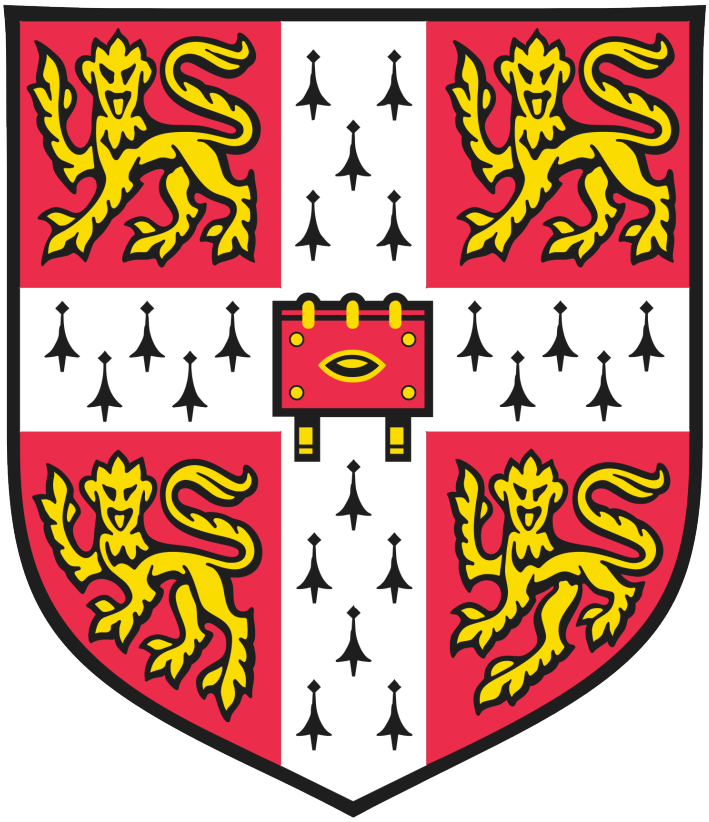
Dan Wendon-Blixrud

DAN:
Distributed Anticheat Networking



University of Cambridge

Computer Science Tripos – Part II
Queens' College

2024

Declaration

I, Dan Wendon-Blixrud of Queens' College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this dissertation I did not use text from AI-assisted platforms generating natural language answers to user queries, including but not limited to ChatGPT. I am content for my dissertation to be made available to the students and staff of the University.

Signed: 

May 10, 2024

Proforma

Candidate Number: **2424C**
Project Title: **DAN: Distributed Anticheat Networking**
Examination: **Computer Science Tripos – Part II, 2024**
Word Count: **9529**
Code Line Count: **4894**
Project Originator: **The candidate**
Project Supervisor: **Prof Andrew Moore**

Original Aims

To design, develop, and evaluate a software framework for developing video games. The framework should be specifiically designed for online multiplayer games, and use state machine replication at its foundation for state synchronisation. After being built, the framework should be evaluated on its success as a practical software library, which may involve using the framework to develop a demonstration implementation game.

Work Completed

All success criteria were met. The software framework was completed along with a demonstration implementation game, a practicality evaluation, and many extensions.

Special Difficulties

None.

Contents

1. Introduction

1.1. Motivation

In online multiplayer video games, *cheating* is defined as any behaviour used to gain an unfair advantage against other players [1]. Being a victim of cheating discourages players from playing or spending money to play the game [2]. Thus both companies and countries want to reduce cheating to reduce its negative economic impact. Some companies have sued for $10Ms [3], and countries like South Korea and China have even made cheating illegal [4]. Armitage et al. explain that "cheating is prevalent in online games because such games combine competitiveness with a sense of anonymity — and the anonymity leads to a lessened sense of responsibility for one's actions" [5]. Cheating is therefore both an important issue worth tackling, and unlikely to ever completely stop.

Of the many forms of cheating found in online multiplayer games [1], some are *solved*. This means that there is an agreed upon and reliable best practice to prevent them. For example, exploiting bugs in game software can be solved by following good software development practices. Some forms of cheating, however, are *unsolvable*. This means they cannot be reliably prevented in any practical way. For example, player collusion, where players communicate with each other outside of the game in an undetectable way.

One form of cheating which is different, however, is the exploitation of *replication*. Replication here is used in the context of distributed systems: it is the process of replicating state across nodes in a network. In the more specific context of online multiplayer video games, this means replicating the game's state across each player's clients, usually in real-time. Exploiting replication is especially interesting because it is neither solved, nor unsolveable — any solution is defined by the compromises it makes.

Despite this, there has been surprisingly little innovation in replication within modern games. This may be due to the increasing popularity of *game engines*, which are software frameworks and toolkits designed specifically for game development. The market share of the top 2 game engines has increased from 8% in 2010 to 65% in 2021 [6], suggesting a homogenisation of video game technology — replication included.

This homogenisation of video game replication presents a great opportunity to innovate, especially when considering that every approach has its own relative strengths and weaknesses.

1.2. The Intersection of Four Easy Problems

Building a game which is tolerant against replication exploitation is not difficult in isolation. The same applies to building a real-time or an online multiplayer game. The challenge game developers face is to build games which have all three properties simultaneously.

It is then even more difficult to build a generalisable game engine for such games, with this intersection of problems visualised in Figure 1. This dissertation will therefore focus more on approaches which address many of these problems at once.

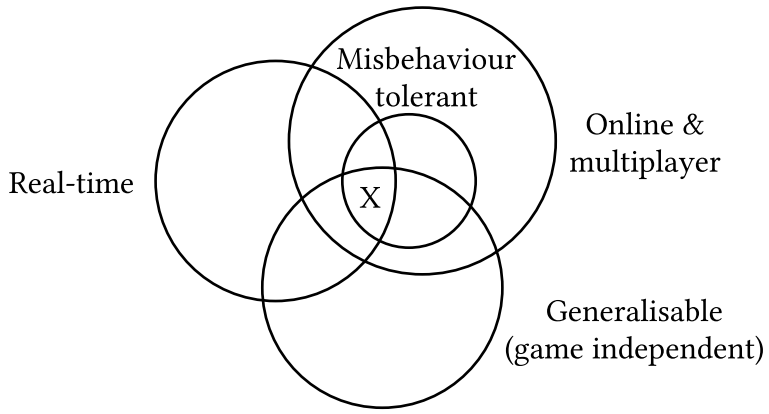


Figure 1: The intersection of these four problems ("X") is significantly more difficult than each problem individually.

1.3. Previous Work

I will briefly provide an overview of the existing approaches taken to replication in video games across three categories: networking models, replication techniques, and game engines.

1.3.1. Networking Models

The most common networking models used in games are *client-server* and *peer-to-peer*. Within client-server models, there is also the distinction between *client-authoritative* and *server-authoritative* models. Similarly, within peer-to-peer models there is a distinction between *dynamic server* and *fully connected* models.

Client-server, client-authoritative games such as *Minecraft* [7] are the simplest to make, based on that they require the least networking logic to implement. The downside, however, is that they are completely vulnerable to cheating from misbehaving clients.

The direct solution to this is to change to a client-server, server-authoritative model. Games such as *CS:GO*, *Overwatch*, and *Roblox* use this [8]–[10]. This moves the misplaced trust from the untrusted client onto the trusted server. While this approach is both very effective and widely used, it significantly increases the cost of running the servers. Roblox, for example, spent "over $400 million over the course of a year" on a single data centre [11].

Games using a peer-to-peer networking model instead avoid these high server costs. The downside is that clients' private IP addresses are exposed, and that the maximum number of players in a game is significantly limited. Dynamic server models such as in *Mario Kart 8* [12] are also susceptible to misplaced-trust cheats in some cases. Fully connected models such as *Doom* [13] instead tend to suffer more generally from the complexities of leaderless distributed systems.

1.3.2. Replication

Independent of networking models is the choice of replication model. Client-server games often use a form of passive replication, whereas peer-to-peer games often use active replication [14].

With passive replication, one node is responsible for computing the state and broadcasting its results to the other nodes. In CS:GO for example, the game is run on the server, and the clients' primary function is to display the data the server has sent them. As with client-server, server-authoritative networking models, passive replication often leads to a high server running cost.

Active replication instead has each node compute the state for itself. For this, games often use one of three forms of state machine replication (SMR) [14]: lockstep [15], delay-based [16], and rollback [16]. Traditionally, lockstep and delay-based were the most common approaches when using active replication, but rollback has become more common in recent games.

The benefits and drawbacks of active replication, and specifically SMR, are significant. SMR has the benefits of eliminating the high server costs of passive replication, while still also being tolerant against misbehaving clients. SMR's downsides consist of its reduced support for high player counts, its requirement for deterministic execution, and the fact that every player has access to all information about the game. While these downsides are severe, there are many games which are unaffected by them and would therefore benefit greatly from SMR.

1.3.3. Game Engines and Libraries

The two most popular game engines in 2021 were *Unity* and *Unreal Engine*. They both provide a client-server, server-authoritative, active replication service which are tolerant against misbehaving clients [17], [18]. Unity also provides a peer-to-peer replication service, however it is not tolerant against misbehaviour [19].

I found no evidence of a professional game engine based on SMR[1], but did find some projects which have come close. The game engine *Bevy* [20], for example, had a long-standing proposal for integrating SMR, but it was eventually dismissed due to being too unrealistic to implement [23]. Another example is *GGPO* [24], which has successfully integrated SMR but not in the form of a game engine. Instead, GGPO is a standalone software library which is integrated into existing peer-to-peer games to add SMR. This approach is effective, but GGPO's limited scope and standalone architecture makes it incompatible with most games' needs.

1.4. Outline

In this dissertation I introduce an SMR-based game engine with rollback called *DAN*. DAN addresses the gap in the market for a general-purpose, professional game engine built with SMR at its foundation.

The rest of this dissertation will describe DAN's design process, implementation, and evaluation.

[1] Game Creators Club's QuickGame game engine [21] uses SMR with rollback [22], but is designed as an educational toy rather than a professional game engine.

2. Preparation

This chapter provides an overview of the work completed before the implementation was started. I developed this project using only knowledge from Part 1A and 1B of the Tripos, meaning is no background material covered in this section.

2.1. Programming Language and Framework

The choice of programming language and development platform was extremely important. A good selection of development tools here would greatly benefit DAN's development, as well as future success beyond this project. I performed a systematic evaluation of 29 potential development platforms, selected from three sources. The full details can be found in the appendices (Appendix A), but I will also provide an overview here.

To evaluate the development platforms, I developed a set of requirements for an ideal platform to meet. These were:

- **Modifiable** — The development platform must allow me to build or integrate my engine. There cannot be too many incompatible elements which would make the project extremely inefficient or time-consuming.
- **Unrestrictive** — It is more important that the development platform is suitable for general-purpose professional game development than that it is accessible to beginners.
- **Cross-platform** — The development platform must be able to build for multiple devices from a single codebase. Customers play games on many different devices, and developers do not want to re-develop their game for each device unnecessarily. My engine's real-world success will be limited if cross-platform support is limited.
- **Popular** — The development platform must be well known and well liked, to help minimise the cost of learning to use my game engine.
- **Performant** — The development platform must be able to develop high performance games, since modern games often push the limits of the hardware's capabilities.

The two options which met all the criteria were:

- Dart with the Flutter UI framework
- C++ with the QT UI framework

I researched both and found that Dart with Flutter had the highest potential for developing a successful game engine. On top of meeting the five requirements, Dart and Flutter were modern, explicitly focussed on a great developer experience [25], and growing in popularity [26].

An important characteristic of Flutter was its relatively uncommon application in game development. This lack of competing game engines meant DAN's chance of success would be higher.

2.1.1. Dart with Flutter

To prepare for the project, I learnt more about Dart and Flutter by studying the online documentation [27], [28]. I focussed on the following topics during my research:

- **Dart and Flutter style** — Writing idiomatic code was important for game developers to be able to read or modify DAN's source code.
- **Flutter's internals** — Writing performant code required an understanding of how Flutter's technology stack works.

