Computation Offloading for Sophisticated Mobile-Games

M. Jiang

Utrecht University

Computation Offloading for Sophisticated Mobile-Games

by

M. Jiang

in partial fulfillment of the requirements for the degree of

Master of Science
in Game and Media Technology

at the Utrecht University,

Supervisors:	Dr. ir. S.W.B. Wishnu,	Utrecht University
	Dr. ir. A. Iosup,	Delft University of Technology
	Ir. O.W. Visser,	Delft University of Technology
Thesis committee:	Dr. A. Egges,	Utrecht University

Contents

1 Introduction **1**

2 Related Work **3**

3 The Offloading Framework **7**
3.1 Architecture . . . 7
3.2 Tick Synchronization . . . 8
3.3 Program Partitioning . . . 11
3.4 Offloading Control . . . 13
3.5 Offloading Decision . . . 14
3.6 Limitations . . . 14

4 Experimental Setup **17**
4.1 OpenTTD . . . 17
4.2 Offloading OpenTTD . . . 19
4.3 Setup . . . 20
4.3.1 Game Settings and Save Games . . . 20
4.3.2 Offloading Parameters . . . 21
4.3.3 Devices . . . 22
4.3.4 Offloading Strategies . . . 23
4.3.5 Data Recording . . . 24
4.3.6 Experiment Sets . . . 24

5 Results **27**
5.1 Performance . . . 27
5.2 Game Smoothness . . . 29
5.3 Game Responsiveness . . . 31
5.4 Bandwidth Usage . . . 32
5.5 Offloading Strategies . . . 34
5.6 Power Consumption . . . 36
5.7 Game Consistency . . . 37

6 Conclusions and Future Work **39**
6.1 Conclusions . . . 39
6.2 Future Work . . . 40

Bibliography **43**

1

Introduction

Mobile gaming [1] is currently a market with incredible growth with new games continually being released for various platforms including Windows, iOS and Android [2]. However, mobile platforms like smart phones and tablets have limited computational power which only allows them to play fairly simple games. Furthermore, mobile platforms have limited battery life which can be depleted very quickly by a combination of, for example, heavy computations, network usage and camera usage [3]. These factors severely limit the complexity and scale of mobile games that can be reasonably run on mobile platforms.

A potential solution to this problem is by offloading the resource usage by a game to a remove server. The basic idea of this is to let remote machines do the resource intensive work and sending the results to the mobile client. This can free up local resources on the mobile client as well as possibly save some of its valuable battery power [4].

There are several resources that could be the potential target of offloading. These include computation time, memory, storage and network usage. However, offloading anything requires an Internet connection, which results in three major problems. The first one is that mobile clients may have unreliable connections with small bandwidths, limiting the amount of tasks that can be offloaded. Secondly, the client has to wait for the results of offloaded tasks, which can introduce unacceptably long delays in a real-time game. Third, additional network usage may decrease the battery life of the client if insufficient hardware is freed up in return.

Network offloading on the other hand can decrease the overall network usage of a game, but requires a certain amount of network overhead itself. Network offloading is used to lighten the burden on the client of collecting and organizing of data that is located somewhere else on the Internet.

Because of these challenges with offloading, which tasks, which resources and how to offload should be carefully chosen to make offloading not only beneficial performance wise, but also to minimize its negative impacts on the gameplay experience like stutter and delay. These choices depend on several parameters including the amount of local resources available, network bandwidth, network delays and the battery level of the client. The most beneficial functions to offload are those that are resource intensive, have input and output of small sizes and can tolerate a long response delay [4]. In this thesis, we will call these types of functions *coarse-grained functions*. *Fine-grained functions* have a worse computation versus input/output size ratio and also are called very often.

The benefits of computation offloading have been proven for games with coarse-grained functions [5, 6]. These mobile games have a single coarse-grained function that is offloaded to a remote machine, like the artificial intelligence of chess and the image recognition of augmented reality games. These games are generally simple in their gameplay features. Many other types of video games however have more complicated and feature rich gameplay and therefore, have their major computational loads spread out across many different fine grained functions. It remains unclear whether offloading can benefit these kinds of games as well.

For computation offloading to be viable for sophisticated mobile games, it has to meet certain minimum requirements. The solution should be general enough so that is can be applied to all, if not many, types of games. The performance of offloaded games should increase even at the absence of any coarse-grained functions. From the players' perspective, the game experience should remain

smooth and responsive as possible. Finally, the solution should not have too much of a negative impact on the overall power consumption on the mobile device.

The goal of our research is to investigate the viability of performing computation offloading for sophisticated games with fine grained functions. Our contributions are as follows. First of all, we present a new generalized offloading framework (OF) which automates part of the offloading process and that simplies the offloading implementation process for mobile game developers. Secondly, we present the *offloadable entity interface*, that allows game developers to partition their program on the class level while allowing dynamic online control over them by the framework on the instance level. Thirdly, we show how our framework can dynamically steer the offloading process to meet certain user level goals by using different *offloading strategies*. Lastly, we show the results from our experiments which indicate that using our framework for fine grained offloading for mobile games significantly increases performance and gives a fairly smooth gaming experience, but also increases power consumption and induces a slight delay on user input.

