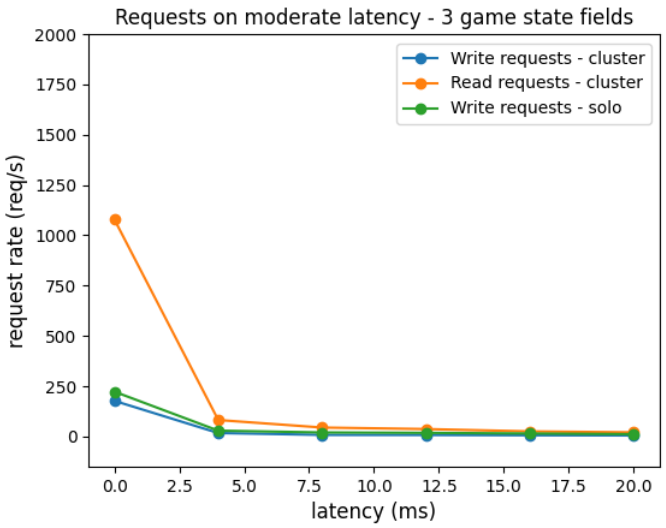


Figure 7.50: Requests processed - ETCD cluster - 3 game state fields.

Finally, we if compare the results of the multiple measurements we just discussed, as presented in Figure 7.51, we can see that the increasing amount of game state fields stored into ETCD greatly impacts its performance in a negative way. Additionally, the introduction of network latency also has a significant impact on performance, making the game completely unplayable in the context where 2-3 game state are being managed with this approach.

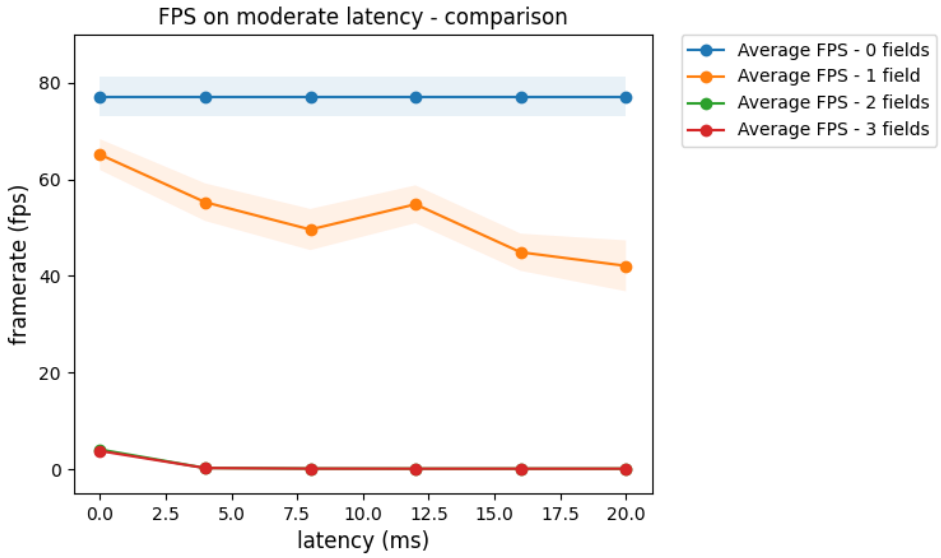


Figure 7.51: Graphics framerate comparison - ETCD cluster.

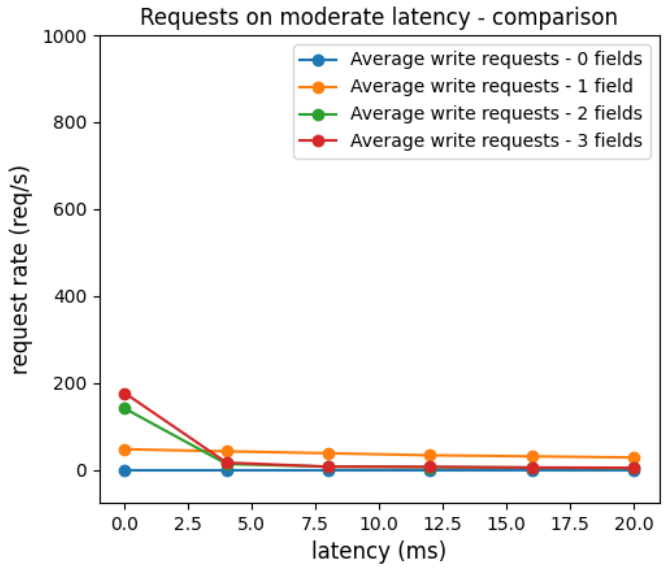


Figure 7.52: Write requests processed comparison - ETCD cluster.

As such, we conclude that using ETCD, in its cluster version, for this type of approach

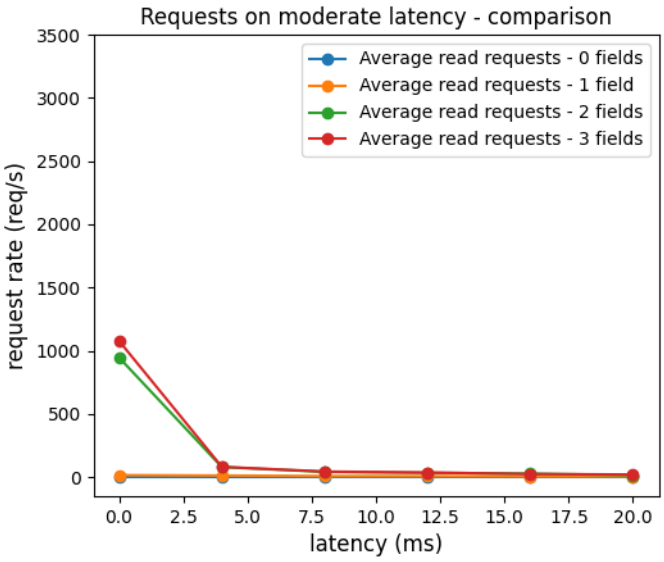


Figure 7.53: Read requests processed comparison - ETCD cluster.

is not a feasible solution, as the impact on system performance of the delay introduced by the distributed DB_G processing of requests, in addition to network latency, is excessively heavy.

Redis 3-members cluster configuration

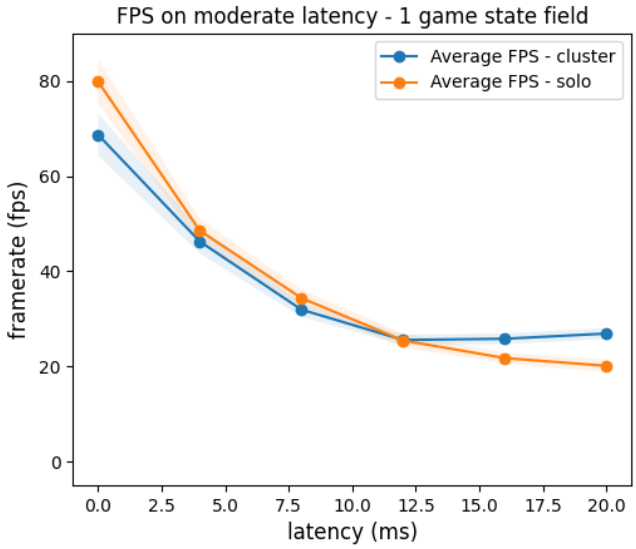


Figure 7.54: Graphics framerate - Redis cluster - 1 game state field.

For the system configuration with 0 game state fields stored we measure an average framerate of 77.170 fps, which is not influenced by any network latency introduced into the distributed database, as there is no communication between TORCS and the

storage. In this configuration, we consider both write and read requests rates.

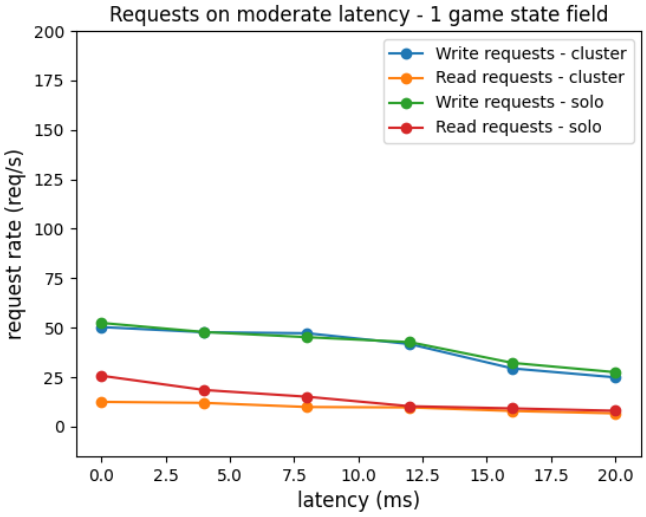


Figure 7.55: Requests processed - Redis cluster - 1 game state field.

As we can see from Figure 7.54, managing 1 game state field (`deltaTime`) using the distributed database does not significantly influences the graphics framerate, when no network latency is introduced. However, when even low amounts of latency are introduced, the graphics framerate rapidly drops, decreasing it from about 80 fps to about 20 fps as latency increases. This decrease is gradual and monotonic, highlighting weaker performance in the context of slow connections. This behaviour is similar to what we experienced in the stand-alone configuration, as we can see from the crossing confidence intervals.

In terms of processed requests, we can see in Figure 7.55 that the behaviour is generally similar to what we experienced on ETCD, with the request rate slowly decreasing as more latency is introduced. If we compare the write requests with the values obtained for the stand-alone ETCD version, the cluster version performs similarly, as expected considering the presence of an Eventual Consistency mechanism that guarantees good scalability.

As we can see from Figure 7.56, managing 2 game state fields (`deltaTime`, `currentTime`) using the distributed database, heavily influences the graphics framerate, decreasing it up to about 60 fps even when no latency is introduced. Still, considering the characteristics of the `currentTime` game state field we introduced, the system performance with this configuration a relatively good.

However, the introduction of network latency in this context greatly affects the system performance, decreasing the framerate up to near 0 fps, thus making the game completely unplayable. This behaviour is exactly the same as what we experienced in the stand-alone configuration, as we can see from the crossing confidence intervals.

In terms of processed requests, we can see in Figure 7.57 that the request rate

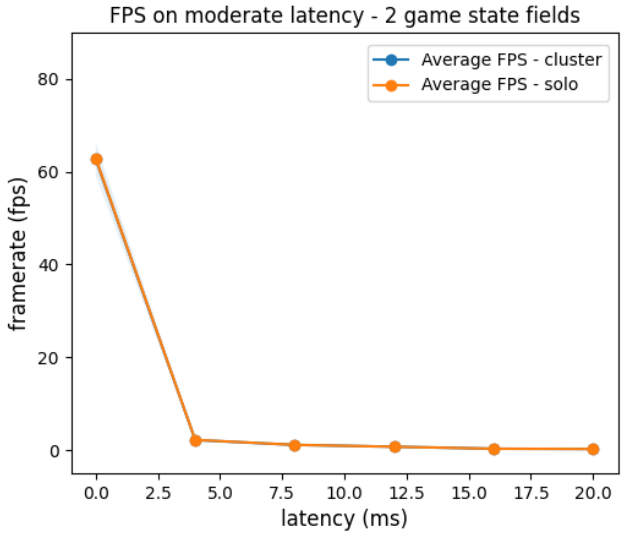


Figure 7.56: Graphics framerate - Redis cluster - 2 game state field.

follows a trend which is similar to the framerate, with an initially high value, in a situation with no latency. However, this value sharply drops after even low amounts of network latency are introduced, proceeding then to decrease gradually as latency increases. If we compare the write requests with the values obtained for the stand-alone Redis version, the cluster version performs exactly the same, for the same reason we previously discussed.