- **Dart's asychronous support** — Given DAN's focus on network replication, a large portion of the codebase would need to be written asynchronously.
- **Cross-platform compilation** — Understanding how source code could compile differently to different platforms was important for ensuring the determinacy needed for SMR.
- **Existing Flutter game engines** — Understanding how similar problems were solved in other Flutter game engines could help avoid re-inventing solutions unnecessarily.

2.2. Success Criteria and Evaluation

In order to measure the success of the project I will use the success criteria defined in the original project proposal (Appendix C). These are reproduced below for convenience:

1. Develop a distributed state synchronisation data structure.
2. Develop a user interface module to accept user input and render the game.
3. Develop a game driver module, which provides the logic to 'step forward' and run the game.
4. Develop a server module to relay events between clients.
5. Combine the modules into a running game.
6. Evaluate the practical success of the project using metrics such as framerate and synchronisation quality (deviation from true state, rate of jitter), under conditions such as varying player count, latencies, and bandwidth.

Criteria 1 and 4 were to be primarily evaluated through unit testing. This would allow me to quantitatively measure how much functionality was developed. Criteria 2, 3, and 5 were to be primarily evaluated through integration testing. This would demonstrate how well each module worked individually and when combined with others. Finally, success criteria 6 was important to determine the real-world viability and success of the project. The relevant metrics would be measured using tools such as Wireshark and the Flutter performance profiler.

2.3. Requirements Analysis

At the start of the project I identified a set of all possible features I could implement. To guide my implementation I then performed a MoSCoW[2] priority analysis on these features. Some features were only identified during development, due to unexpected issues or through understanding the problem better. I performed a priority analysis on these features as they arose and then incorporated them into my development plan accordingly. Table 1 tabulates all these features along with their corresponding priority.

[2] **M**ust have, **S**hould have, **C**ould have, **W**on't have.

MoSCoW Priority	Features
Must have	State machine replication; Client-server networking model; Client-server model with tolerance against misbehaving clients
Should have	Rollback; Hiding input latency; Hiding opponent latency; Minimal assumptions about client hardware and network infrastructure; Low server bandwidth and processing; Easy to modify code
Could have	State interpolation; Multiple games per server; Efficient rollback; Abstracted networking; Re-connecting to a game; Realistic car turning physics; Realistic car resistance; Non-static map; 3d renderer; Compilation to mobile; Precise latency compensation
Won't have	Shared computation; UDP transport; Reputation system; Mobile tilt controls; Mini-map; Detailed graphics and animation; Hybrid networking model; Hybrid model with tolerance against misbehaving clients; Offline game verification

Table 1: MoSCoW priority analysis of features considered for this project.

2.4. Engineering Approach

2.4.1. Software Development Methodology

To best suit the different styles of development needed for different parts of the project, I used a combination of the Agile and Waterfall software development methodologies.

Agile was used at the start of the project when implementing DAN's core functionality. At this phase I was making key design decisions which would affect the rest of the project, so I needed to be able to pivot quickly if an earlier decision had to be changed. Because DAN was designed to be used by other developers, its API had to be as user friendly as possible. Therefore, during this phase I developed a test game alongside DAN to help me view the API from a game developer's perspective.

Once the key design decisions had been made, I switched to the Waterfall methodology alongside Test Driven Development. Since the architecture was mostly fixed at this point, the majority of the remaining work could be planned out in advance.

2.4.2. Testing Strategy

Good tests were important for DAN's development for two reasons. Firstly, since DAN was designed to be used by other developers, it needed to be high quality and free of bugs. Secondly, unit tests were an important metric for measuring and evaluating the success of the project.

To help with testing, DAN was designed to be as modular and decomposable as possible. Each module's dependencies were made explicit, which made stubbing, mocking, and faking easier.

I used the industry-standard `dart:test` library for unit testing. The complete set of unit tests can be found in the code repository, and fragments are listed throughout the Evaluation (Section 4).

Once I had a minimum running project I began basic integration and end-to-end testing. As part of this I developed an artificial network latency utility, which allowed me to test the latency compensation modules while running the clients and server locally. The rendering module (Section 3.3.3) also acted as additional visual validation.

2.4.3. Licensing, Tools, Libraries, and Backups

I used Android Studio and IntelliJ IDEA Ultimate (with Vim bindings), since they are the industry standard for Dart and Flutter. I used Git as my version control system, and GitHub as a remote repository for backups. I also made weekly backups of the entire project onto an external SSD and a Google Drive account.

I made sure to check the software licenses for all SDKs and libraries before I used them. These are tabulated in Table 2.

SDK / Library	License
Dart SDK	BSD-3-CLAUSE [29]
Flutter SDK	BSD-3-CLAUSE [30]
dart:async	BSD-3-CLAUSE [31]
web_socket_channel	BSD-3-CLAUSE [32]

Table 2: All SDKs and libraries used in this project, along with their licenses.

2.5. Starting Point

My starting point was the same as stated in the project proposal (Appendix C), with the exception of the 3D rendering library. I will expand upon the description provided in the proposal and explain the discrepency below.

I started the project with a small amount of experience in Dart and Flutter. This was from having developed a few basic mobile applications for personal learning purposes. I also had a small amount of experience developing client-server applications with NodeJS and JavaScript.

The only relevant theoretical knowledge I had at the start came from the following Tripos courses: *Introduction to Graphics*, *Further Graphics*, *Computer Networking*, and *Concurrent and Distributed Systems*.

The proposal stated that I had partially developed a 3D rendering library before the project started. This was true, however, after gaining a better understanding of how Flutter worked, I completely re-developed the library. Therefore the code mentioned in the proposal did not contribute to this project.

I built the project upon the pre-existing Dart and Flutter SDKs and relevant libraries, however all code beyond that is my own. The code repository and repository overview (Section 3.4) only contain code that I have written.

3. Implementation

This chapter first describes the design and implementation of the three main parts of the project. These complete success criteria 1–5. Section 3.1 describes building the client-side code for DAN. Section 3.2 then describes building the corresponding server-side code. Together, these two parts make up the complete DAN library. Section 3.3 then describes using DAN to build a demonstration implementation game used later for evaluation. The overall architecture of the demonstration game is shown in Figure 2.

The rest of the chapter consists of a brief overview of the code repository in Section 3.4, and a summary of the project's contributions to the field in Section 3.5.

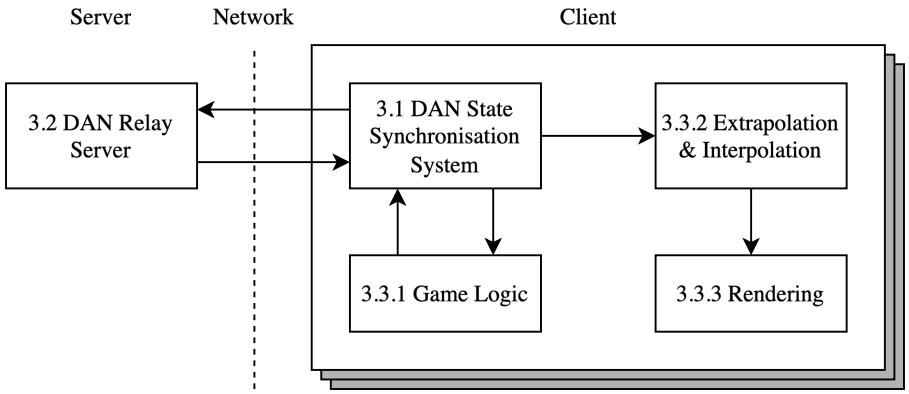


Figure 2: Software architecture of the demonstration game build with DAN.

3.1. Distributed State Synchronisation System

As explained above, DAN consists of two major parts: the client-side code and server-side code. This section discusses the client-side code, which completes success criteria 1.

This part of DAN is focussed on state replication. The "3.1" box in Figure 2 is expanded into Figure 3, showing that the state synchronisation system is made up of five separate parts. Each of these parts will be discussed next.

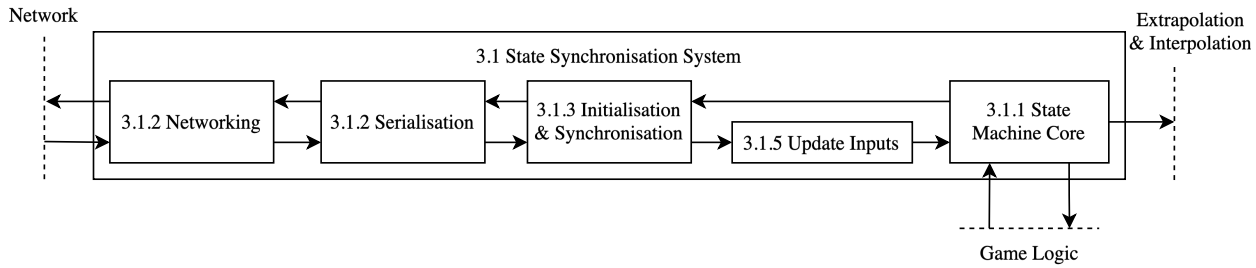


Figure 3: Software architecture of DAN's state synchronisation system.

3.1.1. State Machine Core and Rollback

At its core DAN is a distributed state replication system. In an online multiplayer game there will be many clients playing together. DAN's job is to make sure all of these clients are seeing the same things happen in the game at the same time.

The replication technique I developed is similar to SMR. Usually SMR is used to synchronise states between replicated servers in a distributed network. The servers may also occasionally perform consensus checks in case any states have erroneously diverged. However in DAN, the state is being

replicated across the clients, and since any state divergence would be caused by third party modifications to client-code, no consensus is ever used.

In DAN, the state machine encodes the entire logic of the game. The game developer using DAN would spend most of their time developing this state machine. To run the game, the developer simply gives DAN the state machine.

Usually in SMR, the inputs to the state machines are updates or messages which have been broadcasted to all other servers. In DAN, the broadcasted messages are actions taken by clients. For example, button presses, mouse movements, or touching the screen. These are the atomic unit of change in DAN, and are called *inputs*.

The *state* of a state machine in DAN represents all information about the game at a specific point in time. Figure 4 demonstrates this with a game of noughts and crosses. In this sense, the state machines in DAN are closer to pure functions than finite state machines, since the state is separable from the state machine.

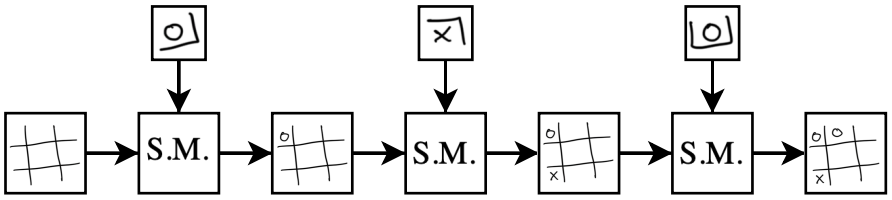


Figure 4: Example of using a state machine to represent the game of noughts and crosses. The 3×3 grid at a particular moment is the state, and an action to add a new nought or cross is an input.

It is possible for any game to be represented by a state machine, not just games as simple as noughts and crosses. For example, consider a fast-paced racing car game such as those in the *Mario Kart* series. The state would contain the instantaneous positions and velocities of every car, as well as any power ups or damage to the vehicle. The inputs would represent changes to the player's controller, such as pressing or releasing the accelerator, or a turn of the steering wheel.

I tested my implementation of SMR with a simple test game. The game worked well, but adding artificial network latency caused the a significant reduction in the game's simulation quality. This is because DAN's form of broadcast required that clients received inputs in the order the inputs were produced — a form stronger than total order broadcast. The variable latency caused inputs to arrive in different orders across clients, and holding back out-of-order inputs caused extreme inconsistencies in the game's flow of time.

To overcome this, I relaxed the broadcast's consistency model from strict to optimistic. I combined this with a system similar to database rollback with transaction logs [14]. In DAN, the transactions are the inputs, and the transaction log is known as the *input list*. Now when an input arrived out-of-order, the state would be rolled back and the input inserted into the correct position in the input list. The state machine would then re-compute from the rolled back state back to the present.