The rest of this thesis is structured as follows. In Chapter 2 we look into the current state of the art in offloading. In Chapter 3 we will present the offloading framework we have created and explain it in detail. We show our experimental setup in Chapter 4 and show and discuss our results in Chapter 5. Finally in Chapter 6 we summarize our findings and present potential directions for future studies.

2

Related Work

The vision that one day computing and communication technologies would be seamlessly integrated into our lives has existed since 1991 [7, 8]. This vision, called *ubiquitous computing* or *pervasive computing*, has been brought much closer to reality by the wide adoption of mobile devices like smartphones and tablets in the modern society.

However, all of the fundamental constraints of mobile devices still apply and need to be alleviated to further progress towards true pervasive computing [9]. These fundamental constraints are that mobile-devices are resource-poor compared to static devices, that mobile connections are unreliable and that mobile devices rely on finite energy sources.

The idea of using offloading to alleviate some of these fundamental constraints was first proposed by Balan and Flinn 2002 [10] as *cyber foraging*. They described the idea of cyber foraging as "living off the land". The client would use local wireless network technologies to discover resourceful machines called *surrogates*. Two characteristics of these surrogates are that they are not trusted by the clients and are also not directly managed by human users during the offloading process. Balan and Flinn 2002 [10] implemented cyber foraging in two ways. The first one is called data staging, where the surrogate acts as an Internet cache for the client, predicting and fetching information from distant servers for the client. The second one is offloading computation by generating code and API at run-time on the client and sending these to the surrogate for execution.

Their work inspired numerous other projects creating variations of the implementation of cyber foraging [11–14]. They differ from each other in various areas like how and which part of applications are profiled and partitioned for local and remote execution, how offloading is decided and controlled and how surrogates are discovered and how they are managed. Despite these variations, the general idea of computation offloading remained the same, which is to somehow relief some work from the mobile client by using the resources of other machines and using a network connection to transfer the information.

Olteanu and Ţăpuş 2014 [15] and Khan 2015 [16] have performed a survey on the subject of offloading. These surveys show that offloading has been a fairly popular topic the past decade, where different works have created frameworks that all vary from each other one way or another. The resource that they offload are almost always CPU load, but there are also some works like Hassan and Chen 2011 [17] and Flores and Srirama 2013[18] can also outsource the processing of local data, Huerta-Canepa and Dongman 2010 [19] mainly focus on offloading memory usage and the frameowrk of Zhang and Jeong 2010 [20] can also offload disk storage. The effects of these different frameworks for the client-side generally follows the same trend of increasing performance and saving power, proving that offloading can be used with success in various different ways.

Olteanu and Ţăpuş 2014 [15] have identified the main concerns of offloading and created a taxonomy for them to better compare the different frameworks. Figure 2.1 is taken from their work and shows the taxonomy used by them. The *application monitoring* area involves how the framework profiles and partitions the application. *Resource management* concerns the monitoring of local resources on the mobile client as well as the discovery of potential machines to offload to. The *offload process* controls when and what to offload, possibly using the information from the application monitoring

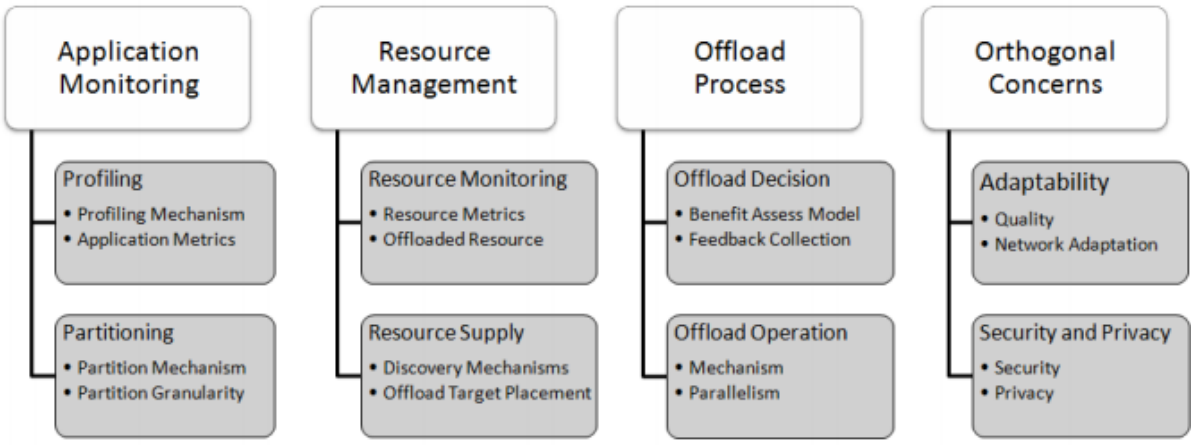


Figure 2.1: The taxonomy for offloading concerns as identified by [15]. Image taken from Olteanu and Ţăpuş 2014 [15].

and resource management areas, to get the best benefits. Finally the *ortogonal concerns* involve for example how flexible the frameworks are and how secure the system is.