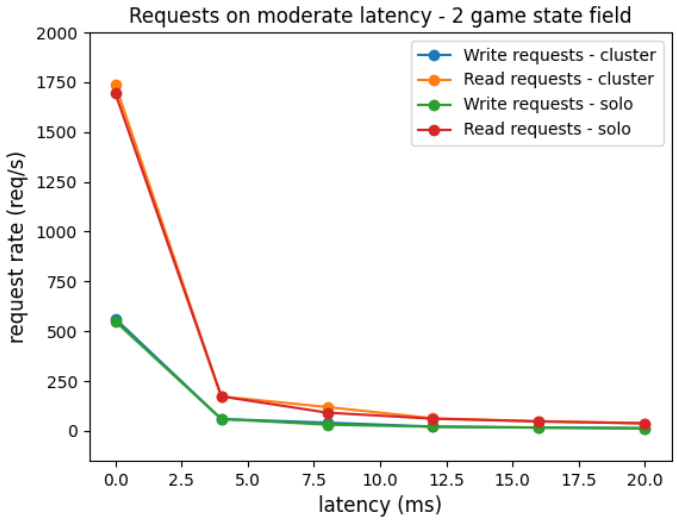


Figure 7.57: Requests processed - Redis cluster - 2 game state field.

As we can see from Figure 7.58, managing 3 game state fields (`deltaTime`, `currentTime`, `raceState`) using the distributed database, heavily influences the graphics framerate, decreasing it up to less than 60 fps even when no latency is introduced. Still, considering

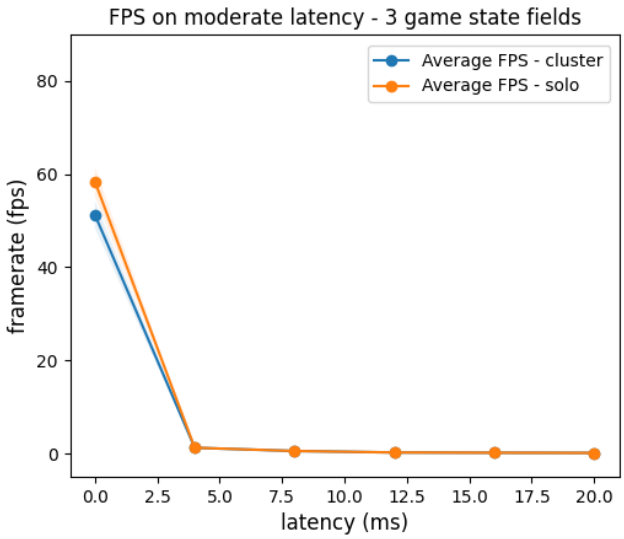


Figure 7.58: Graphics framerate - Redis cluster - 3 game state fields.

the characteristics of the `currentTime` game state field we previously introduced, the system performance with this configuration is relatively good.

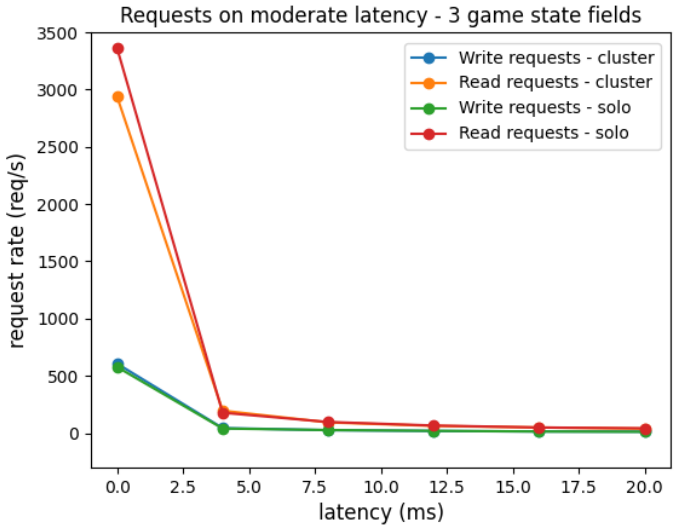


Figure 7.59: Requests processed - Redis stand-alone - 3 game state fields.

The introduction of network latency in this context further affects the system performance, decreasing the framerate up to near 0 fps, thus making the game completely unplayable. This behaviour is similar to what we experimented in the previous configuration, for both the framerate and for the request rate, as we can see in Figure 7.59, with similar root causes. Moreover, this behaviour is also exactly the same as what we experienced in the stand-alone configuration, as we can see from the crossing confidence intervals and the average request rate values.

Finally, if we compare the results of the multiple measurements we just discussed, as presented in Figure 7.60, we can see that the increasing amount of game state fields stored into Redis greatly impacts its performance in a negative way.
Additionally, the introduction of network latency also has a significant impact on performance, making the game completely unplayable in the context where 2-3 game state are being managed with this approach.

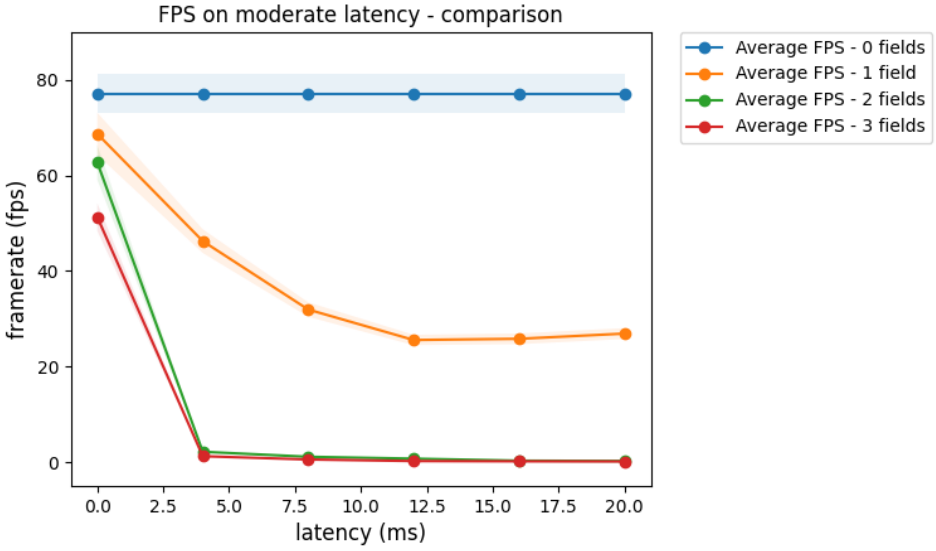


Figure 7.60: Graphics framerate comparison - Redis cluster.

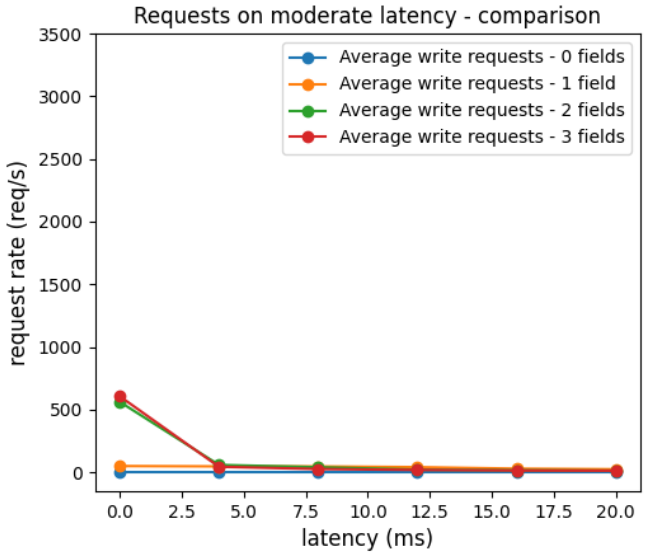


Figure 7.61: Write requests processed comparison - Redis cluster.

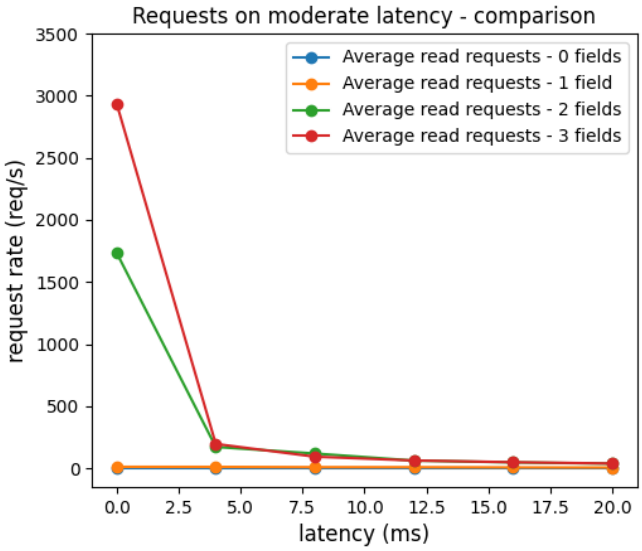


Figure 7.62: Read requests processed comparison - Redis cluster

As such, we conclude that using Redis, in its cluster configuration, is not a feasible solution for this type of approach, as the impact on system performance of the delay introduced by the distributed DB_G processing of requests, in addition to network latency, is excessively heavy.

Final system stand-alone comparisons

We now proceed to compare, with each other, the results obtained in the ETCD and Redis stand-alone configurations, for each number of game state fields introduced into the storage.

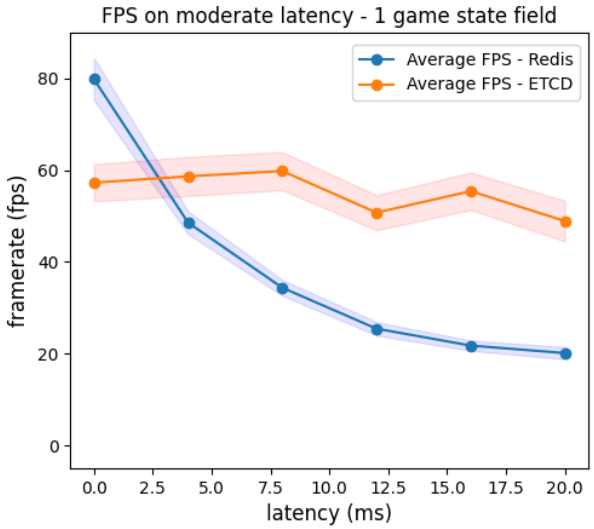


Figure 7.63: Graphics framerate comparison - 1 game state field - stand-alone.

As we can see in Figure 7.63, managing 1 game state field (`deltaTime`), Redis provides a better graphics framerate in situations with no network latency. However, in when network latency is introduced, ETCD immediately performs better than Redis, regardless of the amount of delay. This is likely due to ETCD implementation of HTTP_G pipelining, which allow it to mitigate the effects of network latency on its data communication.
This characteristic is also similarly reflected in the write request rate, where ETCD always perform better than Redis in situations where latency is introduced, as we can see from Figure 7.64.

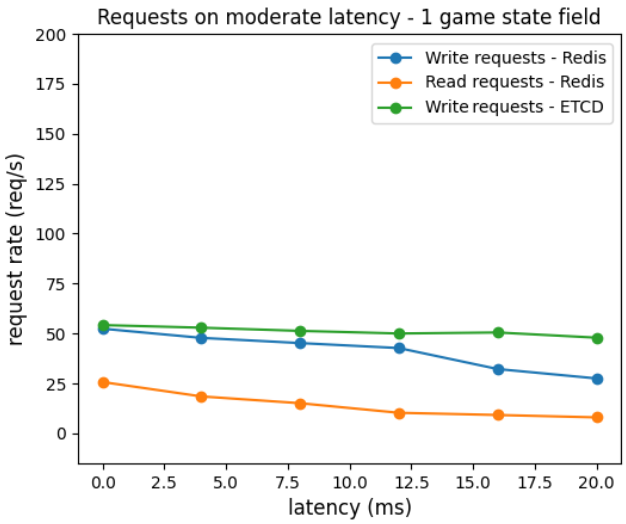


Figure 7.64: Requests processed comparison - 1 game state field - stand-alone.