Rollback worked well, but had issues of its own. This was because the input list was unbounded. The first problem was that the input list could use an unbounded amount of memory on the clients. The second problem was that a misbehaving client could force an arbitrarily large rollback, which could leave the other clients unresponsive as they performed the rollback. The final problem was that arbitrarily large rollbacks would also be an opportunity for cheating. I solved these problems by implementing the *acceptance window*. The acceptance window was a moving window of time

which defined the maximum age an input could be when it arrived at the server. By discarding inputs past a certain age, there was now an upper bound on how far could be rolled back from the present. This removed the possibility for forced unresponsiveness and cheating. The acceptance window also enabled clients' memory usage to be bounded. Because the input list could only be rolled back so far, there was now a point beyond which the sequence of inputs could never change. I utilised this to implement a system similar to database checkpoints. Once an input in the input list left the acceptance window it could no longer be changed. It therefore would be removed from the input list and saved into a rolling checkpoint of the state.

3.1.2. Networking and Serialisation

DAN's communication with other clients is routed through a server, often over the internet. I chose to use the WebSocket protocol for transport and JSON for serialisation. Since Flutter had existing cross-platform support for these, it sped up the implementation significantly.

These modules completely abstracted away the networking from the rest of the project, meaning any internal details could easily be changed. The only assumption I made was that the transport was ordered and reliable.

3.1.3. Initialisation and Synchronisation

After the WebSocket channel is initialised, clients need to perform a brief setup with the server before they can begin broadcasting and receiving inputs with other clients. The setup included assigning a unique client ID number, configuring the acceptance window's parameters, and synchronising the client and server's clocks.

I implemented a pair of modules for each setup task: one for the client and one for the server. The pairs communicated directly with each other similar to that of layers in the IP stack. There was a lot of duplicate code patterns across the modules which I extracted out into two utility classes called `StreamInserter` and `StreamInterceptor`. These classes were used to easily write asynchronous, back-and-forth communication over a `Stream`. For example, part of the time synchronisation module looked like:

```dart
// time_client.dart
inserter.add('time sync start time');

final responseData =
  await interceptor.waitUntil(
    (data) => data.startsWith(
      'sync start time:'
    ),
    timeout: setupTimeout,
    passThrough: false,
    name: 'Sync Start',
);

startTime = DateTime.parse(
  responseData.splitAfterFirst(':')
);
```

```dart
// time_server.dart

interceptor.whenever(
  (data) => data == 'sync start time',
  (data) {
    inserter.add(
      'sync start time:${getStart()}',
    ));
  },
  passThrough: false,
);
```

3.1.4. Input Latency

Similar to Section 3.1.1, adding artificial network latency revealed another problem. With network latency, all inputs needed to travel to the server and back before being processed by a client's state machine. This meant all player actions would be delayed by the client-server RTT.

To solve this, I created a new type of input called a *local input*. Local inputs served as a placeholder to sit in the client's input list until the server input returned. Once the server input had returned, the placeholder would be removed and the server input inserted in its place.

The new type of input was needed to differentiate between inputs which had been server confirmed and those which were placeholders. Confirmed inputs could checkpointed (as defined in Section 3.1.1) safely, but local inputs could not. I proved that if a local input ever reached the point of checkpointing it meant the server had rejected the input. This would be because the client would have received the server confirmation before that point. Therefore, local inputs were simply removed from the input list at the point of being checkpointed.

3.1.5. Update Inputs and the Game Loop

Because DAN used state machine replication, games would only visibly update when an input was received. This was an issue for real-time games, which needed to be constantly updating even if there were no inputs. Consider a racing game for example. A car moving at 70mph should continue moving even if the player isn't turning the steering wheel or adjusting the accelerator to produce any inputs.

To solve this, I first examined why non-SMR games avoid this issue. The answer was because most games are run from a single while-loop, as shown in Listing 1. This while-loop continues to run even if there are no user inputs.

```
1: while gameRunning do
2:     inputs ← readUserInputs()
3:     updateGameWorldAndPhysics(inputs)
4:     updateScreen()
```

Listing 1: Pseudocode of the inner-most while-loop which most non-SMR games use.

From this, I considered a few different approaches. Firstly, I could add a while-loop similar to Listing 1 into the state machine core from Section 3.1.1. Secondly, I could have the state machine run every time the screen requested an update. Neither of these solutions were sufficient to solve the problem, however. The first solution meant hard-coding a loop into DAN, which would stop DAN from being completely generalisable and game agnostic. The second solution meant tying the rate of computation to each device's screen update rate. This would be both non-deterministic and cause a huge backlog of computation if the game were ever minimised and re-opened.

I eventually came up with an ideal third solution which allowed for regular deterministic updates without modifying the state machine core. The solution was to add a module which regularly inserted computer-generated inputs into the state machine. I called these inputs *update inputs*. This way the state machine would continue updating even if no player inputs were received. The update inputs were produced deterministically and locally by each client, and not sent to the server. I therefore needed to separate the update inputs into a new type of input called *local-shared inputs*.

The introduction of update inputs solved the problem, but occasionally caused clients' states to diverge. The issue was caused by some clients' clocks being slower than the server's clock. When this happened, the slower clients would produce update inputs which were increasingly late. Eventually one of the update inputs would be produced outside of the acceptance window and be discarded. The slow client would then have a permanently diverged state from the rest. To solve this, I made the update input module scan all incoming events. If it realised it had missed an update input it would quickly insert it in front.

3.2. Relay Server

This section describes DAN's server-side code, which completes success criteria 4. The server was required to broadcast any inputs which arrived in the acceptance window onwards to all clients. This kept the server simple to implement and cheap to run.

To let clients join games after the game had already started, the server also needed to keep a copy of all broadcasted inputs. Then when a client joined, the server would first let them re-compute up to the present game, and then start sending any new inputs. I built a generalisable utility class for this called `StreamSplitAndBuffer`. It automatically handled buffering data and leaving and joining clients.

One final point to make is that because the server only deals with inputs and is independent of the actual game, the same server code can be re-used for all DAN games.

3.3. Creating the Game

Success criteria 2, 3, 5, and 6 revolved around using DAN to build a demonstration implementation of a game. This meant combining the client and server code from Section 3.1 and Section 3.2 with three new modules, which will be described next. To finish, I then describe the process of combining these modules together into the final game.

3.3.1. Game Logic

As explained in Section 3.1.1, the actual game logic is represented by a state machine. The developer using DAN creates this state machine and gives it to DAN to be run. I chose a basic multiplayer car driving game for my demonstration implementation.

Compared the the earlier example of noughts and crosses (Figure 4), implementing a state machine for a game was slightly more involved. As expected it needed to be able to update the cars' positions and physics whenever it received an update input. On top of this, it also needed to be abel to add or remove players when clients connected or disconnected, and buffer player inputs until the next update input arrived.

I defined four strict requirements that any state machine in DAN would have to follow, and used these when developing my game. DAN requires that a state machine:
1. is a pure function,
2. does not modify any input parameters,
3. takes in a state and an input to return a new state, and
4. is identical for all clients.

I was particularly careful with requirements 1 and 4. Requirement 1 is easily violated by naive use of random number generators, and requirement 4 is easily violated when running a game across different hardware. A reference implementation for a random number generator for DAN can be

found in the appendices (Appendix B). To avoid violating requirement 4, I studied how Flutter compiles to different architectures [33]. For my game, I chose to make sure I always ran the clients on the same architecture.

3.3.2. Interpolation and Extrapolation

These modules solved several unrelated problems at once. They act as a pre-processor for the state, slightly modifying the state before it's shown on the screen. The extrapolation stage happened first, which was followed by the interpolation stage.

The first problem was that the screen would only update when the state machine did, even if the screen had a higher maximum update rate. This would lead to a poor user experience for players who are used to a higher screen update rate. The second problem was that when a rollback occured there was a visible discontinuity when the cars 'teleported' to their new locations. This could be described as the butterfly effect; a small change to the past compounds to a bigger change in the present. This was also related to the third problem, where a similar discontinuity would occur when a client had an input discarded by the server.

The first pre-processing step in the solution was extrapolation. This converted the state machine's output from discrete to continuous updates. The extrapolation was only used in between computations from update inputs, so a simple linear model sufficed. This enabled support for screens of arbitrary update rates.

The second pre-processing step was interpolation. This removed any discontinuities from rollback or input discarding by averaging over the states in a small sliding window. I implemented this with a simple weighted mean, using a cubic curve to prioritise the most recent states. I also experimented with different sliding window sizes and interpolation curves. A larger sliding window and flatter curve made the game feel less responsive, but for players with high latencies a small window failed to properly hide the discontinuities. I therefore chose interpolation parameters which found a good balance between both extremes.

The two pre-processing steps also solved problems which the other introduced. Firstly, the interpolation added a slight delay to the game on the screen, but the extrapolation balanced this by extrapolating further into the future. Secondly, the extrapolation added new discontinuities when it reset after receiving the next update input. This was hidden by the interpolation module.

I experimented with different combinations of extrapolation and interpolation modules, but these two had the best balance of responsiveness, lack of visual discontinuity, and simplicity to implement.

3.3.3. Displaying the Game

The final feature I needed to implement was to display the game on the screen. I will refer to the module which does this as the *renderer*. At a high level, the renderer needed to be able to convert any state into a set of graphical instructions. These instructions are then sent to Flutter's underlying C++ renderer [34]. I first implemented a basic 2D renderer, but later implemented and integrated a 3D renderer as one of my extensions. The 3D renderer was important and allowed me to have a more realistic computation load during the later evaluation. The rest of this section will discuss the implementation of the 3D renderer.

To maximise cross-platform compatability, I used Flutter's built-in `Canvas` class. Because it only supported high-level primitive operations such as `Canvas.drawTriangle`, I needed to use the Painter's Algorithm. Compared to rasterisation which draws one pixel at a time, the Painter's Algorithm

draws entire objects at a time. By sorting the objects from back-to-front, the foreground elements correctly occlude the background elements.

The complete rendering pipeline I developed is visualised in Figure 5. The first step is for the developer to declaratively defines the 3D world by composing objects and various modifiers. Examples of objects and modifiers are `Rect` and `Cuboid`, and `Rotate` and `Shade`. In the next step, the 3D objects are compiled into a tree of primitive 3D shapes, which are usually triangles. Then this tree is sorted back-to-front relative to the camera using custom comparison algorithms. The last step is for the tree to be flattened into a list, and for the shapes to be projected onto the camera plane. This list contains the graphical instructions to be sent to Flutter's underlying renderer.

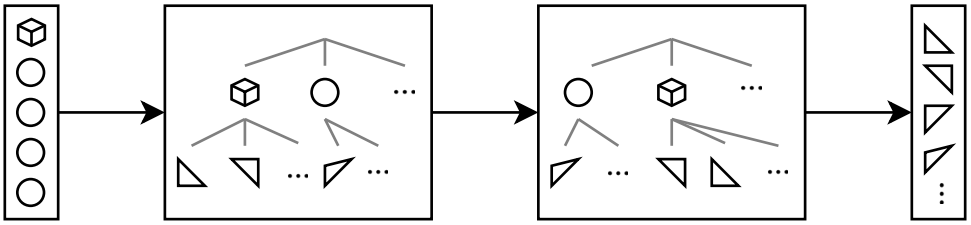


Figure 5: 3D render pipeline for my version of the Painter's Algorithm.

The most complex part of the renderer was the custom comparison algorithms. Determining which triangles occlude others cannot be done with a simple distance check. I therefore needed to identify all possible configurations for a pair of triangles, which is shown in Figure 6. Each of these configurations required its own custom comparison algorithm.

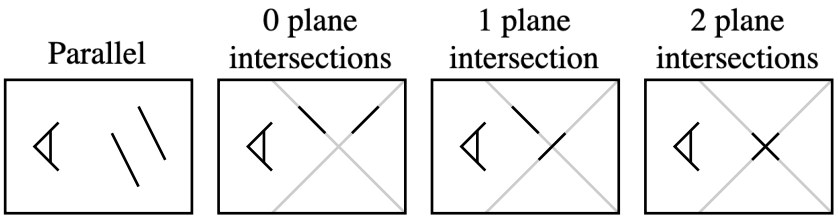


Figure 6: Examples of the four configurations of 3D triangle occlusions. The camera is to the left. The black lines are the triangles from a top-down, orthogonal view. The grey lines represent the infinite planes which each triangle sits within. Excluding the parallel configuration, the difference lies in how many of the triangles lie on the line of intersection of the two infinite planes.