Olteanu and Ţăpuş 2014 [15] have categorized each of the frameworks they have analyzed using their taxonomy. Most created frameworks differ from each other in many of these aspects. Our framework is no exception. The number of variations for every aspect of offloading for all previously created frameworks is too large for us to handle here. We therefore refer you to Olteanu and Ţăpuş 2014 [15] for an overview of what previous frameworks have done.

We will however describe our framework using this same taxonomy of, so it can be more easily compared to others. As our framework was initially created to create a way to perform real-time offloading of sophisticated mobile games and see whether it is beneficial, it lacks many features that belong to more complete frameworks. Below is the description of our framework using the defined taxonomy. For a more detailed explanation of how our framework works, see Chapter 3.

1. Application monitoring
   (a) Profiling mechanism: None. Our framework does not provide any tools to profile the application, but it can be extended with such in the future.
   (b) Application metrics: None, as our framework does not perform profiling.
   (c) Partition mechanism: Manual partition by the developers by implementing our interfaces.
   (d) Partition granularity: Code partioning is on the class level, but during run-time the partioning is done on the instance level of the partioned classes. This allows dynamic fine-grained offloading.
2. Resource management
   (a) Resource metrics: Our framework currently monitors the ticks per second and the bandwidth usage of the application to guide the offloading process, but can extended to use other metrics too.
   (b) Offloaded resource: Computation.
   (c) Discovery mechanism: Currently, the user has to manually input the server's address, but our framework easily supports extension where a more automated discovery and management of servers is supported.
   (d) Offload target placement: The server can be run on basically any machine that the client can connect to and that to be offloaded application can be run on, from home computers to powerful cloud servers.
3. Offload process
   (a) Benefit Assesment: None, but can be extended with a learning process.

(b) Feedback Collection: None.

(c) Offloading mechanism: Our framework uses a form of replication where the same game runs both on the client and the server. The server runs ahead of the client and sends the offloaded state changes to the client when needed.

(d) Paralellism: The client and the server run the game concurrently.

4. Ortogonal concerns

(a) Adaptability: None.

(b) Security and privacy: None.

There has not been many offloading frameworks that were tested using video games and even less were created specifically for them. The frameworks that do have are are Li et al. 2001 [21], Chu et al. 2004 [22], Kemp et al. 2010 [5] and Cuervo et al. 2010 [23]. However, these works only tested their framework using very simple games like board games. They also use a stateless server. The client is then required to serialize its game state at every call of an offloaded function and send it to the server along with the function to call. The server will then perform the function and send the game state back. During all this time, the client is required to block the program. This generally works well for simple, non-real-time games like the board games which are experimented with, but unacceptable in modern games. This is because modern games could have an FPS of 60, which means a single frame must complete within a 16ms. Sending an offloading request and receiving its result of a single function takes too long from this perspective; not only would it require a ping to the server lower than 16ms, it would also require the server to complete the function really fast, the client to serialize its state accurately and fast and also for the result to arrive soon enough so the client still has time to perform the rest of its frame like rendering. And this is only for *one* call to an offloadable function. Clearly, such an approach is not suitable for sophisticated games we want to offload.

The work most similar to our own one is Kemp et al. 2010 [5], which created the *Cuckoo framework*, a system that can perform computation offload for Android applications. The authors have created a programming model that extends the build process of Android applications. It automatically creates interfaces for Android services that the developer has to implement. At run-time, the Cuckoo framework can decide whether a call to a service should be done locally or remotely. Like many other offloading frameworks, the Cuckoo framework also uses a stateless server and offloads functions on a call by call basis. Another limitation of their work is that their framework enforces the use of the Android build structure and also require that the server side uses the Java virtual machine.

Olteanu and Ţăpuş 2014 [15] have also performed experiments with offloading of OpenTTD, which is exactly the same game we that have used. However, it is unclear which offloading framework they have used to obtain their results and also unclear what their exact experimental setup was.

Our work focuses on finding out what the effects are of offloading on sophisticated mobile games. For this we have created a framework that tries meet the high real-time and user experience requirements of sophisticated mobile games. This focus has resulted that although our offloading framework uses some existing concepts, but also being vastly different in other aspects. There are two key aspects that our framework differs from others. The first one is that we do not use a stateless server, but use replication instead. The second one is that the client only need to subscribe to certain in-game events that it wants to offload. The client then does not need to run the logic of those events itself but only needs to wait for the results that the server sends. These two aspects together makes it possible that the client can continue the gameloop without the need to offload and wait for the result of every call to an offladable function. Another benefit is that it significantly lowers the overhead required for offloading because the client does not need to serialize game states.

As far as we are aware of, our work is also the first that thoroughly examines the different effects of offloading on a game. We do not only examine the performance, power consumption and bandwidth usage, but we also examine the delays and stutterings caused by offloading.

An existing alternative to offloading for games is *cloud gaming* [24]. Cloud gaming attempts to allow users to play the most modern games without the need to constantly upgrade their hardware. Cloud gaming systems execute all of the game's logic on a remote cloud server, renders the scene and sends it as a video to the client. The client itself is very thin and only acts as a user input terminal and video player. Cloud gaming can be considered the most extreme form of offloading, where only

the bare minimum is done on the client. Out of the many cloud gaming systems that existed over the years, only a few remain like *StreamMyGame* and *GamingAnywhere* [24].