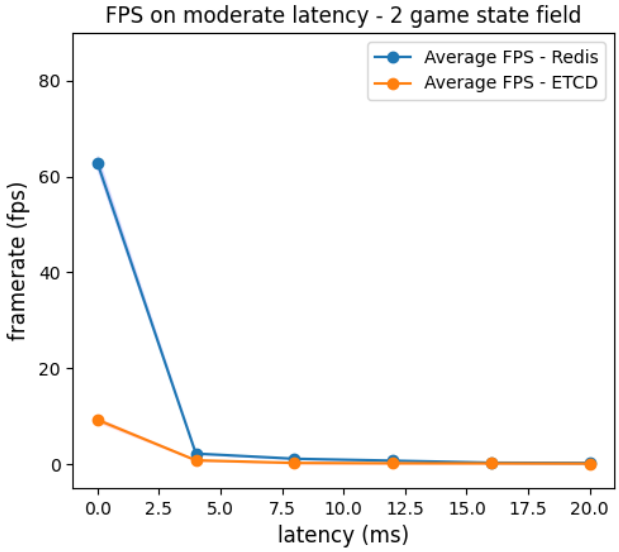


Figure 7.65: Graphics framerate comparison - 2 game state fields - stand-alone.

As we can see in Figure 7.65, managing 2 game state field (`deltaTime`, `currentTime`), Redis provides a better graphics framerate in situations with little to no network latency. However, in when even low network latency is introduced, ETCD tends to reach the same level of Redis in terms of performance, for the same reason we previously discussed.
This characteristic is also similarly reflected in the write request rate, where Redis performs better than ETCD in situations where little to no latency is introduced, as we can see from Figure 7.66.

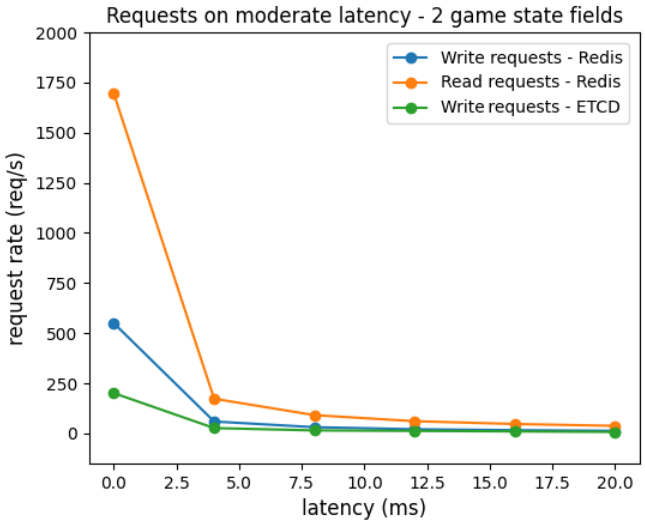


Figure 7.66: Requests processed comparison - 2 game state fields - stand-alone.

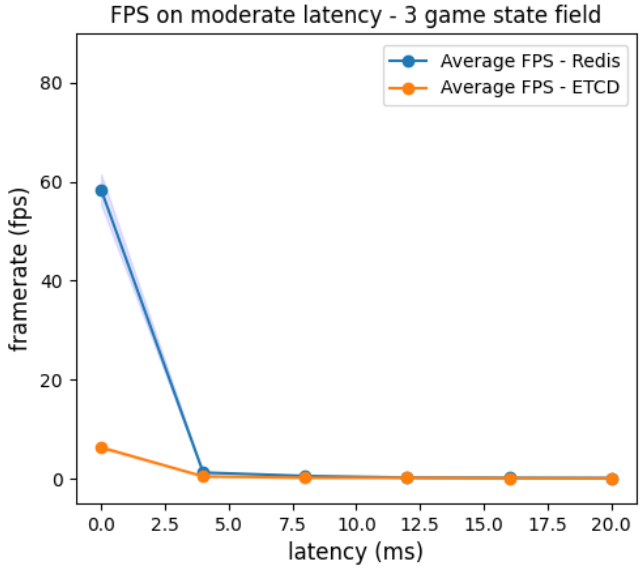


Figure 7.67: Graphics framerate comparison - 3 game state fields - stand-alone.

As we can see in Figure 7.67, managing 3 game state field (`deltaTime`, `currentTime`, `raceState`), Redis provides a better graphics framerate in situations with little to no network latency. However, in when even low network latency is introduced, ETCD tends to reach the same level of Redis in terms of performance, for the same reason we previously discussed.
This characteristic is also similarly reflected in the write request rate, where Redis performs better than ETCD in situations where little to no latency is introduced, as we can see from Figure 7.68.

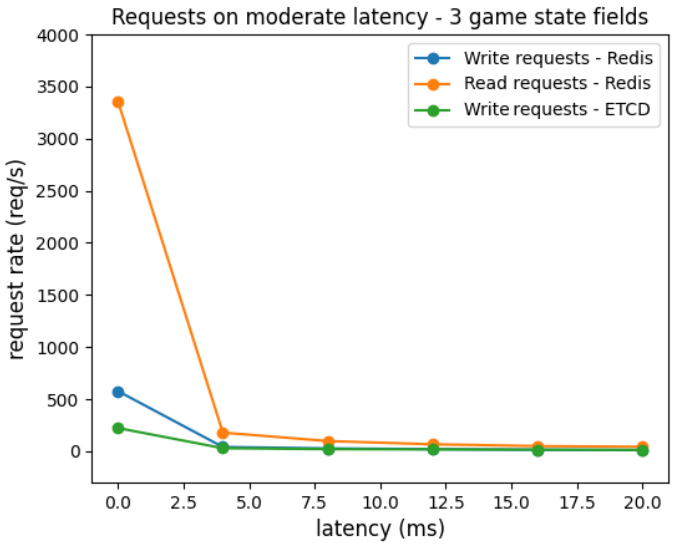


Figure 7.68: Requests processed comparison - 3 game state fields - stand-alone.

Considering these results, we can assert that, in the context of this specific approach, Redis performs better than ETCD in their respective stand-alone versions. Redis provides generally better performance than ETCD in situations with no latency introduced, whereas, when network latency is present, Redis still performs on par with ETCD, despite its HTTP_G pipelining implementation.

Still, neither of these two systems provide performance suitable for this type of approach, mainly due to shortcomings in the approach itself. The extremely large number of requests required to be satisfied in order to allow it the approach to succeed is excessively large for distributed databases. Moreover, the frequency of game state updates further exacerbates this problem. As such, we considered alternative solutions such as the State Manager middleware described in section 5.7.

Final system 3-members cluster comparisons

We now proceed to compare, with each other, the results obtained in the ETCD and Redis 3-members cluster configurations, for each number of game state fields introduced into the storage.

As we can see in Figure 7.69, managing 1 game state field (`deltaTime`), Redis provides a better graphics framerate in situations with no network latency. However, in when network latency is introduced, ETCD immediately performs better than Redis,

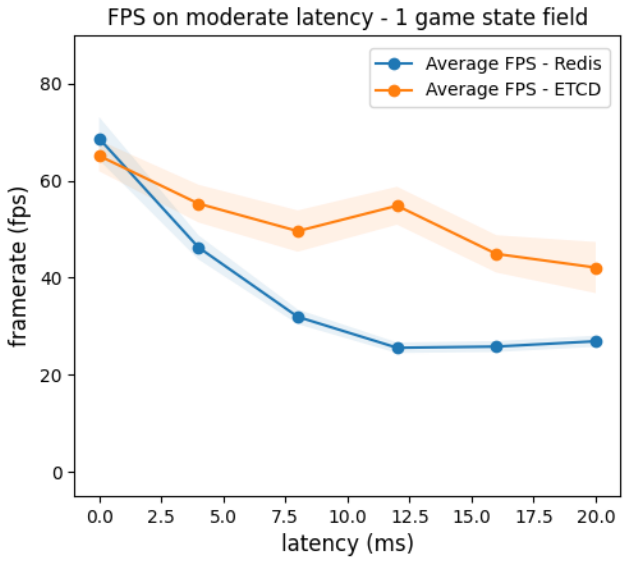


Figure 7.69: Graphics framerate comparison - 1 game state field - cluster.

regardless of the amount of delay. This is likely due to ETCD implementation of HTTP_G pipelining, which allow it to mitigate the effects of network latency on its data communication.
This characteristic is also similarly reflected in the write request rate, where ETCD performs about the same as Redis in situations both is situations with and without network latency, as we can see from Figure 7.70.

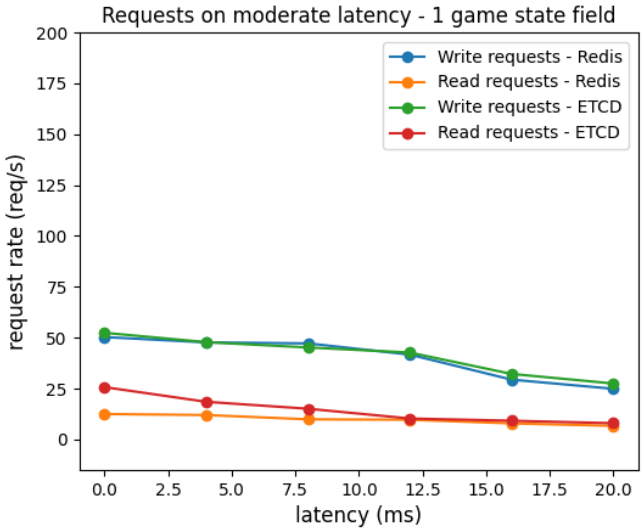


Figure 7.70: Requests processed comparison - 1 game state field - cluster.

As we can see in Figure 7.71, managing 2 game state field (`deltaTime`, `currentTime`), Redis provides a better graphics framerate in situations with little to no network

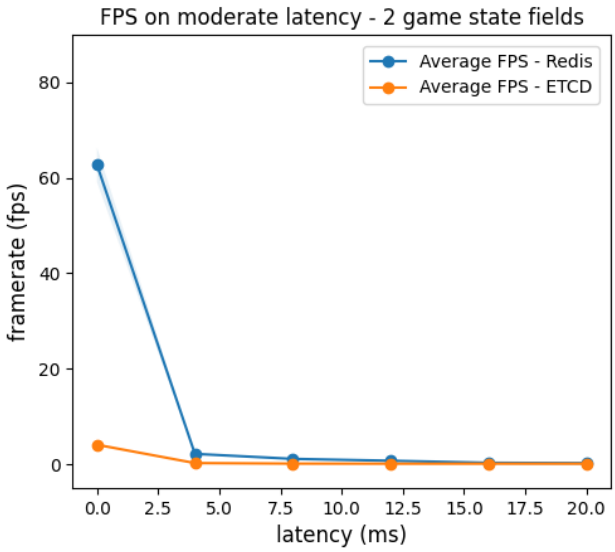


Figure 7.71: Graphics framerate comparison - 2 game state fields - cluster.

latency. However, in when even low network latency is introduced, ETCD tends to reach the same level of Redis in terms of performance, for the same reason we previously discussed. Still, if compared with the situation in the stand-alone configuration (Figure 7.65) Redis manages to provide better performance than ETCD up to an higher value of network latency than before. This is due to the benefits provided by Redis Eventual Consistency mechanism, which should provide generally better performance than ETCD Strong Consistency mechanism, is cluster configurations.
This characteristic is also similarly reflected in the request rate, where Redis generally performs better than ETCD, regardless of the amount of latency, as we can see from Figure 7.72.