Combined with further optimisations and testing, the library could successfully render shaded 3D scenes in real-time. Figure 7 shows a minimal example.

```
ShadeModifier(
  ambientLight: Colors.white.withOpacity(.2),
  directionalLights: [
    (Vector3(-.5, 1, -2), Colors.white),
  ],
  child: RotateModifier.aroundAxis(
      axis: CartesianAxis.positiveZ,
      rotation: pi * 0.25,
      child: CubeObject.fromSize(
        size: 1,
        colour: Colors.red,
      )))
```

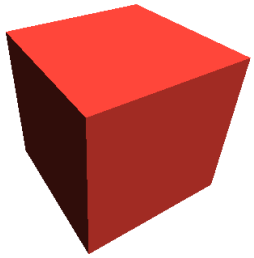


Figure 7: Example of my declarative 3D renderer. It creates the scene by composing a cube object with rotation and shading modifiers.

3.3.4. Combining Into a Game

The final step to produce the game was to combine all the previous modules. The modularity and extensive testing of the previous code made this very easy, and the integration worked on the first try. Figure 8 shows the end result with my best attempt at creating a 3D car.

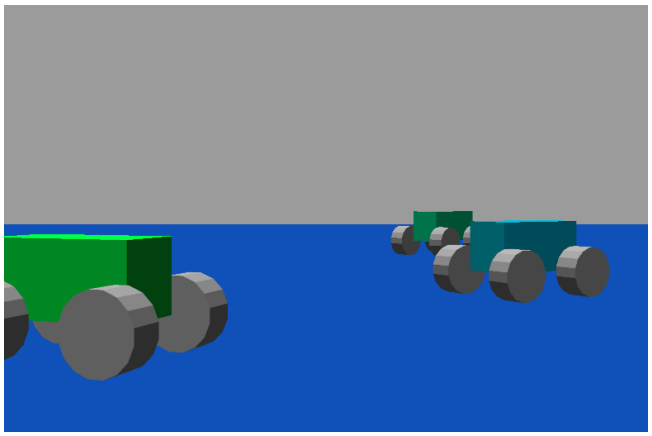


Figure 8: Screen capture of the final game.

3.4. Repository Overview

Directory	Description
`dan/`	Code for the game engine library DAN
`dan/src/core/`	State machine core (Section 3.1.1) and server relay (Section 3.2)
`dan/src/events/`	Server inputs, local inputs (Section 3.1.4), and local-shared inputs (Section 3.1.5), for both the clients and the server
`dan/src/extensions/`	Additional modules, such as networking and serialisation (Section 3.1.2), initialisation and synchronisation (Section 3.1.3), and update input generation (Section 3.1.5)
`dan/test/`	DAN unit tests
`game/`	Client code for the demonstration game
`game/src/main.dart`	Entry point for the game

`game/src/state/`	The state machine which encodes the game (Section 3.3.1) and the interpolation and extrapolation modules (Section 3.3.2)
`game/src/ui/`	The 2D renderer, 3D rendering library glue code, and input handling
`game/test/`	State machine unit tests
`server/`	Server code for the demonstration game
`server/main.dart`	Entry point and logic for the demonstration game server
`render/`	3D rendering engine library (Section 3.3.3)
`render/scene_items/`	Objects and modifiers to declaratively define 3D worlds
`render/sorting/`	Algorithms for sorting 3D triangles relative to a camera

3.5. Contribution to the Field

As planned, I have contributed a game engine for developing online multiplayer games (DAN), as well as a demonstration implementation of a game using this game engine. The game engine is based on SMR with a client-server networking model.

As far as I am aware, DAN is the first general-purpose game engine to use SMR. DAN's generalisability and modularity makes it easily suitable for many types of games, and the use of SMR means that developers do not need to:

- write any networking code,
- spend lots of money running expensive servers,
- worry about misbehaving or cheating clients, or
- worry about implementing complex latency compensation techniques.

The impact of this is far reaching beyond the Tripos. DAN makes developing misbehaviour tolerant, high-quality, online multiplayer games significantly more accessible. For amateur game development, DAN provides a low-cost and easy-to-use entry point for creating networked games. DAN could also be used for single player games, with the option to add online multiplayer networking with almost no additional effort. For professional game development, DAN could save a large amount of time and money when developing and running new online multiplayer games.

4. Evaluation

As discussed in Section 2.2, this project was evaluated through unit testing, end-to-end testing, and data capture and analysis. I first evaluate the client-side code, server-side code, and demonstration implementation game. This is followed by the collection, analysis, and evaluation of various metrics from the demonstration game. Finally, I evaluate the success of any completed extensions.

Some evaluation sections include analyses of the demonstration game's state over time. To increase the consistency of the evaluation, and allow for easier comparisons, I developed a system to automatically record and replay player inputs. Since DAN is completely deterministic, this allowed for identical simulations over many experiements. This is also the reason that the graphs do not contain any error bars, as there is no variation between the experiments.

4.1. Distributed State Synchronisation Data Structure

To evaluate the functional success of this module, I implemented unit tests for every required behaviour. The results of the unit tests can be found in Listing 2. I will provide an overview and explanation of the most important tests below.

```
dart test -r expanded test/**/client_test.dart test/extensions/*_test.dart
00:00 +0: Late events are sorted correctly: timestamp, then senderID, then eventID
00:00 +1: Events replace event with same ID
00:00 +2: Future events are ignored
00:00 +3: Server and local-shared events are unstable, then baked when server event with timestamp is received
00:00 +4: Local events are only added to stable state if they receive server confirmation
00:00 +5: Only local events are sent to the server
00:00 +6: getStateAt returns correct state in future at correct time
00:00 +7: initialise sends "time sync start time" and "time sync clock offset"
00:00 +8: getTime is close (within 50ms) when using same clock
00:00 +9: getTime is close (within 50ms) when client clock is ahead by 1s
00:00 +10: getTime is close (within 50ms) when client clock is behind by 1s
00:00 +11: test that async exceptions can be tested
00:00 +12: Throws FormatException if start time response malformed
00:00 +13: Throws FormatException if clock offset response malformed
00:00 +14: waitUntil triggers exactly once
00:00 +15: whenever triggers multiple times
00:00 +16: Correct trigger order with multiple accepting tests
00:00 +17: waitUntil timeout throws error
00:00 +18: waitUntil timeout cancels trigger
00:00 +19: waitUntil and whenever passThrough false prevents later added triggers
00:00 +20: waitUntil and whenever passThrough false prevents propagation out of stream
00:00 +21: waitUntil and whenever passThrough true allows later added triggers
00:00 +22: waitUntil and whenever passThrough true allows propagation out of stream
00:00 +23: Runs unstable events into returned state
00:00 +24: Ticks have exact correct generated timestamp and correct data
00:00 +25: Tick inserted if timer callback drifts from game time
00:00 +26: Non-tick events pass through correctly
00:00 +27: A tick will be inserted if a *server* event arrives before it should have been inserted
00:00 +28: All tests passed!
```

Listing 2: The unit tests for DAN's client-side code, with 100% passing.

Late events are sorted correctly — When an out-of-order input arrives at the client, the client must rollback to the correct position to insert the input. This test verifies this by using a stub to insert pre-shuffled inputs, and validates that the module's internal order is as expected.

Events replace event with same ID — Local inputs are replaced by their server confirmation input when it arrives. This is done by having new inputs replace existing inputs if they share their ID. This test verifies this by using a stub to insert a sequence of inputs, some with identical IDs, and validating afterwards that the correct sequence of inputs remains.

Server and local-shared events are unstable, then baked... — Once an input in a client's input list leaves the acceptance window it is ready to be checkpointed. Checkpointing can only happen

once the server confirms the necessary time has passed. This unit test verifies that inputs are checkpointed when, and only when, the correct server confirmation arrives. This is done by inserting a sequence of inputs and server confirmations into the module, and validating that the input list and checkpoint are as expected at every point.

Local events are only added... — At the point a local input leaves the acceptance window, it will either be discarded or checkpointed depending on if it received a server confirmation. This test verifies this by inserting a sequence of local inputs and server confirmations, and validating that each local input is corrected discarded or checkpointed.

These unit tests demonstrate that this module functions as required, meaning I have met success criteria 1.

4.2. Relay Server

This module was also evaluated via unit testing. The results of the unit tests are shown in Listing 3, with an overview and explanation of the most important tests below.

```
% dart test -r expanded test/core/server_test.dart
00:00 +0: test/core/server_test.dart: Server removes client events past unstable period
00:00 +1: test/core/server_test.dart: Server relays client events to all clients correctly
00:00 +2: test/core/server_test.dart: Server relays server events to all clients correctly
00:00 +3: All tests passed!
```

Listing 3: The unit tests for DAN's server-side code, with 100% passing.

Server removes client events... — A critical part of preventing misbehaviour in DAN is that inputs which arrive outside of the acceptance window are discarded. This unit test verifies that this happens correctly by inputting a sequence of inputs while artificially modifying the acceptance window through a stub. The test passes if the mock client receives exactly the correct inputs from the server.

Server relays client events... — The server's most basic function is to relay events from one client onwards to all clients in the game. This unit test verifies this by mocking a set of clients and validating that each one receives the correct sequence of inputs.

These unit tests demonstrate that this module functions as required, meaning I have met success criteria 4.

4.3. User Interface, Game Logic, and the Demonstration Game

To evaluate if the implementation of these modules was successful, I performed an end-to-end test of the game. This involved running a server and 3 clients locally, and pressing a sequence of inputs at each client one at a time. I considered the modules to be successful if each client could drive its own car around and have the other clients see this on their screens. A sequence of screen captures of this gameplay is shown in Figure 9, which helps to demonstrate that the modules were successful.

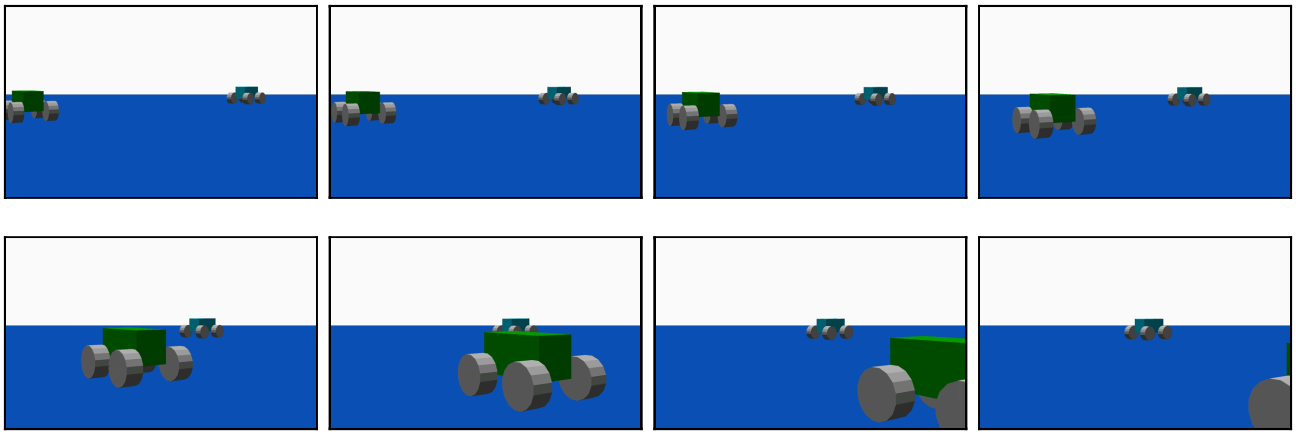


Figure 9: Sequence of screen captures of an end-to-end test of the game.

The only part of these modules' evaluation which isn't demonstrated by Figure 9 is showing that the game logic module is deterministic and that it doesn't modify its inputs. I confirmed these properties with 1000 iterations of fuzzy testing each. The results of the tests are shown in Listing 4.

```
 % flutter test test/* -r expanded
00:00 +0: loading /Users/dan/dev/p2project/code/blitzmania/test/driver_test.dart
00:00 +0: Fuzzy test that driver is deterministic
00:00 +1: Fuzzy test that state is driver doesn't modify state
00:01 +2: All tests passed!
```

Listing 4: Fuzzy tests for the game logic module.