The exact reason for why commercial cloud gaming systems have not been very successful is outside the scope of this paper, but from a computer science point of view, cloud gaming systems suffer a few major problems that are inherent to the Internet [25]. The first one is that no matter how fast and stable the user's connection is, they will always experience a delay between their moment of input and the moment the result of the input is fed back to the user. This delay will be at least as long as a single round-trip time between the client and the server. The second problem is that the Internet is inherently unstable and that packets can and will get lost during transit. Cloud gaming systems are very sensitive to both of these types of problems as they cause artifacts and stutter in the streamed video [26]. This reduces playability and can also break the immersion of the player. Furthermore, users of mobile platforms do not always have access to a high quality Internet connection, making cloud gaming rather unsuitable for mobile gaming.

Our work is somewhat similar to cloud gaming systems in the sense that the client and server need to be synchronized through a stream of messages. Any delays in these messages can cause stutter on the client side. Our framework also creates a delay beteen the moment of user input and the moment the user can see the effects of the input. This is also a neccessary evil caused by our architecture, which is explained in detail in Chapter 3. We have analyzed the extend of both of these problems in Chapter 5.

3

The Offloading Framework

In this chapter we will describe the offloading framework (OF) we have created as a tool to help game developers perform computation offloading for their mobile games.

The offloading framework is designed with mobile games in mind, but the idea is general enough for games on other platforms too. Our framework does have a requirement on the delta-time of the game. The delta-time of the game defines how far the simulation of the game world should progress with each update and can vary from update to update. Our framework requires that the delta-time of the game to be fixed throughout a session. A common genre of games that use such a scheme are real-time strategy games (RTS), where the whole game is divided into equal sized simulation ticks.

We will first present the general architecture of the offloading framework in Section 3.1. In the sections that follow we will go deeper into the specifics of the components of the offloading framework. In Section 3.2 we will show a variation of the lockstep algorithm being used to timestamp in-game events. This is needed for event synchronization. In Section 3.3 we will explain a core concept of our framework, the *offloadable entity*, that allows program partitioning as well as dynamic offloading. In Section 3.4 we explain which parts of the offloading process the offloading controller will automate and what is required from the game for that. Section 3.5 explains how we decide which entities to offload and presents our concept of *offloading strategies* that guides the offloading process to meet certain user preferences. Finally, in Section 3.6 we go through some of the limitations of our framework and aspects that could be improved.

3.1. Architecture

Figure 3.1 shows the general architecture and the main components of the offloading framework. At the highest level it consists of a client and a server running the same game that uses the offloading framework in respectively client and server mode. The server will be the machine that the client will offload computation to. In principle this can be any machine that the game can run on and that the client can connect to using an Internet connection. These machines can range from home computers to virtual machines in the cloud. The framework requires that the game is pre-installed on the server and that the IP-address and port number of the server are known to the client beforehand.

The framework has been made to be as indepedent from the actual game as possible, so it can be compatible with more than one game. This has resulted in two separate interfaces that are used to communicate with the underlying game, which are the *offloadable entity* and the *offloading game controller* interfaces. These two interfaces will be explained in detail in Section 3.3 and Section 3.4 respectively. Furthermore, all offloading related network is controlled by the framework itself.

Both the client and the server will run the same game at the same time. We use this replication technique for several reasons. The reason that the whole game is on the client-side is because in a situation where no network connection is available, the client should still be able to run the whole game, although possibly with lower performance. We think this is important as mobile users may often temporarily lose network connection as they move. However, in our current implementation, it is not possible yet to continue playing and reconnecting later, but the framework is adjustable to allow this in the future. The reason that the server-side runs the whole game too is to avoid the need to synchronize