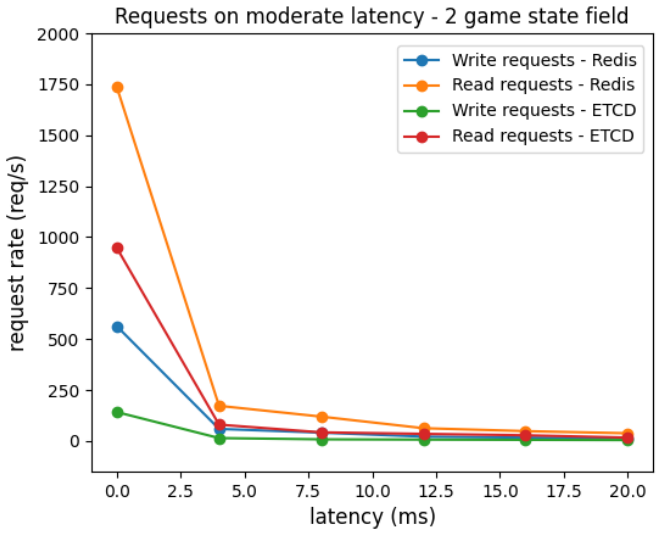


Figure 7.72: Requests processed comparison - 2 game state fields - cluster.

As we can see in Figure 7.73, managing 3 game state field (`deltaTime`, `currentTime`, `raceState`), Redis provides a better graphics framerate in situations with little to no network latency. However, in when even low network latency is introduced, ETCD tends to reach the same level of Redis in terms of performance, for the same reason we previously discussed. The benefits of the Redis replication mechanism are also still present, as previously discussed.
This characteristic is also similarly reflected in the request rate, where Redis generally performs better than ETCD, regardless of the amount of latency, as we can see from Figure 7.74.

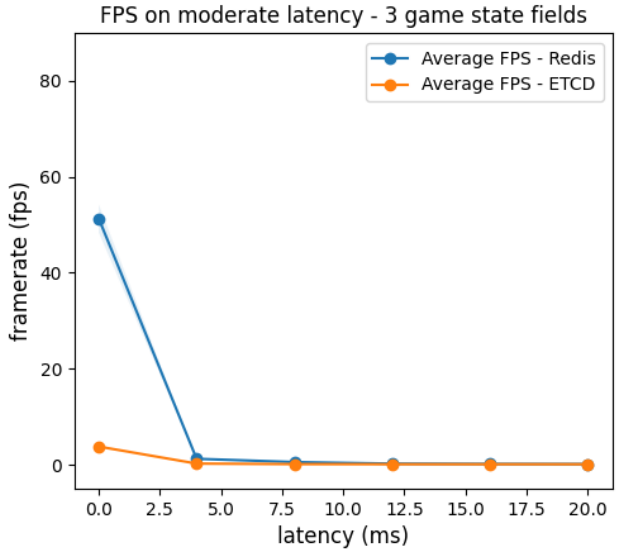


Figure 7.73: Graphics framerate comparison - 3 game state fields - cluster.

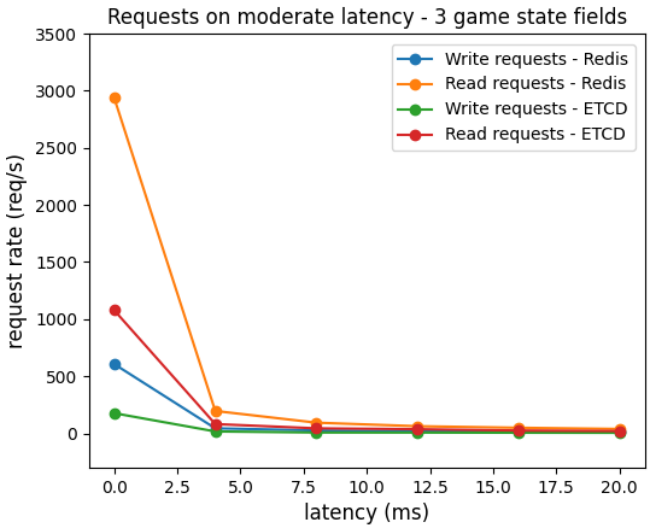


Figure 7.74: Requests processed comparison - 3 game state fields - cluster.

Considering these results, we can assert that, in the context of this specific approach, Redis performs better than ETCD in their respective 3-members cluster versions. Redis provides generally better performance than ETCD in situations with no latency introduced, whereas, when network latency is present, Redis still performs on par with ETCD, despite its HTTP_G pipelining implementation. In the cluster configuration this difference is even more evident than in the stand-alone version, considering the benefits provided by Redis Eventual Consistency mechanism.

7.2.5 Distribution of static game state data

In this experiment we began by introducing the storage of the `totLaps` game state field into the distributed database, in a configuration with only write requests.

Table 7.12: *totLaps* field storage write performance.

Field	Write request rate	Average framerate	CI low. bound	CI up. bound
totLaps	0.01 req/s	59.681 fps	59.239 fps	60.124 fps

If we then introduce the read requests, without changing the write request rate, we obtain results which are similar to the previous configuration, with a stable framerate around 60 fps.

Table 7.13: *totLaps* field storage read performance.

Field	Read request rate	Average framerate	CI low. bound	CI up. bound
totLaps	59.90 req/s	60.001 fps	59.993 fps	60.008 fps

Considering the lack of significant impact on performance, we proceed by replacing the `totLaps` field with the `maxDamage` field, and follow up with the same experiment.

Table 7.14: *maxDamage* field storage write performance.

Field	Write request rate	Average framerate	CI low. bound	CI up. bound
maxDamage	0.01 req/s	59.856 fps	59.573 fps	60.140 fps

If we then introduce the read requests, without changing the write request rate, we obtain results which are similar to the previous configuration, with marginally lower values with respect to the situation with the `totLaps` field. This is likely due to the larger amount of read requests, as we can see from table 7.15.

Table 7.15: *maxDamage* field storage read performance.

Field	Read request rate	Average framerate	CI low. bound	CI up. bound
maxDamage	1028.73 req/s	59.613 fps	59.291 fps	59.936 fps

Still, there is no significant impact on performance, as such we proceed by replacing the `maxDamage` field with the `raceType` field, and follow up with the same experiment.

Table 7.16: *raceType* field storage write performance.

Field	Write request rate	Average framerate	CI low. bound	CI up. bound
raceType	0.01 req/s	59.925 fps	59.775 fps	60.074 fps

If we then introduce the read requests, without changing the write request rate, we obtain results which are similar to the previous configurations, with values set around 60 fps. In this case, the rate of read requests is about half of the one measured for `maxDamage`, as we can see from table 7.17.

Table 7.17: *raceType* field storage read performance.

Field	Read request rate	Average framerate	CI low. bound	CI up. bound
raceType	532.29 req/s	59.616 fps	59.211 fps	60.021 fps

Considering the lack of significant impact on performance, we proceed by replacing the `raceType` field with the `ncars` field, and follow up with the same experiment.

Table 7.18: *ncars* field storage write performance.

Field	Write request rate	Average framerate	CI low. bound	CI up. bound
ncars	0.01 req/s	59.355 fps	58.088 fps	60.622 fps

If we then introduce the read requests, without changing the write request rate, we obtain results which are quite different from the previous configurations, as we can see from table 7.17.

Table 7.19: *ncars* field storage read performance.

Field	Read request rate	Average framerate	CI low. bound	CI up. bound
ncars	2141.24 req/s	26.062 fps	25.568 fps	26.557 fps

In this case the TORCS framerate is heavily impacted by the presence of read

requests, decreasing it by more than one half. This can be partially linked to the request rate, which is particularly high in this situation, with respect to the previous configurations. However, the `maxDamage` setting did not highlight a significant decrease in performance, even of the request rate was about half of the current one. As such, it is likely that other factors are the root cause for this performance degradation.

We assume that one relevant element to consider in this context is also the location in the TORCS codebase where these requests are performed. Some specific TORCS module, in fact, can be more heavily impacted by the inevitable delays in the operations, introduced by the implementation of distributed databases.
In particular, we notice that, differently from the other static game state fields, `ncars` requires multiple requests to be made from inside the *simulation module* of TORCS, which manages the game physics.
As such, we proceed with measuring the system performance without any simulation module references:

Table 7.20: *ncars* field storage read performance - no simulation.

Field	Read request rate	Average framerate	CI low. bound	CI up. bound
ncars	1421.61 req/s	57.588 fps	56.785 fps	58.392 fps

As we can see in table 7.20, without simulation module references, the performance greatly improves, despite still presenting an relatively high request rate. This validates our assumption that specific TORCS modules, such as *simulation*, are particularly impacted by operational delays. Moreover, the fact that the graphics framerate is heavily impacted by delays present in the physics management component, suggests that there might be a correlation between the two modules. We explore more of this aspect in the following experiment, in section 7.2.6.

7.2.6 Graphics and physics engine correlation

We now present the results obtained by graphics framerate measurements conducted on the TORCS system, with an increasing amount of delay introduced into the *simulation module*.
As we can see from Figure 7.75, even low amounts of delay introduced into the simulation module greatly impact the performance of the TORCS graphics module, by reducing the framerate by more than one half with even just 2 ms of delay introduced. After the initial sharp drop, the framerate decreases more gradually as the delay increases, still verifying our assumption of a direct correlation between the graphics and physics module in the TORCS architecture.
Then, after computing the theoretical upper bounds for the framerate at each step of the increasing delay, we proceed with comparing them with the actual measured framerate. Our expectation is that the actual framerate never exceeds the bounds we computed. Considering that at the 0 ms mark there is effectively no upper bound to be computed, we fix it at 125 fps in our representation, since this value is never reached by the measured framerate.
As we can see from Figure 7.76, our theoretical expectations are validated by the results we obtained, with the average graphics framerate never completely crossing the line related to the framerate bounds.

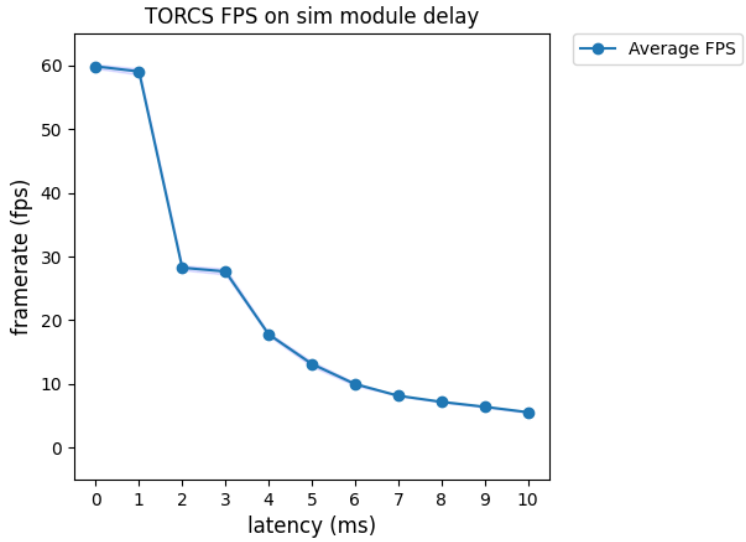


Figure 7.75: TORCS gprahics framerate on increasing simulation delay.

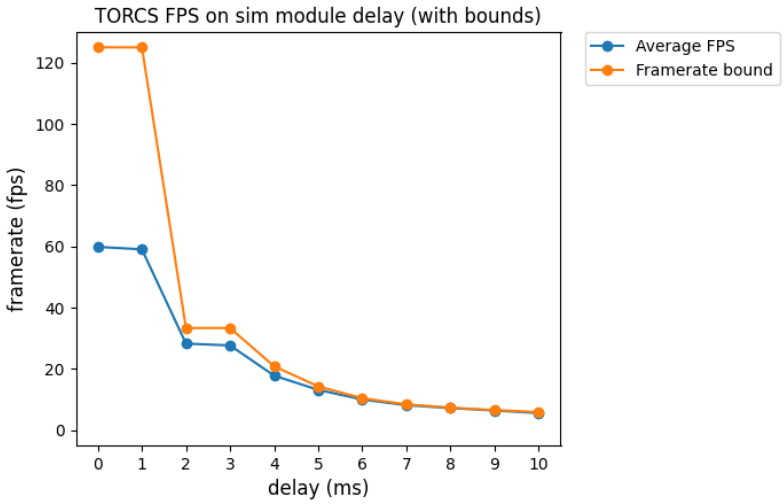


Figure 7.76: TORCS graphics framerate on increasing simulation delay w/ bounds.