The combination of end-to-end testing and fuzzy testing demonstrates that these modules function as required, meaning I have met success criteria 2, 3, and 5.

4.4. Extensions

4.4.1. Hiding Input Latency

To evaluate the success of this extension, I built a separate version of DAN and the demonstration game with the extension removed. As a baseline metric I recorded the position of the car with zero latency added. I then ran the two separate versions of DAN with 200ms of latency, and tracked the cars. The results are shown in Figure 10.

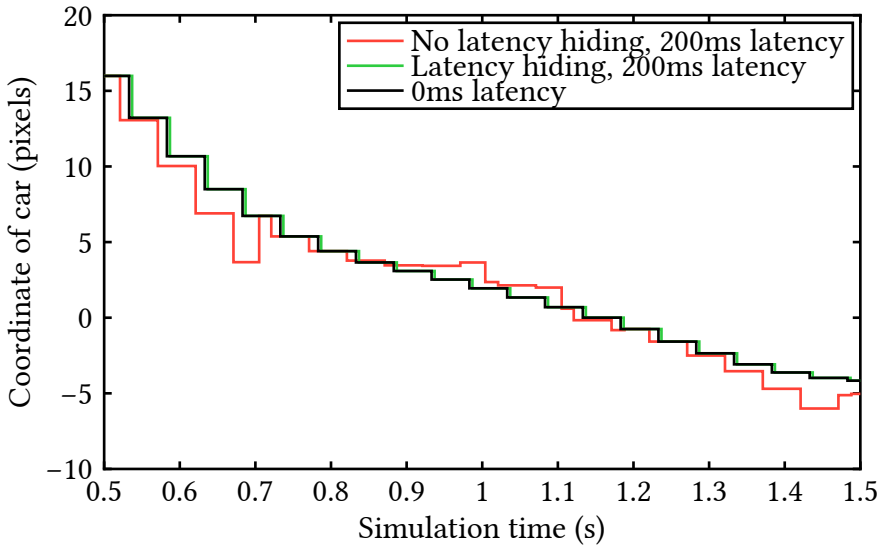


Figure 10: Demonstation of input latency hiding on a client with 200ms of RTT latency

As shown by the red line, when the extension is removed the car visibly re-adjusts once the input arrives back from the server. The reason the car readjusts to the correct position is because DAN still had rollback enabled. Without rollback, the red line would show as being shifted left of the baseline by the latency.

With the extension enabled, the position of the car is identical to the baseline. This shows that the extension functions as required.

4.4.2. State Interpolation

To evaluate the success of this extension, I built a separate version of DAN and the demonstration game with the interpolation extension removed. The tracked positions of the car is shown in Figure 11.

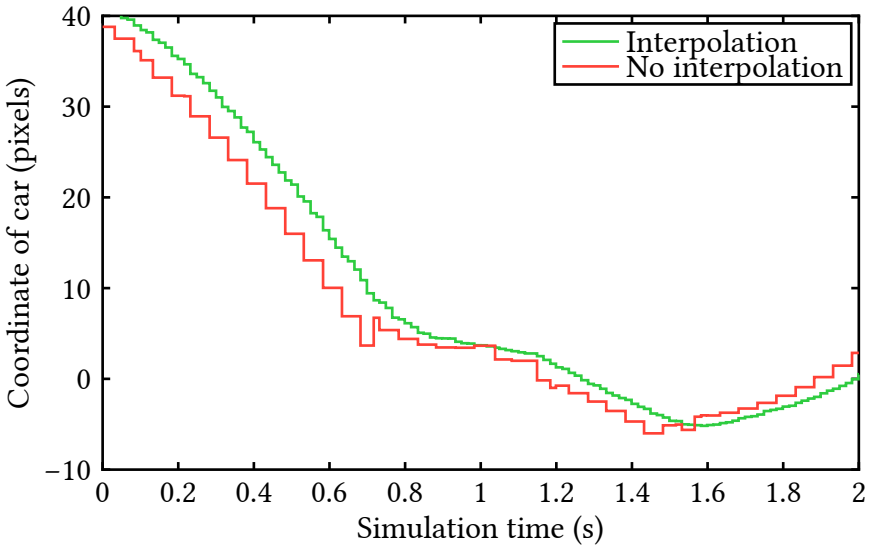


Figure 11: Demonstration of the state interpolation module.

At approximately 0.7s a rollback occured and the car's position was corrected. The black line shows that without interpolation, the rollback is visible as a visual discontinuity. The blue line shows that the interpolation extension successfully hides the discontinuity, and therefore functions as required.

4.4.3. Re-Connecting to a Game

This extension required that clients could connect to a game after it had already started, or that a client could disconnect and re-connect to the same game. To demonstrate this, I ran a server and two clients locally, and recorded the server's console outputs. These are shown in Listing 5.

The test consisted of the following actions:

1. Connect client A to the server
2. Client A produces inputs
3. Connect Client B to the server, after the game had already begun
4. Client B successfully produces inputs
5. Disconnect and re-connect client A to the server
6. Client A successfully produces inputs after re-connecting

```
Listening on 127.0.0.1:4040
time: 12 -- server  : player 1 connected
time: 14 -- player 1: left
time: 14 -- player 1: forward
time: 15 -- player 1: stop
time: 16 -- player 1: forward
time: 18 -- player 1: straight
time: 18 -- player 1: stop
time: 24 -- server  : player 2 connected
time: 27 -- player 2: right
time: 27 -- player 2: left
time: 28 -- player 2: forward
time: 28 -- player 2: straight
time: 32 -- server  : player 1 disconnected
time: 34 -- server  : player 3 connected
time: 36 -- player 3: forward
time: 37 -- player 3: right
time: 40 -- player 3: stop
time: 40 -- player 3: straight
```

Listing 5: Extract from the server's console, showing one client joining late, and the other client re-connecting.

The results in Listing 5 demonstrate that the clients were able to successfully produce inputs and play the game in both scenarios, meaning the extension functioned as required.

4.4.4. 3D Renderer

This extension was required convert a game state into a 3D graphical representation. While Section 4.3 has already demonstrated this extension's success, it did not show the extension's capabilities as a general purpose 3D library. Figure 12 demonstrates that the extension is able to render a variety of 3D scenes with simple lighting, although it does not support shadows.

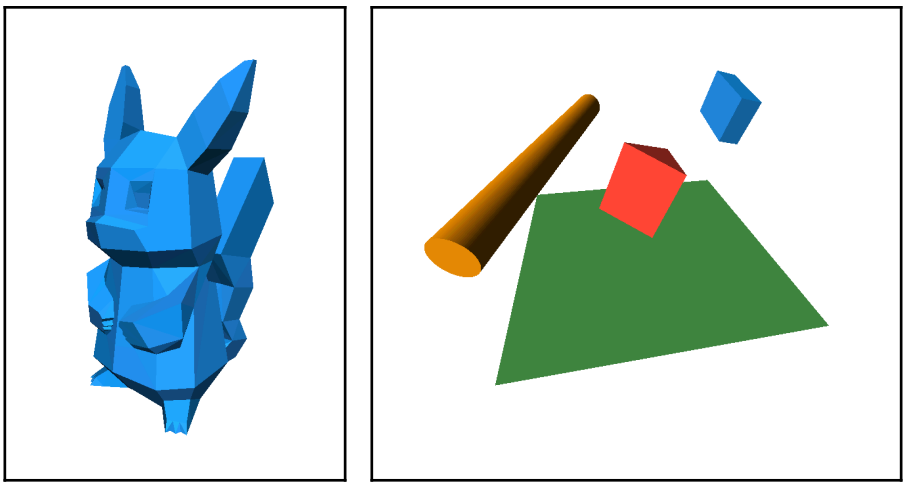


Figure 12: Demonstration of my 3D renderer with varying scenes.

4.5. Practicality Evaluation

The final part of the project was to evaluate the practical success of DAN, using metrics such as visual update rate and bandwidth usage. This section completes success criteria 6.

4.5.1. Ease of Use

As defined in Section 1.3.1, implementing an online multiplayer game is considered simpler if there is less networking logic for the developer to write.

This is demonstrated by inspecting the project's code in the provided repository. The file containing the game logic which the developer would write is found at `/game/src/state/driver.dart`, and contains no use of any networking APIs.

I also exceeded this evaluation criteria by abstracting the networking away from all modules in DAN. Further inspection of the repository will show that only the two files in `/dan/src/extensions/networking/` work with any networking APIs.

4.5.2. Server Performance

One of the benefits of using SMR in DAN is the reduced server bandwidth usage compared to passive replication. To compare DAN against a passive replication system, I built a lightweight server which would receive inputs from clients while sending back each of them the live game state. Because the pattern of inputs into an SMR system affects its bandwidth usage, I ran four different experiments and repeated each one ten times.

During each run of an experiment, identical inputs were sent to both servers, with the resulting network traffic then being analysed in Wireshark [35]. Figure 13 shows the total data transferred between the client and server, excluding any protocol setup or teardown.

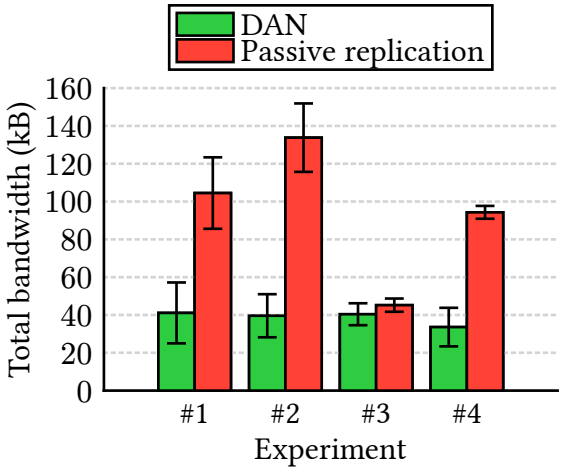


Figure 13: Experimental bandwidth usage between DAN and an equivalent passive replication model. Error bars show ±1 standard deviation.

DAN outperformed the passive replication system on every experiment, including experiment #3 which was designed to maximimse DAN's bandwidth output.

4.5.3. Client Performance

As mentioned in Section 1.3.2, one of SMR's drawbacks is its lower maximum number of players in a game compared to passive replication. This evaluation metric is therefore important to determine how significant of a limitation this is in DAN.

To test this, I used the Flutter performance profiler [36] to measure the average screen update rate with different numbers of connected clients. I ran each test with both 2D and 3D renderers for 10 seconds, with the results shown in Figure 14.

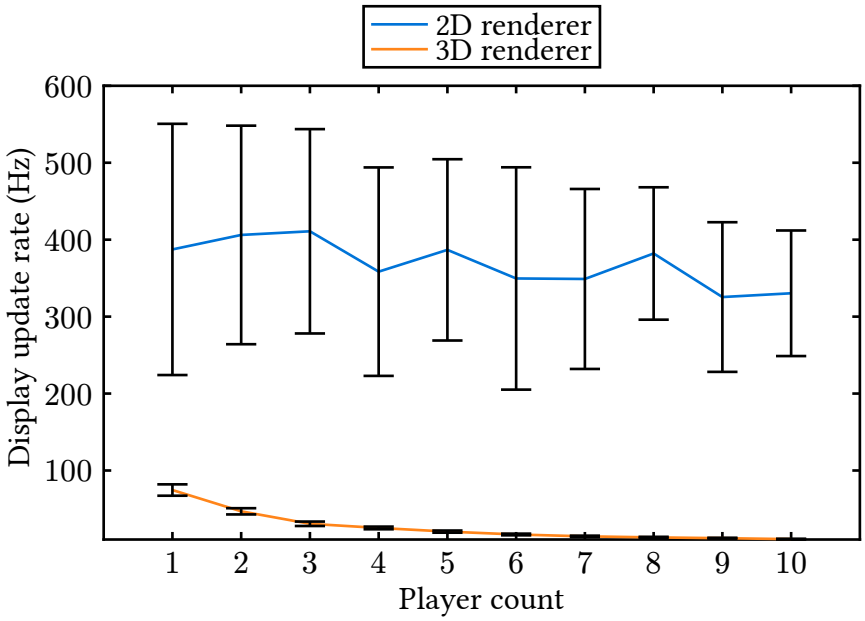


Figure 14: Frequency of client screen updates per number of players. Error bars show ±1 standard deviation.