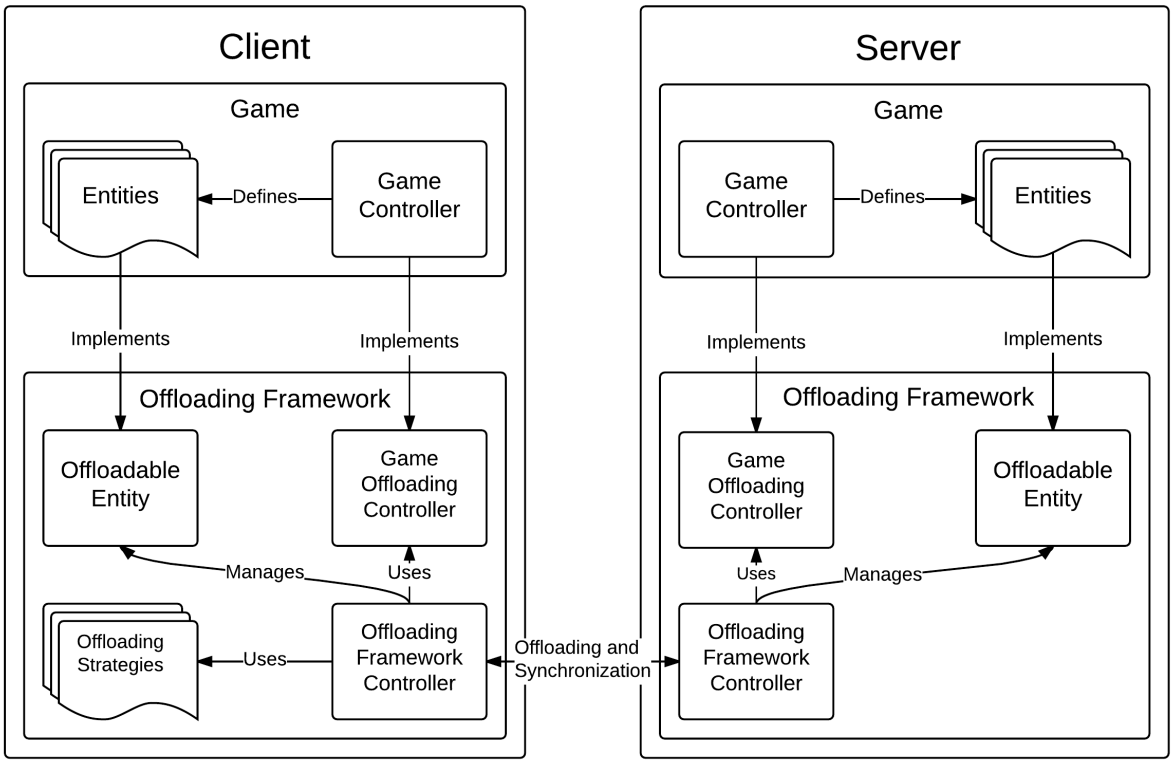


Figure 3.1: Architecture of the offloading framework and its components. The game depends on the offloading framework, but not vice-versa. The client and server both run the same game. Offloading and synchronization are completely managed by the framework at run-time. The server-side does not employ offloading strategies as the client decides which entities to offload.

the complete game state. This is very important as this dramatically reduces network usage as well as simplifying the framework.

An offloading session can be started by the client's request. The client will first connect to the server and then communicate a game starting point using a savegame that is either pre-loaded on the server or sent on the fly. Both sides then run the game like normal, but with the framework in control of offloading. In our current implementation, it is required to manually copy the save from the client to the server beforehand. Both saves also must have the same name and the save must be stored into the default save folder of OpenTTD on the server-side.

During an offloading session, the client will subscribe to certain types of events in the game. The client will then stop handling these types of events itself and instead, waits for the server to handle such an event when it arises and sending the client a message with the result of it. We will call such a message an *offloaded event message*. This method removes the need to send each individual offloading request and in doing so, removes the need to block the program at each call of an offloaded function. This latter is essential to get a smooth game experience.

The benefits of using this architecture are as follows. Firstly, the client does not need to serialize and send the game state for every offloading request, which dramatically reduces the overhead required to perform offloading. Secondly, the client does not need to send an offloading request at every call of an offloadable function. The client simply subscribes to an event and waits for offloaded event messages. This allows developers to beneficially offload even the most fine-grained functions, as the client does not need to make the decision of whether it is faster to just execute the function itself instead of sending an offloading request and waiting for the result. The client will never need to wait for an offloaded event message as the server will send the result to the client ahead of time.

3.2. Tick Synchronization

Many events in video games have a temporal aspect to them; their logic must be performed at some specific time for them to have the desired effect. When running a game normally without offloading,

an event can simply be performed the moment it arises. When performing computation offloading however, it becomes necessary to time-stamp events as they need to be communicated with the other party so they are executed at the exact same moment in the simulation for all parties.

The framework that we have created uses a variation of the *lockstep* system to achieve this. Bettner and Terrano 2001 [27] describe how they implemented lockstep for *Age of Empires*. The lockstep system in video games basically consists of multiple machines running the same game in parallel, notifying each other of their progress using some time unit T and synchronizing user input so they happen at the same time on all machines. At some point during the session, one of the machines may notice that the difference between its own T and the T with one or more of its peers is larger than a pre-defined amount. In such a case the machine must wait till its peers catches up with it.

Our framework only supports games that simulates its game state in discrete *ticks*, which is also typically what lockstep games use. A tick is a single game simulation step, where the delta-time of each tick is the same no matter how much time has passed between two ticks. The game may set this delta-time as any value as long as all ticks have this value. The T of the application can then simply be denoted as the tick at which it is at. The framework will then time-stamp all events using the tick at which they occurred. This allows the framework to accurately communicate events in such a way that they happen at the exact same moment in the game of both the client and the server.

However, some genres of games typically use a variable delta-time for each tick, like first-person shooter games. Having this requirements makes our framework incompatible with them. We still decided to have this requirement as it makes synchronizing events much easier. To illustrate this, imagine that the server wants to synchronize an event at $T = 1$. If the game has a variable delta-time, the may be possible that the client never arrives at $T = 1$ and instead, may jump from $T = 0.9$ to $T = 1.1$ after a single tick. Although these types of games can often tolerate these margins of error, we think that for offloading, these errors can become too big causing desynchronizations. If the delta-time of each tick is equal, every instance of the same application will start at the same T at some point in time.

During an offloading session, both the client and the server notify each other of their respective ticks at certain intervals as they progress. This *tick update frequency* (TUF) can be adjusted to compromise between game smoothness and bandwidth usage. High frequency tick updates use more bandwidth, while low frequency tick updates causes both parties to have more inaccurate information about the state of the other party. Accurate information on the tick of the other party is very important in our framework, as it can influence the smoothness of the game on the client-side as well as the response time of user-input. Both of these aspects have to do with how our lockstep system works, which is explained below.