To conclude, we have managed to validate our assumed correlation between the TORCS physics and graphics module. Additionally, we also succeeded in computing the actual impact even low amounts of delay have on the performance of the system, if located inside specific TORCS modules (e.g. simulation, graphics).

7.2.7 Graphics and game engine framerate correlation

Using the same configuration as in the previous experiment, we now measure the graphics framerate and compare it with measurements conducted on the Game Engine framerate, in order to verify the correlation between these two values. Additionally, we

compute the actual net operational time, considering the increasing delay we introduce into the system.

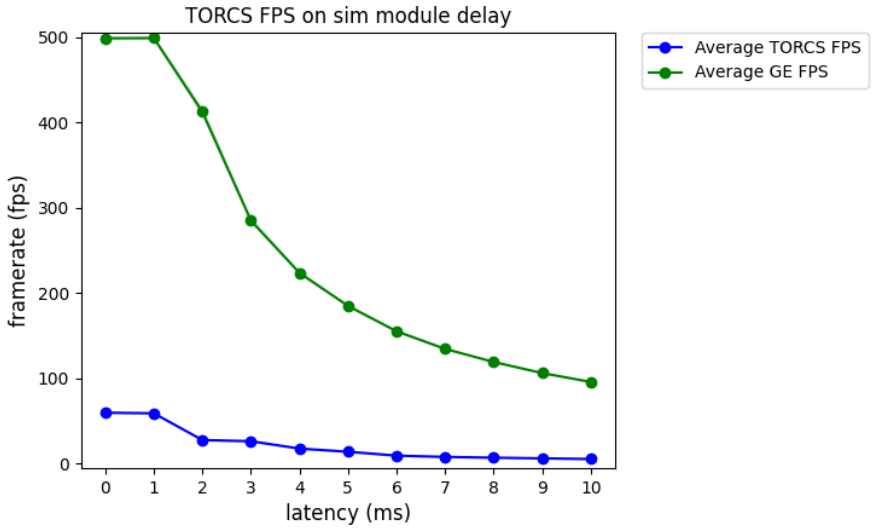


Figure 7.77: Graphics and GE_G framerate comparison.

As we can see from Figure 7.77, both framerates values gradually decrease, as the delay value increases. This is expected behaviour, considering the effects on the graphics framerate we verified in the previous experiment, but also indicates a direct correlation between the graphics framerate and the Game Engine framerate.

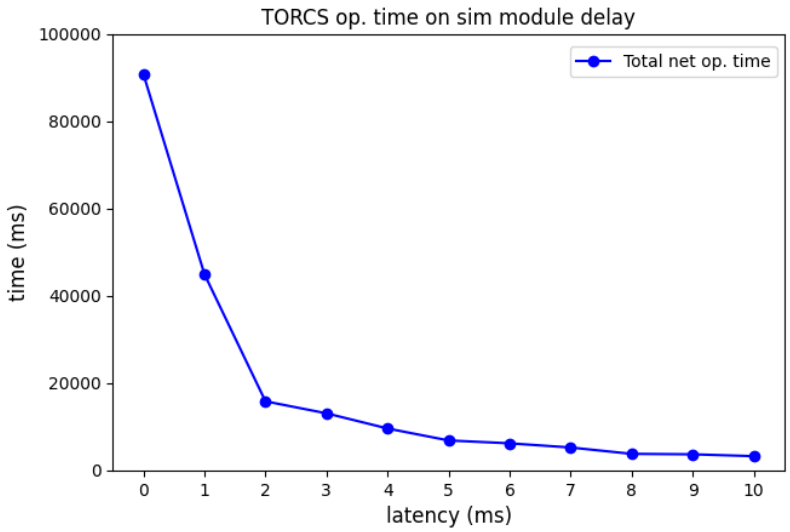


Figure 7.78: Net operational time w/ delays.

At the same time, in Figure 7.78, we can see how the net operational time becomes increasingly lower with the increasing delay, up to little more than 3000 ms over 90000 total ms of execution. This means that the actual amount of time available to compute

the frames is very limited, in a situation with moderate delay in the simulation module.

To conclude, we can assert that introducing delays into the simulation module is very problematic for the TORCS system performance. As such, any approach at game state distribution should avoid direct involvement in the TORCS main loop or in its modules operations.

7.2.8 Temporal State Manager inconsistency

In order to quantify the inconsistency between the local instance state and the remote instance state of TORCS, we compare the time before writing the state and after reading it from the distributed database.
The results we obtained for the different configurations are as follows:

Table 7.21: Temporal inconsistency in the State Manager.

System	Average time inconsistency	CI low. bound	CI up. bound	Sample size
Redis solo	4.804 ms	4.736 ms	4.872 ms	6439
ETCD solo	54.916 ms	51.050 ms	58.782 ms	736
Redis cluster	6.433 ms	6.320 ms	6.547 ms	4396
ETCD cluster	100.847 ms	91.539 ms	110.157 ms	443

As we can see from table 7.21, Redis provides much better performance than ETCD both in its cluster and stand-alone configuration, with a much lower inconsistency value. This is to be expected, considering the large amount of requests to be computed and the lack of any network latency.

We can also see how, similarly to what we identified in other experiments, the difference between the performance of the Redis cluster and stand-alone versions is much smaller than the one between the two ETCD configurations. This is also expected behaviour considering the technical characteristics of Redis, which allow it to provide an higher degree of horizontal scalability, as previously discussed.

Considering the moderate inconsistency values provided by the Redis configurations, we can conclude that they are a reasonable solution for the purpose of the State Manager. On the other hand, ETCD does not provide satisfactory performance in this context, with a much higher temporal inconsistency.

In Figure 7.79 we can see that the temporal inconsistency value does not change in a monotonic manner during the course of the race. In fact, regardless of the sampling temporal segment considered, the values always range from 0 to about 9 ms, with some value spikes above such value.

If we then look at the value distribution for all the samples, in Figure 7.80, we see a reflection of the behaviour we identified in the previous graph. The probability of each sample to assume a value between 0 and 9 ms is higher than values above 9

ms, with a generally homogeneous distribution. As such, considering this distribution, we validate the value of 4.804 ms, computed and presented in table 7.21, as arithmetic mean for the Redis stand-alone configuration.

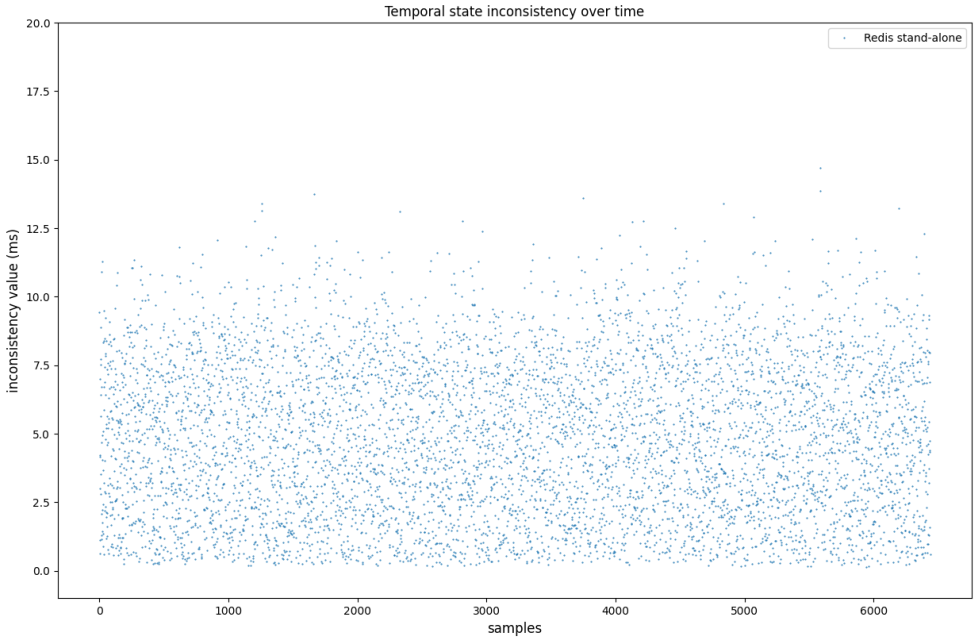


Figure 7.79: Temporal inconsistency over time - Redis

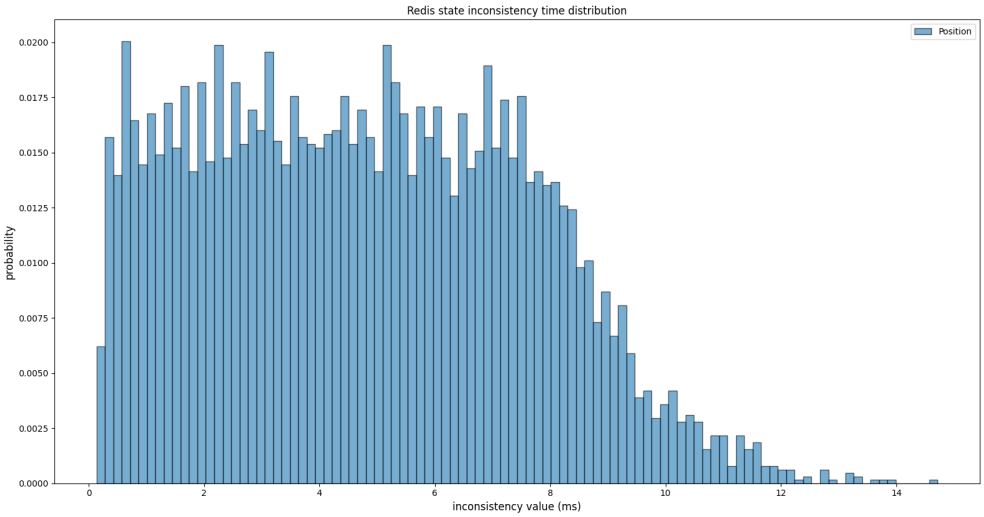


Figure 7.80: Temporal inconsistency values distribution - Redis

7.2.9 Positional State Manager inconsistency

During this experiment we measured the positional state inconsistency between remote and local instances of TORCS, in terms of: position, velocity and acceleration. We performed this measurement while introducing an increasing amount of network latency into the distributed database, in order to quantify the its effect on the positional state inconsistency.

Redis stand-alone configuration

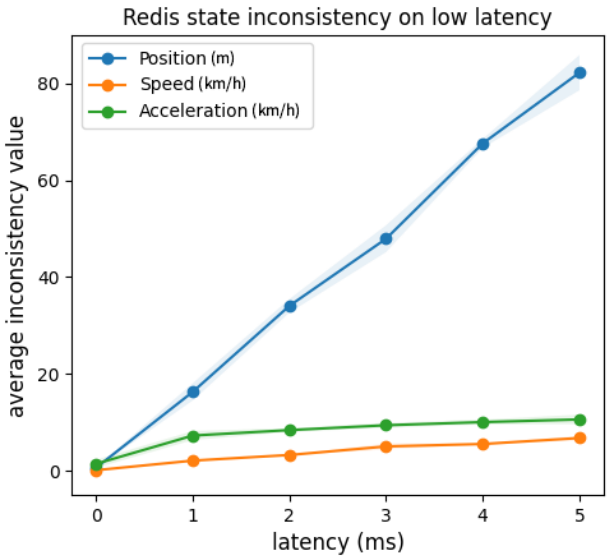


Figure 7.81: Redis positional state inconsistency.