When performing the test, I ran the clients and server locally on the same machine, with all but one of the clients' applications minimised. This means that the processing load is exaggerated in the results. Despite this, however, the results indicate that the majority of the processing was due to the 3D renderer, with only a slight decrease in update rate for the 2D renderer between 1 and 10 players.

This suggests that games with high player counts (beyond 10) are possible with DAN, as long as the renderer is well-optimised and the game logic is not too complex.

4.5.4. Quality of Replication

Game developers are more likely to use DAN if it is successful in hiding the effects of network latency. I will be evaluating two metrics: firstly, DAN's ability to support arbitrary visual update rates, regardless of the underlying update input rate; and secondly, DAN's ability to hide the visual discontinuities caused by rollbacks.

To evaluate these metrics, I ran the demonstration game and tracked the car's position across latencies of 0ms, 200ms, and 400ms. For each latency, I also separated out the individual effects of the extrapolation and interpolation modules.

The blue line in Figure 15 demonstrates DAN's ability to adapt to any visual update rate. The extrapolation module converts the 20Hz underlying update rate into an approximately 60Hz visual update rate.

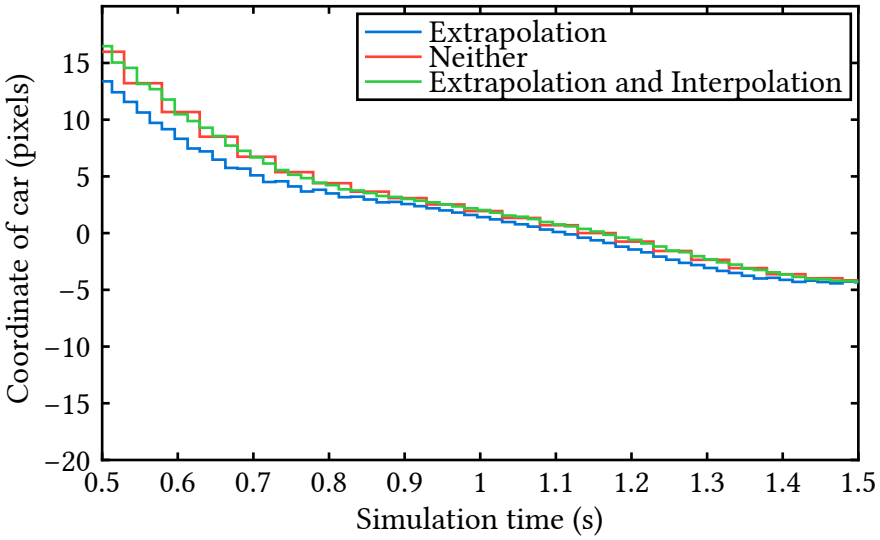


Figure 15: The position of the car over time, with 0ms of network latency and a 100ms interpolation window.

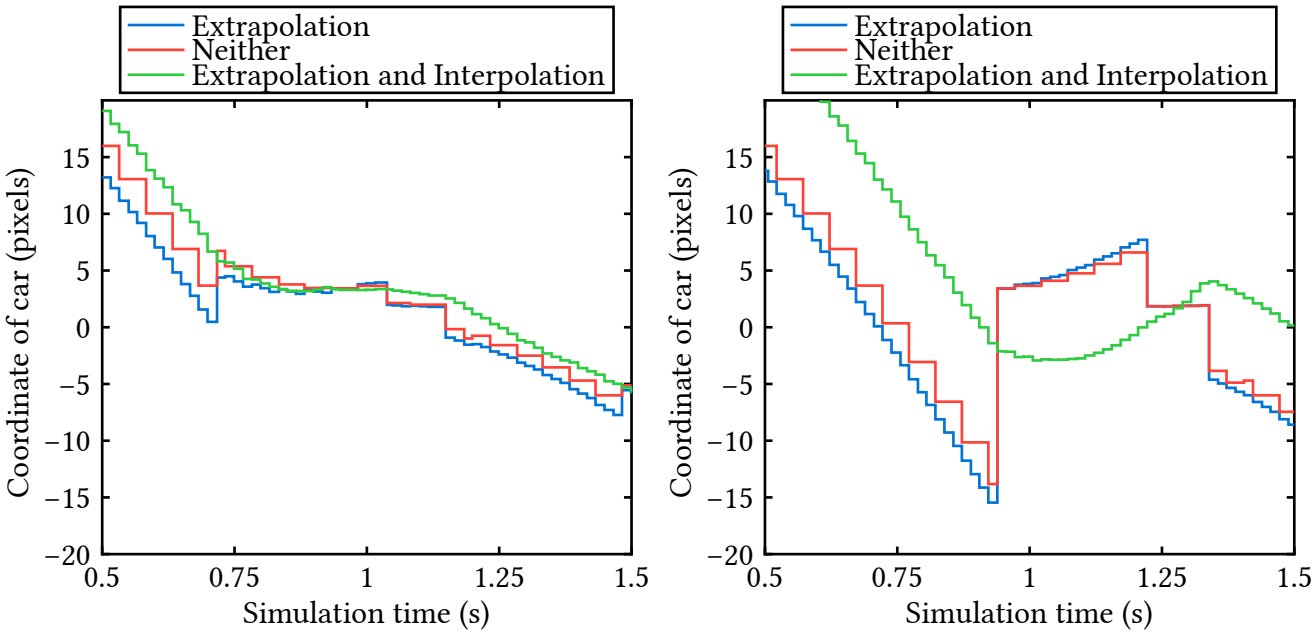


Figure 16: The position of the car over time, with RTT network latencies and interpolation window sizes of 200ms (left) and 400ms (right).

The green lines in Figure 16 demonstrate that DAN's interpolation module is able to hide roll-back discontinuities in clients of varying network latencies. The only downside is the increased discrepency between the actual and displayed positions of the car, although this could likely be resolved with further fine-tuning of the extrapolation and interpolation modules' parameters.

5. Conclusions

I defined six success criteria for the project, which were:

1. Develop a distributed state synchronisation data structure.
2. Develop a user interface module to accept user input and render the game.
3. Develop a game driver module, which provides the logic to 'step forward' and run the game.
4. Develop a server module to relay events between clients.
5. Combine the modules into a running game.
6. Evaluate the practical success of the project using metrics such as framerate and synchronisation quality (deviation from true state, rate of jitter), under conditions such as varying player count, latencies, and bandwidth.

Through the implementation in Section 3.1 and evaluation in Section 4.1, I have shown I have developed a distributed state synchronisation data structure, and have therefore met success criteria 1.

The implementation and evaluation of the relay server in Section 3.2 and Section 4.2 respectively demonstrate I have met success criteria 4.

Section 3.3.1, Section 3.3.3, and Section 3.3.4, combined with their collective evaluation in Section 4.3 demonstrate I have built and combined the necessary modules for a demonstration implementation game using DAN. I have therefore met success criteria 2, 3, and 5.

Lastly, The practicality evaluation of DAN in Section 4.5 demonstrates I have met success criteria 6. The practicality evaluation also demonstrated that DAN was easy to use, significantly improved upon passive replication systems in terms of bandwidth usage, supported games with up to at least 10 connected clients, and had an extremely high quality of replication.

It is clear I can claim the project was a success. All initially proposed success criteria were met, many extensions from Section 2.2 were met or exceeded, and the project has a large potential for real-world impact.

5.1. Reflections

I had never used test driven development before this project, and was very pleasantly surprised at how effective it was. While it did take some adjustment to become familiar with writing tests before the implementation, the development of this project went extremely smoothly as a result. There were very few bugs and errors, and in the one instance I needed to use a debugger I ended up finding a bug with Dart itself.

Another lesson learnt was about building a publically usable software library under time pressure. Taking shortcuts during development would have been a big risk, given the potential for poor decisions early on to affect the project negatively later. I learnt to be purposeful with the shortcuts I was taking, and began to develop an intuition for what sorts of tasks ought to be implemented 'properly', and which could suffice with a less optimal solution.

I also began to better understand the importance of the user experience of a software library's API, through developing my demonstration implementation game alongside DAN.

This project also taught me about scope management. This was one of the first times I needed to develop a large software project unguided, for a purpose other than just enjoyment. It meant learning to focus and limit exploration of ideas which did not directly contribute to the project's end result.

My final two reflections came from writing this dissertation. This was the area of the project I was least familar with. I found the style of writing and the 'purpose' of the dissertation difficult to internalise. I also struggled with the idea that ideas and designs for systems are only useful in a dissertation if you have evidence to back them up.

5.2. Future Work

The next steps for DAN will focus on making it a more complete game engine. At the moment DAN is primarily a replication system, but most game engines provide many more features built-in. For example:

- Better graphics, including more abstractions over 2D and 3D rendering, and using more efficient rendering techniques with hardware acceleration.
- Support for audio, which would need custom support due to rollback.
- A built-in physics engine for both 2D and 3D environments, with pre-provided extrapolation and interpolation modules.
- More support for writing deterministic games, such as automatic testers and software implementations of operations which are inconsistent across hardware.
- Documentation, tutorials, and transition guides for developers moving from other game engines.

On top of completeness, there are many parts of DAN which would benefit from being optimised. Examples include:

- Moving to a UDP-based transport protocol, to reduce replication latency and bandwidth usage.
- More efficiently re-computing states after a rollback. Most of the time, only a small part of the state needs to be re-computed. I have designed a dependency-based tree-diff algorithm but which hasn't yet been implemented.
- Making joining an existing game more efficient, since clients currently need to re-simulate all inputs from the start of the game before they can start playing. I have also designed an algorithm for this, but which hasn't been implemented.

The final set of future work for DAN is to improve beyond other existing game engines, as well as tackle some of SMR's fundamental limitations:

- Support for precise latency compensation techniques required in precision shooter games. I have designed an algorithm which enables a game-agnostic way of completely eliminating the effect of latency for certain important player actions.
- Support a peer-to-peer or hybrid networking architecture. DAN's high level abstraction over networking means it would be possible to support multiple different networking architectures from an identical API.
- Support for games with large numbers of players, potentially through sharing computation between clients which trust each other.
- Support for hiding specific information from clients.
- Explore the possibility of relaxing the determinism requirement, in favour of occasional re-synchronisations.

6. Bibliography

[1] Jeff Yan and Brian Randell, "Security in Computer Games: from Pong to Online Poker," Feb. 2005. [Online]. Available: https://api.semanticscholar.org/CorpusID:10329044

[2] Irdeto, "Irdeto Global Gaming Survey: The Last Checkpoint for Cheating," 2018. [Online]. Available: https://resources.irdeto.com/media/irdeto-global-gaming-survey-report

[3] Andy Maxwell, "Bungie & Destiny 2 Cheat Creator Agree 13.5m Copyright Damages Judgment". May 10, 2022. [Online]. Available: https://torrentfreak.com/bungie-destiny-2-cheat-creator-agree-13-5m-damages-judgment-220610/

[4] Lorenzo Franceschi-Bicchierai, "Inside The 'World's Largest' Video Game Cheating Empire." May 01, 2021. [Online]. Available: https://www.vice.com/en/article/93ywj3/inside-the-worlds-largest-video-game-cheating-empire

[5] G. Armitage, M. Claypool, and P. Branch, *Networking and Online Games: Understanding and Engineering Multiplayer Internet Games*. 2006. doi: 10.1002/047003047X.