We define *transmission delay* (TD) as the single-trip time measured in ticks for a message to arrive from the client to the server or vice-versa. This value is measured dynamically during run-time and can fluctuate depending on network conditions. The framework measures this value by sending, at the application level, a TD measurement packet to the server. Upon receiving this packet, the server will immediately respond with a TD measurement response packet. When the client receives this response, it will then calculate the difference in ticks between the moment of sending the the TD measurement packet and the moment it received the response packet. Dividing this value by two and ceiling the result yields the TD value. A thing to note about the TD value is that it has a minimum value of one. This is because due to the update order of the game, it is impossible for the client to send and receive the result of a TD measurement packet within a single tick. A TD smaller than one is also nonsensical, as a single tick is inidivisible, so events cannot be scheduled at different times within a single tick.

Our decision to measure TD in ticks instead of milliseconds like ping does introduce some subtle inaccuracies. The TD is measured at the client-side and on the same thread as the game loop. This means that when the game slows down, the relation between TD and the actual ping between the client and the server changes. It might have indeed be a better idea to measure TD in a separate thread outside of the gameloop in milliseconds instead of ticks and then converting this value to ticks to schedule events. This way, we avoid inaccuracies caused by the varying speeds of the game loop.

To ensure that game event messages from the server arrive at the client in time before their scheduled ticks, the client should always stay at least TD number of ticks behind the server. The client machine itself should enforce this rule by waiting when it detects that the difference between its own tick and the last known server tick comes too close to TD. This way, the client will always remain at least TD ticks behind the server even when the client does not have the actual tick of the server. The

actual tick of the server can only be equal or higher than the server tick known by the client, thus the actual tick difference will never be lower than the one calculated by the client.

However, the above reasoning is only true for a perfect Internet connection where the measured TD remains the same for all messages. This is obviously unrealistic, so one should let the client stay behind the server for more ticks than TD to cope with network uncertainties. We define the *lower tick difference* (LTD) as the advised number of ticks the client should stay behind the server to maximize the likelihood that messages arrive in time even under flunctuating connection qualities. During an offloading session, the client will check whether it is still indeed behind the currently known server tick by at least LTD number of ticks. If not, it will stop simulating the game till it receives a server tick update message that would cause the tick difference to be larger than LTD ticks again. A downside of this scheme is that the player of the game may experience hiccups in the game as the client waits for the server.

With this rule, we have created a way for offloading event messages from the server to likely arrive at the client in time for execution. One problem remains to be solved however, which is how to communicate user input events from the client to the server. The client cannot execute the user event immediately, as otherwise it would cause an inconsistent state for the server as the server has already passed the tick of that event. The solution we use to solve this problem is to schedule the user event in both the future of the client and the server. In particular, to ensure that user input is process synchronously, we schedule the event at the current known server tick at the client plus at least *input tick delay* (ITD) ticks.

Choosing the ITD value is much harder than choosing the LTD value. This is because we measure TD from the client's perspective. There is no guarantee that the server will not simulate faster than the client, in fact it will likely do so as it is a more powerful machine. This results in the time between two ticks on the server-side having a different value than on the client-side. As our TD measurement scheme is dependent on this time between two ticks, it makes the measurement more uncertain.

Moreover, if the server simulates the game much faster than the client, the tick the server is at will be increasingly further away for the client as time passes. This means that when the client wants to schedule a user input event, it has to schedule it very far into the future, causing huge user input delays, which is unacceptable for playability.

To remedy both of the above problems with scheduling user input, we define the *upper tick difference* (UTD) to be the maximum number of ticks the server is allowed to be in front of the client. Unlike LTD, it will be the server this time which will check and enforce this rule. This rule ensures that the server will not simulate too far ahead of the client, which minimizes user input delays perceived by the player of the game. A second result of this rule is that in the case that the server can indeed simulate much faster than the client, it is likely that the client's tick will often stay at UTD ticks behind the server. This causes the server to often stop simulating and wait for the client to keep the client's tick within UTD ticks behind it. The result of this is that the server will then more or less simulate as fast as the client does, which makes the TD measurement for user input delays more accurate.

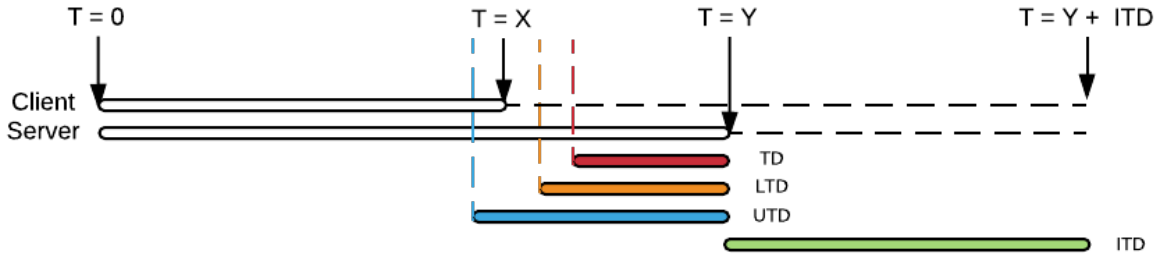
With this additional rule, we can then make a rough estimate of the minimum value of ITD. We estimate this value to be at least $2 \cdot TD$. We need to at least schedule the event $1 \cdot TD$ ahead of the server due to message sending delays. An additional $1 \cdot TD$ is required as the client may not have the most up to date server tick. For example, if the $TUF = 1$ for the server-side, there may be TD number of tick update messages underway from the server to the client.