As we can see in Figure 7.81, the impact on the positional state inconsistency caused by the introduction of even low (<5 ms) amounts of latency is very significant. The increase in inconsistency is mostly regular, with a much heavier impact on the position of the car, rather than speed or acceleration.

ETCD stand-alone configuration

In the ETCD stand-alone configuration, the behaviour is essentially similar to the Redis stand-alone configuration, as we can see in Figure 7.82, even if with a more irregular increase in inconsistency value.

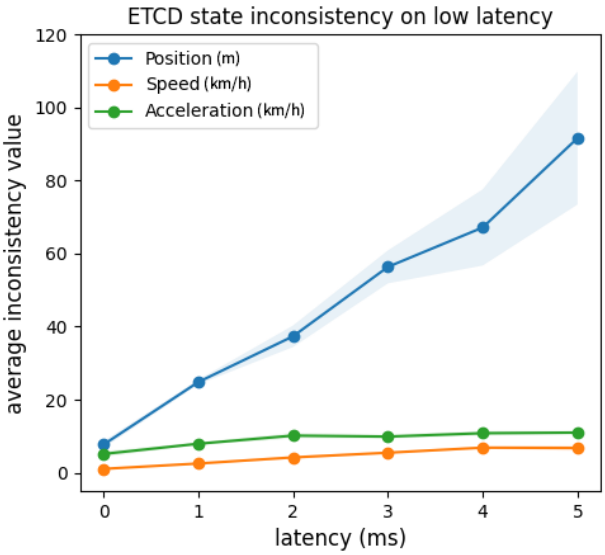


Figure 7.82: ETCD positional state inconsistency.

Redis 3-members cluster configuration

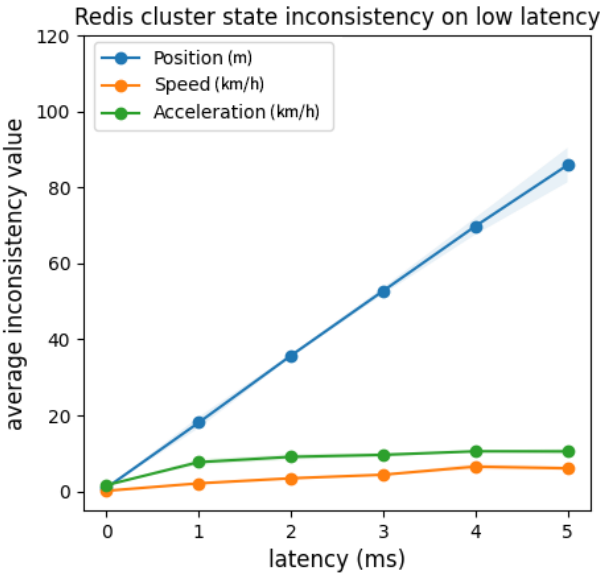


Figure 7.83: Redis cluster positional state inconsistency.

In the Redis 3-members cluster configuration, we experience a similar situation as in the stand-alone version, with a mostly regular increase of the inconsistency value as the amount of delay introduced increases.

ETCD 3-members cluster configuration

In the ETCD 3-members cluster configuration, we experience a similar situation as in the stand-alone version, with a mostly regular increase of the inconsistency value as the amount of delay introduced increases.

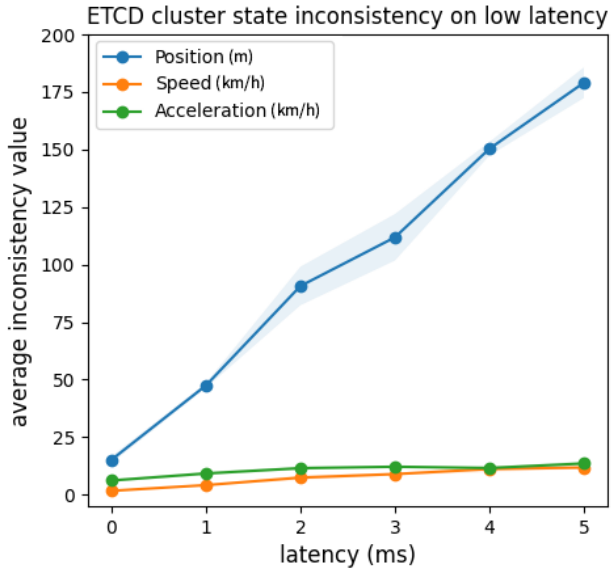


Figure 7.84: ETCD cluster positional state inconsistency.

Inconsistency configuration comparison

After experiencing a mostly similar behaviour in term of inconsistency increase with the network latency, in the four configurations we considered, we now compare the actual values obtained.
From the graphs (figures 7.85, 7.86, 7.87) we can see that the ETCD cluster version is presenting a higher degree of inconsistency, in terms of car position, speed and acceleration. The other three system configurations, on the other hand, perform better, with Redis generally providing slightly lower inconsistency with when no latency is present. However, leveraging the benefits provided by HTTP_G pipelining, ETCD provides performance similar to Redis, when latency is introduced.

Still, comparing the two cluster versions, Redis present better scalability in the number of nodes, with respect to ETCD, and thus better performance.

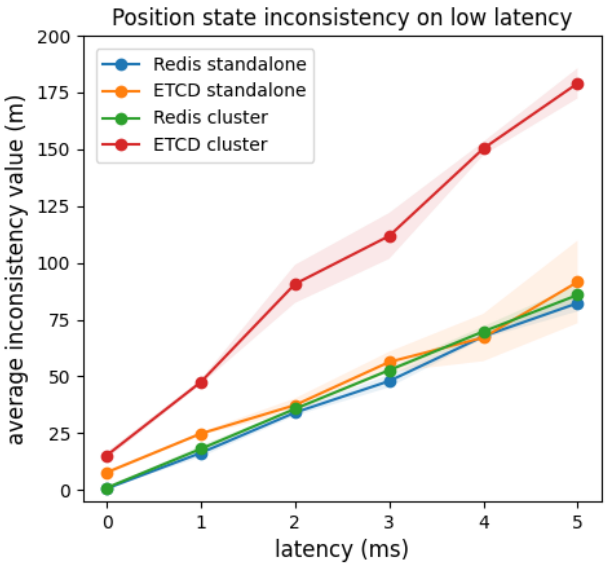


Figure 7.85: Position state inconsistency comparison.

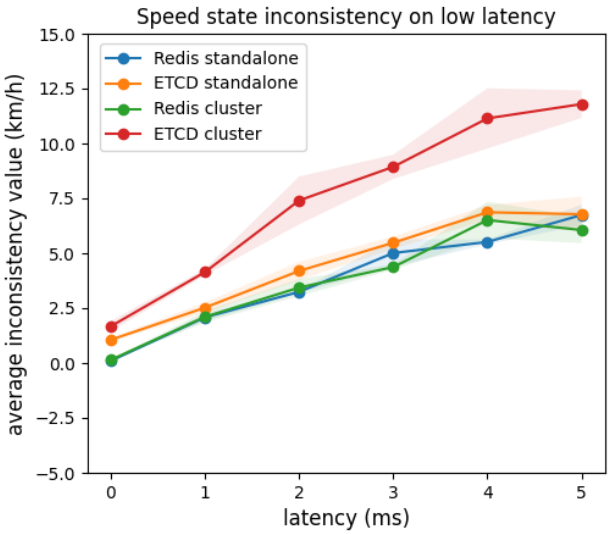


Figure 7.86: Speed state inconsistency comparison.

As we can see in Figure 7.88, the positional inconsistency value does not change in a monotonic manner during the course of the race and it's mostly measured within the 0.1 and 1 meters range. In this context, we only consider the samples obtained after the first 10 seconds of the race, in order to filter out the irregular behaviour of the cars at the starting grid.

It is interesting to notice how the samples, despite being independent points in the sampling population, seem to converge on specific values during the whole race duration. This behaviour is likely tied the State Manager middleware update mechanism and

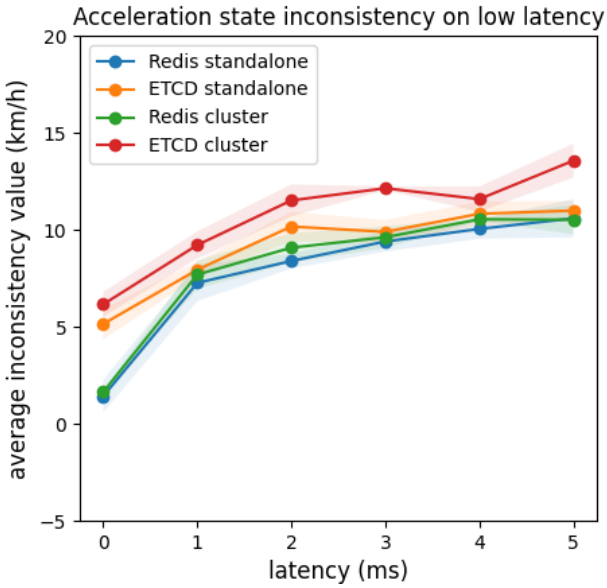


Figure 7.87: Acceleration state inconsistency comparison.

the sampling method used for this experiment.

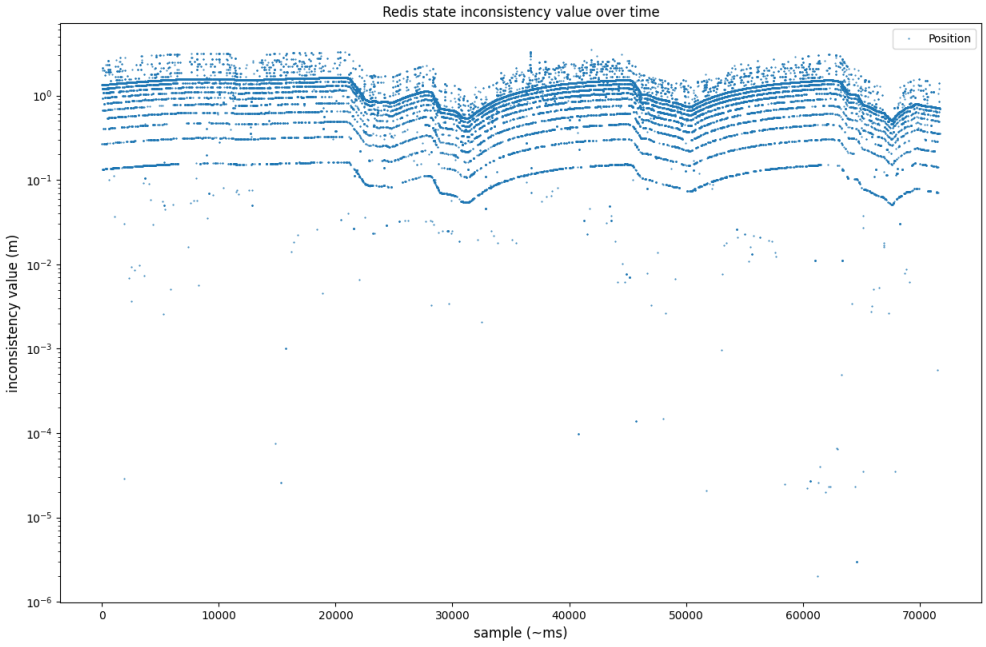


Figure 7.88: Positional inconsistency over time - Redis 0 latency.

The TORCS instance writing in the distributed database, in fact, writes the local car state values in the distributed DB_G by iterating sequentially on each car. At the same time, the read-only TORCS instance reads the values present in the distributed

database and iterates on each car sequentially, in order to set the new local car state. This process requires multiple write/read requests and a certain amount of time (ms) to be carried out completely for every car in the race, we call this time: *update window*. At the same time, the sampling method for this experiment reads the state of the two TORCS instances at a regular frequency of about 1 ms.