[6] Lars Doucet and Anthony Pecorella, "Game engines on Steam: The definitive breakdown." Sep. 02, 2021. [Online]. Available: https://www.gamedeveloper.com/business/game-engines-on-steam-the-definitive-breakdown

[7] "Classic server protocol." [Online]. Available: https://minecraft.fandom.com/wiki/Classic_server_protocol#Client_%E2%86%92_Server_packets

[8] "Source Multiplayer Networking." [Online]. Available: https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking

[9] Timothy Ford, *Overwatch Gameplay Architecture and Netcode*, (2017). [OnlineVideo]. Available: https://www.gdcvault.com/play/1024001/-Overwatch-Gameplay-Architecture-and

[10] "Client-Server Runtime." [Online]. Available: https://create.roblox.com/docs/projects/client-server

[11] "Roblox (RBLX) Q4 2022 Earnings Call Transcript." [Online]. Available: https://www.fool.com/earnings/call-transcripts/2023/02/15/roblox-rblx-q4-2022-earnings-call-transcript/

[12] "Compatability Between NAT types." [Online]. Available: https://en-americas-support.nintendo.com/app/answers/detail/a_id/12472/~/compatibility-between-nat-types

[13] "Doom networking component." [Online]. Available: https://doom.fandom.com/wiki/Doom_networking_component#Multiplayer_component

[14] "Concurrent and Distributed Systems, University of Cambridge Computer Science Tripos Part 1B." [Online]. Available: https://www.cl.cam.ac.uk/teaching/2324/ConcDisSys/materials.html

[15] N. E. Baughman, M. Liberatore, and B. N. Levine, "Cheat-proof playout for centralized and peer-to-peer gaming." 2007.

[16] Cameron Cohu, "Rollback Networking in Peer-to-Peer Games." Feb. 26, 2021. [Online]. Available: https://cameroncohu.com/wp-content/uploads/2021/05/Rollback-Networking-in-Peer-to-Peer-Video-Games.pdf

[17] "Unity Netcode for Entities." [Online]. Available: https://docs.unity3d.com/Packages/com.unity.netcode@1.0/manual/index.html

[18] “Unreal Engine Networking and Multiplayer.” [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/networking-and-multiplayer-in-unreal-engine

[19] “About Netcode for GameObjects.” [Online]. Available: https://docs-multiplayer.unity3d.com/netcode/current/about/

[20] “Bevy Engine.” [Online]. Available: https://bevyengine.org/

[21] “Game Creators Club.” [Online]. Available: https://gamecreatorsclub.com/

[22] “Deterministic Lockstep Games With Rollback For The Web.” [Online]. Available: https://www.gamecreatorsclub.com/blog/deterministic-lockstep

[23] “Networked Replication.” [Online]. Available: https://github.com/bevyengine/rfcs/pull/19

[24] “GGPO | Rollback Networking SDK for Peer-to-Peer Games.” [Online]. Available: https://ggpo.net/

[25] Flutter Team, “Flutter 2023 Strategy,” Mar. 2023. [Online]. Available: https://flutter.dev/go/strategy-2023

[26] Theodoros Karasavvas, “Why Flutter is the most popular cross-platform mobile SDK.” [Online]. Available: https://stackoverflow.blog/2022/02/21/why-flutter-is-the-most-popular-cross-platform-mobile-sdk/

[27] “Dart documentation.” [Online]. Available: https://dart.dev/guides

[28] “Flutter documentation.” [Online]. Available: https://docs.flutter.dev/

[29] “LICENSE.” [Online]. Available: https://github.com/dart-lang/sdk/blob/main/LICENSE

[30] “LICENSE.” [Online]. Available: https://github.com/flutter/flutter/blob/master/LICENSE

[31] “LICENSE.” [Online]. Available: https://github.com/dart-lang/async/blob/master/LICENSE

[32] “LICENSE.” [Online]. Available: https://github.com/dart-lang/web_socket_channel/blob/master/LICENSE

[33] Dart Team, “Numbers in Dart.” [Online]. Available: https://dart.dev/guides/language/numbers

[34] Flutter Team, “Impeller rendering engine.” [Online]. Available: https://docs.flutter.dev/perf/impeller

[35] “Wireshark · Go Deep.” [Online]. Available: https://www.wireshark.org/

[36] “Flutter performance profiling.” [Online]. Available: https://docs.flutter.dev/perf/ui-performance

A Systematic Evaluation of Tools

Cross-platform development platforms from the StackOverflow Developer Survey 2021-2023. https://survey.stackoverflow.co/2023/#other-frameworks-and-libraries https://survey.stackoverflow.co/2022#other-frameworks-and-libraries https://survey.stackoverflow.co/2021/#other-frameworks-and-libraries Spring, Flutter, React Native, Electron, QT, Swift UI, Xamarin, Ionic, GTK, Cordova, .NET MAUI, Tauri, Capacitor, MFC, and Uno Platform.

10 most popular game engines from SteamDB and Itch.io each, with duplicates removed. https://steamdb.info/tech/ https://itch.io/game-development/engines/most-projects Unity, Unreal Engine, GameMaker, RPGMaker, PyGame, RenPy, Godot, XNA, Cocos, Adobe Air, Construct, Twine, Bitsy, and Pico-8.

Each requirement has listed the development platforms which were removed because they didn't meet it. Each platform will only appear once, even though it may have failed to meet multiple requirements. There may also be additional reasons for a platform failing the requirement than what is listed after it.

Unmodifiable:
- Unity - closed source
- Unreal Engine - too large and too much existing code
- Godot
- Cocos - closed source

Restrictive:
- GameMaker
- RPGMaker
- RenPy - just for visual novels
- Pico-8
- Adobe AIR
- Twine - just for interactive story games
- Bitsy

Not cross-platform: Must support 2 of the following 4 platforms: console (2 of: Xbox One & X, PlayStation 4 & 5, Nintendo Switch), desktop (Windows, MacOS), mobile (iOS, Android), and web.
- Swift UI - only supports MacOS and iOS
- GTK - only supports desktop
- MFC - only supports Windows

Not performant:
- React Native - JavaScript
- Electron - JavaScript
- Xamarin - JavaScript
- Ionic - JavaScript
- Cordova - JavaScript
- Tauri - JavaScript
- Capacitor - JavaScript
- Uno Platform - JavaScript
- PyGame - Python
- Construct - Python or JavaScript

- Spring - Java
- .NET MAUI - C#

There were two remaining development platforms which met all requirements: Flutter and QT.

B Random Number Generator Reference Implementation

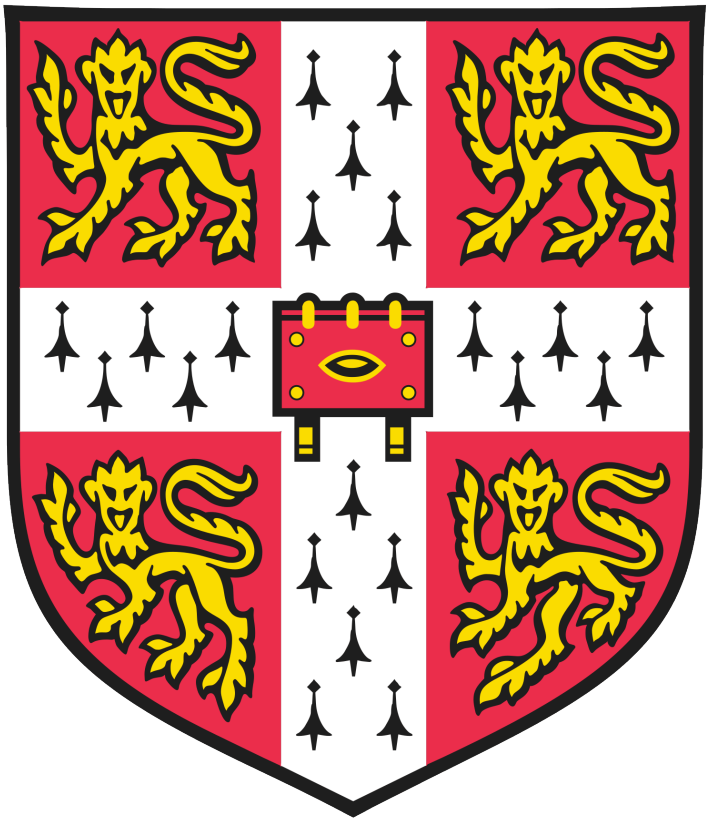
```
import math;
EventClientInInCore Function(State, EventClientInInCore) addRNG<State>({
  required EventClientInInCore Function(
    State,
    EventClientInInCore,
    Random Function(String?) getRNG
  ) driver,
  required int gameID,
}) {
  EventClientInInCore retFunc(State state, EventClientInInCore event) {
    final getRNG = (String? seed) => math.Random(
      gameID ^ event.senderID ^ event.eventID ^ (seed.hashCode ?? 0)
    );
    return driver(state, event, getRNG);
  }
  return retFunc;
}
```

C Project Proposal

The candidate

DAN:
Distributed Anticheat Networking

Part II Project Proposal



2266F

Computer Science Tripos

May 10, 2024

Project Originator: The candidate
Project Supervisor: Prof Andrew Moore
Project Checkers: Prof Alan Blackwell and Prof Srinivasan Keshav
Director of Studies: Dr Ramsey Faragher

Contents

1. Introduction

Video games are one of the biggest markets worldwide, worth more than both the music and movie industry combined [1]. Gaining an unfair advantage in such games is, to no surprise then, extremely desirable; the act of *cheating*.

Cheating in video games is so prevalent that *anticheat* exists, that is, software designed solely for its prevention and detection.

Both cheats and anticheat come in many forms, with the primary focus of research and this project being on online multiplayer games. The more cheaters present in such games will result in fewer (paying) users, reducing profit. Games overrun by cheaters can have their livelihoods destroyed and companies are well aware, some offering up to hundreds of thousands of dollars for finding bugs in their anticheat systems [2].

Anticheat development today is an arms race of machine learning and increasingly intrusive scanning software. But such high level approaches are only pursued under the assumption that a game is already secured at the lower levels, meaning it has anticheat built into its core right from the start.

To elaborate on exactly what "core" anticheat is, lets briefly look at two popular online multiplayer video games *Minecraft* and *Counter Strike: Global Offensive* (CS:GO). Both games feature client-server networking and the ability to move a character around in a virtual 3D environment. When a

user wishes to move their character in Minecraft, the client calculates the new position and sends this to the server [3]. In CS:GO however, the client would instead send the *action* (e.g. "forward key pressed") to the server, and have the server calculate the new position [4].

The difference here is that a cheater in Minecraft could send any position to the server, regardless of what is possible within the game's bounds. CS:GO prevents this because its core game logic does not trust the client to calculate their own movements, instead relying on the server. In short, Minecraft does not implement core anticheat, while CS:GO does.

Despite core anticheat having been in use for many years it is still not widely adopted, with many games suffering unnecessarily as a result. The goal of this project is to help tackle this problem.

To introduce the approach this project will take it is first helpful to understand some of the reasons why many games don't implement core anticheat:

1. **Inexperience:** developers may not know that integrating anticheat within the core game logic can be necessary or how to do it. Often the obvious approach is to add online multiplayer *after* writing the core game logic, as a layer 'on top'.
2. **Game Engines:** built-in anticheat is either non-existent or highly limited and expensive [5], [6], so must be implemented by the developer. But the abstractions in game engines designed to simplify development then make it harder to access the necessary core game logic to integrate core anticheat.
3. **Development Cost:** including this anticheat increases the complexity of development, using more time and money – a resource which is likely already at its limit.
4. **Ongoing Cost:** this anticheat requires servers to be continually running and computing all player movement, which is expensive at scale.
5. **Backwards Compatibility:** updating an existing game's logic may stop different versions of the game working together, which is especially problematic when users and surrounding infrastructure make it difficult for everyone upgrade.[1]

At first glance, a good solution would be to develop a reuseable form of this anticheat, and either integrate it within a popular game development tool or provide it as a base for others to build on. This alleviates the need for developers to have anticheat development experience (point 1), to modify a complex game engine (point 2), and to spend time implementing it themselves (point 3). However, given the maturity, volume of code, and deep rooted paradigms of popular game development tools, this approach would be both too expensive and disruptive to be successful.