The LTD, UTD and ITD values are dynamically calculated at run-time according to the measured TD of that time. This means that they change along with TD. To obtain the value of LTD, UTD and ITD at run-time, we multiply TD with some real number and then ceil it. The real number TD is multiplied with can be chosen differently for LTD, UTD and ITD as long as they satisfy the conditions given by Equation 3.1 and Equation 3.2. The value of ITD can be chosen freely from the other two as it is used for something completely different. For example, one can set LTD as $1 \cdot TD$, UTD as $1.5 \cdot TD$ and ITD as $2 \cdot TD$.

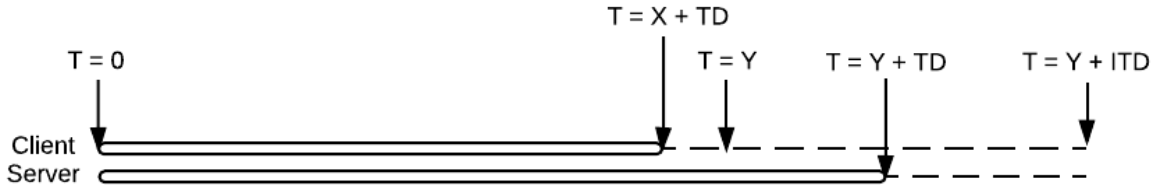
$$TD \leq LTD \leq UTD \tag{3.1}$$

$$2 \cdot TD \leq ITD \tag{3.2}$$

Figure 3.2 is an exmaple of a situation with all the above rules combined. Ticks are represented as lengths in the figure. To summarize the above rules and explain what happens during an offloading



(a) In this situation, the client is currently at tick X and the server at tick Y. The values for TD, LTD, UTD and ITD are shown. Note that the client tick is maintained between LTD and UTD. Suppose that two things happen at this moment. Firstly, the client receives user input, it will schedule the event at tick $Y + ITD$ and sends the message to the server. Secondly, the server detects an offloaded event, schedules it at tick Y and sends the message to the client.



(b) This is the situation after TD ticks have passed since the last situation. The client is now at X + TD ticks and the server at Y + TD ticks. Both parties likely have received the message from the other party, as the chosen LTD, UTD and ITD values are all equal or larger than the measured TD.

Figure 3.2: An example scenario to visualize our lockstep scheme. a) Initial situation. b) Situation after TD ticks.

session, we periodically measure the TD value on the client-side to estimate the time in ticks required for a message from the client to arrive at the server and vice-versa. The user of the framework will choose the LTD, UTD and ITD values as a multiple of the measured TD according to Equation 3.1 and Equation 3.2. The client's tick will be kept between LTD and UTD number of ticks behind the server. Offloaded events will and must be scheduled at the tick at which the server detects them. Because the client will be at least TD ticks behind the server, offloaded event messages from the server to the client will likely arrive in time before the client reaches the tick at which the event occurred. User inputs will be scheduled ITD number of ticks ahead of the server tick. This means that the client will need to wait at least $LTD + ITD$ number of ticks before it can process the user input event. This causes a user input delay from the player's perspective, which can be a problem for the gameplay experience.

3.3. Program Partitioning

For computation offloading, one needs to create well defined parts of the program that will act as candidates for offloading. This can be done in various ways. An extra requirement for our work is that it must be possible to dynamically turn offloading on and off for the offloadable parts of the program. In this section, we will present the method we have chosen to partition games.

We have decided to do the partioning of the game during development, instead of during run-time. This is because we believe that run-time partioning is too computationally intensive by itself to be of any use in a game.

To partition a game and to be able to dynamically select offloadable parts, we have decided to divide a game into *offloadable entities* (OEs), where each OE is some instance of a class that implements the OE interface. An OE can be for example, a vehicle that runs its own logic, or an abstract controller object like an artificial intelligence that manages the vehicles.

The motivation behind this partioning scheme is that a typical video game is always separated into well defined classes of which multiple instances may be created of at run-time. Each separate instance runs its own logic and contributes to the overall computational load of the game. Intuitively, it seems like a good idea to implement an interface for each type of object one wants to offload while deciding which instance to offload at run-time.

```
Vehicle : OffloadableEntity {
   void Update() {
      Move();

      if (IsServer) {
         DoPathPlanning();

         if (Offloaded)
            SendEventMessage(newDirection)
      }
      else if (IsClient & !Offloaded) {
         DoPathPlanning();
      }
   }

   void SendEventMessage(payLoad) {
      OffloadingFramework::SendEventMessage(payLoad);
   }

   void ReceiveEventMessage(payLoad) {
      direction = payLoad;
   }
}
```

Listing 3.1: Example code of a simplified class implementing the OE interface. The function Move() is always executed while DoPathPlanning() is only executed on the client if the entity is currently not being offloaded. If an instance of the class is marked as offloaded and a path-planning event is triggered on the server side, a messages will be sent by the server by creating a custom payload and sending it by calling SendEventMessage(payLoad). When the client receives the message, it will pass its contents to ReceiveEventMessage(payLoad).