Considering that the state sampled during this experiment is related to only one car (the first one), it is reasonable that each sample inside the same State Manager update window presents an increasing amount of positional inconsistency. In fact, if the first car state is written at the beginning of the window, each following sample collected during the same update will be 1 ms farther from the current state until the end of the window. When the end is reached, a new update starts, as such the first car state is written again and the sampled state becomes synchronized again with the current state, thus the inconsistency value decreases. Taking into account that the sampling is performed at regular intervals, it is reasonable that the samples always converge to similar values during execution. A simplified representation of this behaviour is presented in Figure 7.89.

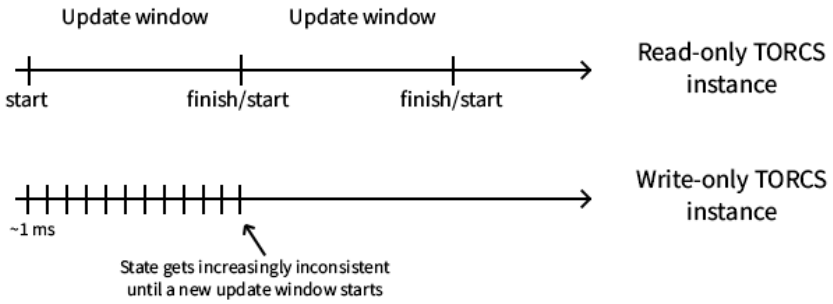


Figure 7.89: State inconsistency increase during sampling.

In the graph we can also notice how, while the repeating values are mostly regular, there is significant variation in certain temporal segments. This is likely tied to specific events happening to the car in the race (e.g. car crashing, car slowing down), which may reduce or increase the general amount of inconsistency we measure.

Finally we remark that, in the logarithmic scale representation we use, the values equal to 0 are not present. This is important to consider since, as we can see in Figure 7.91, a significant amount of samples actually presents a value equal to 0.

From the distribution of the values presented in Figure 7.90, we can see that the samples have a 1/4 probability of presenting a value equal to 0. This is a reasonable result considering the good performance provided by TORCS during the experiment execution.
As previously discussed, when the sampling is performed in a situation where the car state has just been written, the inconsistency is not present (equals to 0), since the local and remote states in that moment are identical. Additionally, we know that the time necessary to change the car state in the local instance of TORCS is about *2 ms*, since this is the update frequency of the TORCS simulation.
This means that, any sample obtained within the time between 0 and 2 ms is going

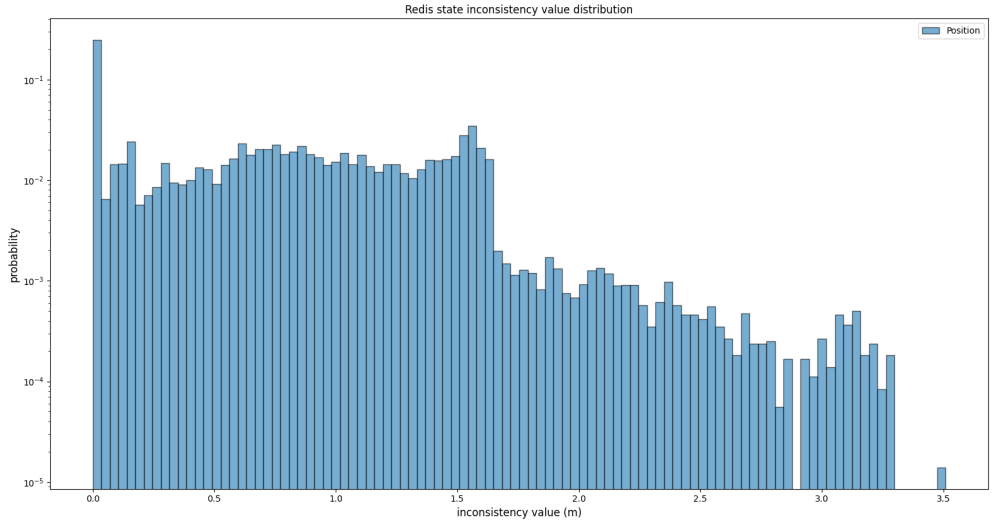


Figure 7.90: Positional inconsistency sample distribution - Redis 0 latency.

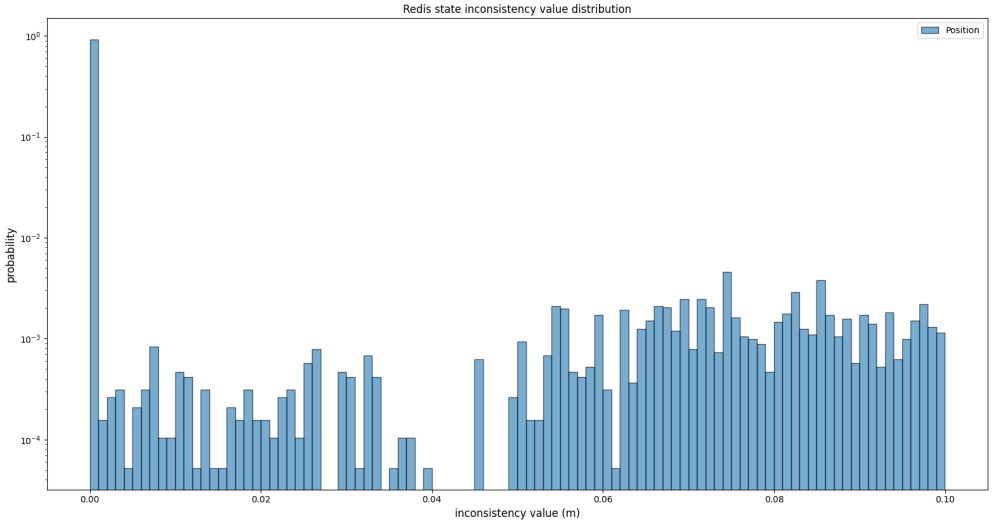


Figure 7.91: Positional inconsistency sample distribution - low values zoom-in.

to provide us with a remote game state which is completely synchronized with the local instance, since the car state in the local instance has not yet been updated by its simulation.

Most other samples have less than 1/40 probability to assume values in the range between 0 (m) and 1.5 (m). This regular distribution is likely related to the gradual increase in inconsistency we previously discussed. Moreover, it is reasonable that the distribution in Figure 7.90 does not present significant peaks of probability for any value in this range. In fact, even if some inconsistency values seem to be more regular and frequent in Figure 7.88, there is still significant variation inside the $\sim 0 \div 1.5$ range, which reasonably provides us with a more homogeneous distribution of the samples in

this graph.

We also have a probability which is marginally higher than 1/40 that the value measure about 1.5 (m) or more. This may be related to the values sampled during graphical glitches, considering the value and the probability we computed.

Finally, in this graph we can see how the inconsistency values validate the result we obtained as arithmetic mean: 0.696 (m).

Chapter 8

Conclusion

*This chapter provides a final discussion about the outcome of the whole research effort, considering also the **limitations** of the study design and findings, and the **recommandations for future research**.*

8.1 Final assessment

In this work, we investigated the feasibility of containerizing Legacy Game Engine modules and having them communicate with each other, with the purpose of implementing a Distributed Game Engine architecture. Our results show that it is indeed possible to containerize these modules and have them communicate with each other using various inter-container communication techniques.

Moreover, we also explored the use of distributed databases as a solution for allowing consistent, but not necessarily quick, data communication between the Game Engine modules. We found that distributed databases can be a viable option for ensuring data consistency between modules, although their performance may not always meet the high-speed requirements of some gaming applications, such as TORCS.

To this end, we compared the performance of Redis and ETCD in the context of a distributed version of TORCS. We found that Redis generally provided better framerate compared to ETCD, both in their standalone and 3-member cluster versions. This was evident in the various configurations that we tested, with Redis consistently delivering higher framerate than ETCD, in context where no latency is present.
Still, when network latency is introduced, ETCD's implementation of HTTP_G pipelining may provide interesting benefits to mitigate the effect of delays on the single packets, especially in the stand-alone version of the systems, where their difference is not particularly pronounced.

In the cluster versions of the systems, the difference in framerate between Redis and ETCD was particularly pronounced. Thanks to its Eventual Consistency mechanism, Redis was able to achieve significantly higher framerate than ETCD, indicating its superior performance in this setting.

Overall, our results suggest that Redis is a more suitable choice for distributed applications that require high responsiveness, such as a distributed version of TORCS.

On the other hand, ETCD may be more suitable for applications that require strong consistency and support for distributed transactions, but may not be as well-suited for high-performance applications.

Working on an alternative solution, we implemented a State Manager middleware as a means of storing and distributing the state of a distributed version of TORCS. Our results showed that the State Manager was able to provide state storage and distribution without significantly impacting the performance of the system and with a generally moderate state inconsistency, in its Redis-based implementation.
Our experimental results demonstrated that the State Manager was able to provide fast and reliable state storage and distribution, with minimal impact on the framerate of the distributed version of TORCS. We also found that the State Manager was able to scale well with increasing numbers of cars, showing good performance even if with some graphics glitches.

We also found that network latency has a noticeable impact on the performance of the distributed version of TORCS, leading to a significant degradation in framerate and a less enjoyable gameplay experience. This highlights the challenges of using distributed databases in these types of high-performance and latency-sensitive systems. Despite the use of distributed databases optimized for fast and reliable data storage and retrieval, we found that they did not provide good performance when network latency was introduced. This suggests that more advanced techniques may be needed to effectively handle the high responsiveness requirements of TORCS in a distributed setting.

8.2 Limitations

While the tests conducted in this work were able to provide interesting insights into the performance of a TORCS distributed system based on shared storage, it is important to note that they were conducted in a local Docker environment, rather than a real-world Cloud$_G$ Computing scenario. This means that the results may not necessarily generalize to a production setting, where the system would be deployed on a Cloud$_G$ platform such as Amazon Web Services, Microsoft Azure or Kubernetes.

A local Docker environment does not necessarily simulate the network latency and bandwidth constraints that are commonly encountered in a Cloud$_G$ environment. This can lead to differences in performance and resource utilization compared to a real-world context.

One additional limitation is that only a few of the TORCS modules and libraries were containerized and incorporated into the final architecture. This means that this distributed version of TORCS is currently not complete, and only serves as a prototype for evaluating the feasibility of this approach.

Finally, another limitation is that the State Manager middleware is limited in the amount of game state related data that it can store on the distributed database. The TORCS game state consists of a large amount of data, including information about the track, the cars, and their positions and velocities. In order to maintain a reasonable level of performance, the State Manager can only store a subset of this data on the

distributed database.

This means that the State Manager is not able to store the full game state at any given time, and some information may be lost or not accurately represented. This could potentially impact the accuracy and representation of the game, as well as generate graphics glitches in the distributed TORCS instances.

Despite these limitations, this distributed version of TORCS and the related State Manager middleware developed during this project represent a significant step forward in the development of Distributed Game Engines.

8.3 Recommandations for future research

Considering the limitations we discussed in the previous section, there are multiple areas where future research can be conducted in order to provide interesting additional development.

For instance, further work could be done to fully decompose the TORCS system architecture and address other aspects such as security and usability. Multiple additional libraries and modules of the original architecture may require considerable work to correctly manage all the reference present in the monolithic structure. Still, this effort is important to achieve a completely distributed version of TORCS.

One area of further research that could be conducted in the development of a distributed version of TORCS is the use of alternative distributed databases. In this work, the State Manager middleware was developed using ETCD and Redis as the underlying distributed databases. However, there are many other distributed databases available that could potentially be used for this purpose, and could prove to be more fitting solutions, depending on their technical characteristics.