Addressing this, a better solution would be to develop such a reusable form of this anticheat with *Flutter*, a popular and fast-growing cross-platform UI framework [7]–[9] with a small but equally fast-growing game development community. The small size means a lack of existing paradigms and tools to compete with, meaning a new and effective approach has the potential to have a large impact as the community grows. However, without also reducing the ongoing cost of running the anticheat on servers (point 4) this solution would still not be effective.

[1] As backwards incompatibility stems from using incompatible protocols it's not possible to solve this in a generalisable manner. A solution would instead involve modifications to other layers in the stack. It is therefore listed here for completeness, rather than to justify an aspect of the project.

Therefore, this project will attempt to implement a modified version of the previously described core anticheat, where instead computation is *distributed between clients* rather than on the server. This reduces the role of the server to little more than a matchmaking and data relay system, meaning scaling is now significantly more affordable (point 4). It will be built using Flutter, and to evaluate its feasibility for real-world game development, this project will then use the anticheat module to build an extremely basic online multiplayer racing game.

The next section will discuss how the project will be split up into individual development units, and how each unit joins together to form the final game.

2. Project Structure

This project consists of five parts, which are explored in further detail below:

1. **Client Networking:** The distributed synchronised data structure acting at the centre of the program, processing inputs and scheduling further processing. The anticheat's "core".
2. **Server Networking:** The platform to facilitate client communication.
3. **Game Driver:** The game logic which runs on the anticheat core. This defines what the game *is*.
4. **UI and Graphics:** The rendering engine, user interface, and input handling.
5. **Practicality Evaluation:** An evaluation of this anticheat system's real-world use.

2.1. Client Networking

For this part I will develop the data structure at the centre of the program's operation – the "core" of this anticheat. By design this data structure is modular, with additional pieces of functionality as layers 'wrapped' around previous layers. It makes minimal assumptions about a game's operation and allows developers to add or remove parts based on the nature of their game.

The core receives 'events' and merges these into its current perception of the environment – events such as "player A move forward" and "player B jump". At any given time clients' perceptions may differ depending on relative latencies, but will all stabilise within a fixed time period I call the *latency cutoff*. Clients will continue to receive new, unstable events as they continue to stabilise old ones, so while the perceptions may never be identical they will always be close.

At this basic level the core offers a distributed, synchronised, ordered, timestamped, sequence of events. Because racing games are very latency sensitive this project would likely develop an additional client-side-prediction (CSP) layer, which 'fast forwards' other clients' simulations to a predicted position 'in the future'. This gives the illusion of near-zero latency.

2.2. Server Networking

This is a simple module by design. At a minimum, it should relay and broadcast events sent from each client to all other clients. It may also delete invalid events it detects.

2.3. Game Driver

To better evaluate this anticheat module's practicality for real-world game development, this project will use the module to build a simple racing game.

For a most basic implementation the game will consist of cars driving around a 2D environment. There may be no collision, realistic turning, or complex resistive forces implemented at this point, but these may be developed later on as extensions.

This module may be developed as a 'pure' function to best fit with how the anticheat core works internally, ensuring replicability and deterministic execution. This means it contains no internal state and, given two identical environment states as input, will return two identical output simulated next states. During execution the core module will be re-computing the same point in time but including newly received events, hence the requirement for determinacy.

2.4. UI and Graphics

As part of evaluating the anticheat module's practicality, a rendering engine and input handling must be included for minimum functioning of the game. This will consist of no less than a static, top-down, 2D view of the cars and track.

This feature may, like the Game Driver, be developed as a 'pure' function, taking in an environment state and producing a consistent rendered output.

2.5. Practicality Evaluation

A key part of this project is to evaluate both quantitatively and qualitatively how practical this anticheat is for game development. Key factors will be ease of implementation of the game, performance with many simulated clients, and quality of state synchronisation (e.g. accuracy, stability), all evaluated under various conditions such as varying player counts, latencies and bandwidth.

3. Starting Point

I will be building this project on top of the pre-existing SDK from Dart and Flutter.

I am familiar with Dart and Flutter from building a small variety of applications, and during the two weeks preceding Michaelmas 2023 I partially developed a 3D rendering engine with them. The rendering engine may be completed and used in the project as part of an extension.

I have only studied graphics in *Introduction to Graphics* and *Further Graphics*, and networking and asynchronous systems in *Computer Networking* and *Concurrent and Distributed Systems*.

I additionally have some experience writing simple client-server applications in NodeJS and JavaScript.

4. Success Criteria

If the following criteria are fulfilled, the project will have been successful.

1. Develop a distributed state synchronisation data structure.
2. Develop a user interface module to accept user input and render the game.
3. Develop a game driver module, which provides the logic to 'step forward' and run the game.
4. Develop a server module to relay events between clients.
5. Combine the modules into a running game.
6. Evaluate the practical success of the project using metrics such as framerate and synchronisation quality (deviation from true state, rate of jitter), under conditions such as varying player count, latencies, and bandwidth.

5. Possible Extensions

This project by its modular design and nature, has extensions which are similarly modular and self-contained, resulting in many smaller extensions rather than few large ones. Some potential extensions are listed below, and given the volume there is no expectation for them all to be completed.

- **Interpolation Extension** - *depends on Client Networking*
  This module will reduce the jittery effects of the unstable period (before events pass the latency cutoff), by wrapping the engine's output and smoothing / interpolating all values over a short period.
- **Shared Computation Extension** - *depends on Client Networking, Server Networking*
  As each client must simulate all players in the game locally, a large number of players will reduce the game's performance to the point of failure. For example, if client A 'trusts' client B, then B can stream up its computed values for A to stream down. Depending on the 'trust' graph formed by all clients (i.e. who trusts who), total computation can be reduced significantly.
- **Re-Connection Extension** - *depends on Client Networking, Server Networking*
  If a client loses its state of the game (e.g. briefly disconnects and re-connects) it would need to re-compute the entire game's worth of events to get back to the present. A more effective solution would be to have all clients regularly upload snapshots of their stable game state, which the server can then check for equivalence and store. When a client requests re-connection, the server can send the latest snapshot and all events since, allowing the client to re-start quickly and efficiently.
- **Room Extension** - *depends on Client Networking, Server Networking*
  Adds support for multiple game sessions at once. Will be useful to demonstrate the low strain this anticheat method has on the server as the number of concurrent sessions increases.
- **UDP Extension** - *depends on Client Networking, Server Networking*
  As the anticheat system can handle out-of-order events, modify the networking systems to run over UDP rather than TCP. This would require implementing packet re-sending and evaluating the change in performance and stability.
- **P2P Extension** - *depends on Client Networking*
  Given the server's limited role, modify the Client Networking module to run over a peer-to-peer network and evaluate the change in security, stability, and performance.
- **Reputation Extension** - *depends on Server Networking, Shared Computation Extension*
  In a real online environment most users will not be playing with "trusted" users, meaning they are limited in their ability to share computation. This extension would automate this process giving each player a reputation which is based on the amount of times they've truthfully shared their computation with others.
- **Turning Extension** - *depends on Game Driver*
  Make vehicle turning more realistic. Turning would then depend on a vehicle's speed and its origin of rotation.
- **Resistance Extension** - *depends on Game Driver*
  Make vehicles' resistance forces more realistic, with separate parallel and perpendicular tire and air resistances. Potentially implemented as a stateless function which given a vehicle's state returns the resistive forces.

- **Non-Static Map Extension** - *depends on UI and Graphics*
  Makes the screen follow the current player.
- **3D Extension** - *depends on UI and Graphics*
  Move to a 3D rendering engine, including vehicles and environment. Camera would follow the current player.
- **Mobile Extension** - *depends on UI and Graphics*
  Application runs on mobile either as a native app or web app. Uses touchscreen alternatives to the arrow keys for input.
- **Tilt Extension** - *depends on UI and Graphics, Mobile Extension*
  Uses phone's gyroscope for tilt-based steering.
- **Mini-Map Extension** - *depends on UI and Graphics, Non-Static Map Extension*
  Adds overlaid static view of the track and all vehicles.
- **Animation Extension** - *depends on UI and Graphics, Non-Static Map Extension, 3D Extension*
  Adds nicer 3D models. Animated wheels, particle effects, etc.

6. Work Plan

Michaelmas week 3-4 — *Oct 19 - Nov 1*
- UI and Graphics core
- Game Driver core
- Draft dissertation introduction chapter

Michaelmas week 5-6 — *Nov 2 - Nov 15*
- Client Networking core
- Finish dissertation introduction chapter

Michaelmas week 7-8 — *Nov 16 - Nov 29*
- Client Networking core cont.
- Server Networking core
- Produce dissertation outline
- **Milestone: Completed core game**
- **Milestone: Completed dissertation introduction chapter**
- **Milestone: Completed whole dissertation outline**

Vacation week 1-2 — *Nov 30 - Dec 13*
- Interpolation Extension
- Shared Computation Extension
- Draft dissertation preparation chapter

Vacation week 3 — *Dec 14 - Dec 20*
- Re-Connection Extension
- Finish dissertation preparation chapter
- **Milestone: Completed dissertation preparation chapter**

Vacation week 4 — *Dec 21 - Dec 27*
- *Christmas Break*

Vacation week 5-6 — *Dec 28 - Jan 10*
- Non-Static Map Extension
- Resistance Extension
- Turning Extension
- Draft dissertation implementation chapter

Vacation week 7 — *Jan 11 - Jan 17*
- *Break*

Lent week 1-2 — *Jan 18 - Jan 31*
- Buffer time

Lent week 3-4 — *Feb 1 - Feb 14*
- Buffer time
- **Milestone: Completed project**

Lent week 5-6 — *Feb 15 - Feb 28*
- Perform Practicality Evaluation
- Finish implementation chapter
- **Milestone: Completed dissertation implementation chapter**

Lent week 7-8 — *Feb 29 - March 13*
- Write dissertation evaluation chapter
- Write dissertation conclusions chapter
- **Milestone: Completed draft dissertation**

Submission — *May 10*

7. Resource Declaration
- My laptop for all work. It is a 16″ 2021 M1 MacBook Pro, with 32GB memory and 1TB SSD. This will be the device on which the project will be run and tested on. If this machine fails, I will promptly buy another machine and continue development. *I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.*
- If the Mobile Extension is undertaken, I will be using my iPhone 13 Pro for running and testing.
- The Dart and Flutter SDK. Used to execute and compile code for various platforms, including MacOS, iOS, and web.
- The following Dart/Flutter libraries:
  - "`dart:io`" for server-side WebSockets
  - "`web_socket_channel`" for cross-platform client-side WebSockets
- A local network if testing across real devices is to happen.
- Git and GitHub as VCS for my source code and dissertation, and I will be making weekly backups of all relevant files to Google Drive and an external SSD.

8. Bibliography

[1] "Chart: Are You Not Entertained? | Statista." Accessed: Oct. 13, 2023. [Online]. Available: https://www.statista.com/chart/22392/global-revenue-of-selected-entertainment-industry-sectors/

[2] "Riot Games Vulnerability Disclosure Policy." Accessed: Oct. 07, 2023. [Online]. Available: https://hackerone.com/riot?type=team

[3] "Protocol." Accessed: Oct. 07, 2023. [Online]. Available: https://wiki.vg/Protocol#Set_Player_Position

[4] "Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization." Accessed: Oct. 07, 2023. [Online]. Available: https://developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_In-game_Protocol_Design_and_Optimization

[5] "Anti-Cheat system for Unity3d." Accessed: Oct. 14, 2023. [Online]. Available: https://discussions.unity.com/t/anti-cheat-system-for-unity3d/35730

[6] "Using Anti-Cheat | EOS Online Subsystem." Accessed: Oct. 14, 2023. [Online]. Available: https://docs.redpoint.games/eos-online-subsystem/docs/networking_anticheat/

[7] "Flutter - Build apps for any screen." Accessed: Oct. 13, 2023. [Online]. Available: https://flutter.dev/

[8] "Stack Overflow Developer Survey 2023." Accessed: Oct. 07, 2023. [Online]. Available: https://survey.stackoverflow.co/2023/

[9] J. Lee, "Flutter 2023 Q1 survey — API breaking changes, deep linking, and more." Accessed: Oct. 07, 2023. [Online]. Available: https://medium.com/flutter/7ff692f974e0