As mentioned in the previous section, a possibly interesting development is the storage of the full TORCS game state into the distributed database, through the State Manager middleware. This might be difficult to realize with the solutions we currently evaluated, however newer or different approach may allows this kind of operation to be performed with reasonable performance.

Finally, as we saw, network latency and bandwidth constraints can have a significant impact on performance and scalability of the TORCS distributed system. As such, there is much room for further research and innovation in the optimization of network communications. By improving the efficiency of the system's network communications, it may be possible to achieve even greater levels of performance and scalability.

Appendix A

Glossary

AI: Artificial intelligence (AI), refers to the ability of a computer or machine to perform tasks that would normally require human intelligence, such as learning, problem-solving, decision-making, and language understanding.

AoI: in the context of game state storage, an Area of Interest (AoI) is a region in the game world that is being actively used or modified by players.

API: an application programming interface (API) is a set of rules, protocols, and tools for building software and applications. It specifies how software components should interact and allows for communication between different systems.

CLI: Command-line interface (CLI) is a type of interface that allows users to interact with a computer, device or application by entering commands through a command prompt.

Cloud: Cloud (computing) refers to the delivery of computing resources, such as servers, storage, and software, over the Internet. It allows users to access and use these resources on demand, without the need to build and maintain their own infrastructure.

DB: referencing Database. A database is a collection of data that is organized in a structured manner and can be accessed electronically. In this document we often refer to Distributed DBs, which are databases spread across multiple machines, often in different locations.

edge-cloud: edge-cloud is a term used to describe a distributed cloud computing architecture that brings cloud computing resources and services closer to the edge of the network, closer to the devices and users that need them.

FPS: First-person shooter (FPS) is a genre of video games that is characterized by fast-paced action and the use of weapons, typically viewed from a first-person perspective.

GE: referencing Game Engine. A game engine is a software development environment designed for creating video games. It provides a set of tools and frameworks for developers to build games more efficiently by abstracting away lower-level hardware and

graphics processing. Game engines often include a physics engine, collision detection, scripting, animation, and other features that are common to most games.

GUI: Graphical User Interface (GUI) is a type of user interface that allows users to interact with a computer or device using visual elements such as windows, icons, and menus.

HTTP: HTTP (Hypertext Transfer Protocol) is a standard application layer protocol for transmitting hypermedia documents, such as HTML. It is used for communication between clients and servers on the World Wide Web.

OS: an Operating System (OS) is a software program that manages the hardware and software resources of a computer. It provides a platform for running other applications and controls the way in which a user interacts with the computer.

peer-to-peer: also known as P2P, it refers to a type of network architecture in which each node or "peer" in the network can act as both a client and a server. In a P2P network, there is no central server that controls the network; instead, each node is able to communicate directly with other nodes and share resources such as data, processing power, or bandwidth.

PaaS: Platform as a Service (PaaS) is a type of Cloud computing service that provides a platform for building, deploying, and managing applications over the Internet.

QoE: Quality of Experience (QoE) refers to a measure of how well a user is able to use a product or service. In the context of technology, QoE is often used to describe the overall satisfaction and enjoyment that a user experiences when interacting with a device or system.

RR: Round robin is a scheduling algorithm that is used to allocate resources fairly among a group of requests or processes. In a round robin system, each request or process is given a turn in a fixed order, and the order is repeated until all requests have been serviced.

RTT: Round-Trip Time (RTT) is a measure of the time it takes for a packet of data to be sent from a source to a destination and for a response to be received back at the source.

soft-time: in the context of simulations, "soft time" refers to a type of simulation time that is not tied to real-time clock and can be accelerated or slowed down as needed.

SSH: SSH (Secure Shell) is a network protocol that allows secure remote login and other secure network services over an unsecured network.

SUT: in the context of software testing, the system under test (SUT) is the software or system that is being tested.

UDP: User Datagram Protocol (UDP) is a connectionless protocol that is used to transmit data across a network.

UI: User Interface (UI) refers to the means by which a user interacts with a computer or device. It includes the visual elements of the interface, such as the layout, appearance, and visual design, as well as the functionality and control elements, such as buttons, menus, and input fields.

WAL: Write-Ahead Log (WAL) is a type of log file that is used to record changes to a database. It is called a "write-ahead" log because the log is written to disk before the corresponding changes are made to the database itself.

XML: Extensible Markup Language (XML) is a markup language that is used to encode data in a text format. It is designed to be both human-readable and machine-readable, and it is often used for storing and transmitting data in a structured manner.

YAML: human-readable data serialization language that is commonly used for storing and transmitting data.

Bibliografy

Bibliographical references

[1] A. Andrade. *Game engines: a survey.* EAI Endorsed Transactions on Game-Based Learning, 2015.

[2] Glen Berseth and Ravjot Singh. *Asynchronous Real-time Multiplayer Game With Distributed State.* 2015.

[3] Ashwin R. Bharambe, Jeff Pang, and Srinivasan Seshan. *A Distributed Architecture for Interactive Multiplayer Games.* Carnegie Mellon University, 2005.

[4] Kajal T. Claypool and Mark Claypool. *On frame rate and player performance in first person shooter games.* Springer Multimedia Systems Journal (MMSJ), 2007.

[5] Mark Claypool and David Finkel. *The effects of latency on player performance in cloud-based games.* Worcester Polytechnic Institute, 2014.

[6] Ziqiang Diao. *Consistency Models for Cloud-based Online Games: the Storage System's Perspective.* Otto-von-Guericke University Magdeburg, 2013.

[7] Davide Gadia et al. *A Distributed Game Engine for Mobile Games on the Android Platform.* University of Milan, 2017.

[8] Luigi De Giovanni et al. *Revamping Cloud Gaming with Distributed Engines.* IEEE Internet Computing, 2022.

[9] Jason Gregory and Richard Lemarchand. *Game Engine Architecture - second edition.* CRC Press, 2015.

[10] Qi Liu. *Integrating Game Engines into the Mobile Cloud as Micro-services.* University of Saskatchewan, 2018.

[11] Daniele Loiacono, Luigi Cardamone, and Pier Luca Lanzi. *Simulated Car Racing Championship: Competition Software Manual.* arXiv, 2013.

[12] Dario Maggiorini et al. *SMASH: a Distributed Game Engine Architecture.* IEEE Symposium on Computers and Communication (ISCC), 2016.

[13] Laura Mazzuca. *Distributed Cloud Gaming Pipeline.* Universidad Complutense de Madrid, 2021-2022.

[14] Farouk Messaoudi et al. *Performance Analysis of Game Engines on Mobile and Fixed Devices.* Association for Computing Machinery, 2017.

[15] Philipp Moll, Sebastian Theuermann, and Hermann Hellwagner. *Distributing the Game State of Online Games: Towards an NDN Version of Minecraft*. Klagenfurt University, 2019.

[16] Gabriele Pozzan and Tullio Vardanega. *Rafting multiplayer video games*. University of Padua, 2021.

[17] Karzan Hussein Sharif and Siddeeq Yousif Ameen. *Game Engines Evaluation for Serious Game Development in Education*. Duhok Polytechnic University, 2021.

[18] Ransi Nilaksha De Silva et al. *Towards Understanding User Tolerance to Network Latency and Data Ratein Remote Viewing of Progressive Meshes*. National University of Singapore, 2010.

[19] Tianzhu Zhang, Carla Fabiana Chiasserini, and Paolo Giaccone. *TAME: an Efficient Task Allocation Algorithm for Integrated Mobile Gaming*. IEEE Systems Journal, 2018.

Web references

[20] *Consensus (computer science) - Wikipedia*. URL: https://en.wikipedia.org/wiki/Consensus_(computer_science).

[21] *Development tutorial of the Minecraft Clone*. URL: https://www.youtube.com/watch?v=YgvNuY8Iq6Q&list=PL6_bLxRDFzoKjaa3qCGkwR5L_ouSreaVP.

[22] *Docker - Website*. URL: https://www.docker.com/.

[23] *Docker - Wikipedia*. URL: https://en.wikipedia.org/wiki/Docker_(software).

[24] *Docker-compose documentation*. URL: https://docs.docker.com/compose/.

[25] *Docker overview*. URL: https://docs.docker.com/get-started/overview.

[26] *ETCD C++ API*. URL: https://github.com/etcd-cpp-apiv3/etcd-cpp-apiv3.

[27] *ETCD data model*. URL: https://etcd.io/docs/v3.4/learning/data_model/.

[28] *ETCD Docker guide*. URL: https://etcd.io/docs/v2.3/docker_guide/.

[29] *ETCD documentation*. URL: https://etcd.io/docs/v3.5/.

[30] *ETCD FAQ*. URL: https://etcd.io/docs/v3.3/faq/.

[31] *ETCD maintenance*. URL: https://etcd.io/docs/v3.2/op-guide/maintenance/.

[32] *ETCD website*. URL: https://etcd.io/.

[33] *Flightgear development*. URL: https://wiki.flightgear.org/Howto:Understand_the_FlightGear_development_process.

[34] *Flightgear Github*. URL: https://github.com/FlightGear/flightgear.

[35] *Flightgear performance*. URL: https://wiki.flightgear.org/Troubleshooting_performance_issues.

[36] *Flightgear Wiki*. URL: https://wiki.flightgear.org/Main_Page.

[37] *FoFix GitHub*. URL: https://github.com/fofix/fofix.

[38] *Game engine - Wikipedia*. URL: https://en.wikipedia.org/wiki/Game_engine.

[39] *General ETCD description*. URL: https://alibaba-cloud.medium.com/getting-started-with-kubernetes-etcd-a26cba0b4258.

[40] *Godot Architecture*. URL: https://docs.godotengine.org/en/stable/development/cpp/introduction_to_godot_development.html.

[41] *Godot Custom modules*. URL: https://docs.godotengine.org/en/stable/development/cpp/custom_modules_in_cpp.html.

[42] *Godot Documentation*. URL: https://docs.godotengine.org/en/stable/.

[43] *Godot GitHub*. URL: https://github.com/godotengine/godot.

[44] *Godot servers*. URL: https://docs.godotengine.org/en/stable/development/cpp/custom_godot_servers.html.

[45] *HTTP pipelining - Wikipedia*. URL: https://it.wikipedia.org/wiki/HTTP_pipelining.

[46] *Minecraft - Wikipedia*. URL: https://it.wikipedia.org/wiki/Minecraft.

[47] *MMO games - Wikipedia*. URL: https://it.wikipedia.org/wiki/Massively_multiplayer_online.

[48] *Nvidia-docker documentation*. URL: https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/overview.html.

[49] *Nvidia-docker repository*. URL: https://github.com/NVIDIA/nvidia-docker.

[50] *OrbitDB GitHub*. URL: https://github.com/orbitdb/orbit-db.

[51] *Patched TORCS GitHub*. URL: https://github.com/fmirus/torcs-1.3.7.

[52] *Python Minecraft Clone Github*. URL: https://github.com/obiwac/python-minecraft-clone.

[53] *PyTorcs GitHub*. URL: https://github.com/gerkone/pyTORCS-docker/tree/master/torcs.

[54] *Raft Consensus Algorithm*. URL: https://www.geeksforgeeks.org/raft-consensus-algorithm/.

[55] *Redis C++ API*. URL: https://github.com/sewenew/redis-plus-plus.

[56] *Redis clients*. URL: https://redis.io/resources/clients/.

[57] *Redis Docker guide*. URL: https://redis.io/docs/stack/get-started/install/docker/.

[58] *Redis documentation*. URL: https://redis.io/docs/.

[59] *Redis replication*. URL: https://redis.io/docs/management/replication/.

[60] *State machine replication - Wikipedia*. URL: https://en.wikipedia.org/wiki/State_machine_replication.

[61] *Torcs official site*. URL: http://torcs.sourceforge.net/.

[62] *Torcs source code*. URL: https://sourceforge.net/projects/torcs/.