Listing 3.1 shows an example implementation of the OE interface. To mark a game object as an OE, the game developer must implement the OE interface for that type of object. The OE interface requires the developers to implement several things for the class that implements it. The first one is to separate the logic of the entity into two parts; one part that will always be executed by both the client and the server and one part that will only be executed on the client if the OE is currently not being offloaded. The game itself is still fully in control of the logic of the OE. The OE does provide attributes to tell the game whether the entity is currently in an offloaded state, so the instance can know which parts of its logic can be skipped.

The second part to implement are functions that create and read event messages for that type of entity. The creation functions need to be called at the server side when the entity is marked as offloaded and a corresponding event occurs with the entity. The creation function should then generate a custom payload and pass it to the OF to handle further. The reading function is for when the OF on the client side receives an event message for this particular entity and passes the custom payload to the reading function to be processed.

Implementing the interface allows the offloading framework to know which entities can be offloaded and also allows dynamic control over them. The necessary skills required for implementing the OE interface is similar to synchronizing objects in a multiplayer online game. Developers familiar with network games should have no problem in doing this.

That said, our approach in program partioning still requires significant work from the developers. Reducing the amount of work by integrating more into the framework seems to be impossible as the partioning and the logic behind them are game specific. We have considered taking over the game loop of each OE by the framework, but abandoned the idea as it would likely cause too many problems depending on the roles of the classes that implemented the OE interface.

To make sure the client and the server can communicate information about a certain OE, each OE must have a unique identifier, represented by a number. This ID is very important as it will be used to properly communicate all information about a certain OE. Although it is possible for the offloading framework to handle the assigning of the IDs, in our current implementation we have decided that the game using the framework should assign the IDs instead. This is because we think it might be handy if the game has more control over it, as it might already use an ID system itself. Moreover, the game needs to be aware of these IDs anyway, as it is required to be able to save and load these IDs when creating or loading a save game. This is to make sure save games can be used as a synchronized initial state for offloading. The saving and loading of the game is impossible to automate by the offloading framework, as the creation of the save games is unique to the game itself.

3.4. Offloading Control

The *offloading framework controller* in Figure 3.1 manages the overall offloading process. It has numerous tasks including managing incoming and outgoing messages, maintaining tick synchronization, scheduling offloaded events and on the client side, deciding which entities to offload and communicating this with the server.

For some of these tasks it is necessary for the framework to be able to interact with the game that uses it. A part of this is done using the offloadable entity interface explained in Section 3.3. Another part is through the offloading game controller shown in Figure 3.1. The game using the offloading framework should provide it with an object that implements the offloading game controller interface. This interface defines functions that the OF can call to control the flow of the game, including pausing, resuming, loading and quitting the game. It also defines an event that the OF can subscribe to that should be invoked each time the game progressed a tick. These components together give the OF enough control over the game to automate the rest of the offloading process.

Offloading and unoffloading registered entities is automatically controlled by the framework itself. The decision process of which entities and how many entities to offload is done at the client side and will be discussed in Section 3.5. As entities can be offloaded and unoffloaded dynamically at run-time, these events need to be synchronized too. This is done by the client first sending a message to the server with the ID of the entity it wants to change the offloading state of as well as the offloading state to change to. After which the client will mark the entity as pending and will still do the same thing as before, which means it should still run all of the entity's logic as if it was marked as not offloaded. When the server receives the offloading state change message from the client, it will change the state

of the entity it corresponds to. Note that the server will run all the logic of the entity whether the client wants to offload that entity or not. The only difference is that the server will start sending the client offloaded event messages if the client wants to offload the entity. After the state change, the server will send a response message to the client with the same information the original message had as well as the server tick number the server processed the message. This tick number will be used by the client when it receives the server response to change the state of the entity at the same tick as the server.

This way, it is impossible for the client to mark the entity as offloaded before the server has, which would have resulted in desynchronization due to logic not being run as well as not having received offloaded event messages for a certain entity. However, it is possible that the client marks the entity as not offloaded later than the server, which would cause desynchronizations in the same way as above. This can happen when the client wants to change the state of an entity from offloaded to not offloaded, but receiving the response from the server later than the tick the server received the request. Fortunately, we think that the chance of this happening is fairly low due to our tick synchronization scheme. The tick the server receives the request of the client is in the future of the client, which gives the response from the server some arrive at the client in time.

An alternative to this scheme is to use the same scheme as we do to schedule user input events explained in Section 3.2. With that scheme, it is then impossible for the client to mark an entity as not offloaded later than the server, but instead, it is possible that the client marks an entity as offloaded earlier than the server. Although we think that a single packet arriving too late in this alternative scheme is equally unlikely as in our chosen scheme, we find that the characteristics of the alternative scheme are less desirable for changing the offloading state of entities. This is because marking an entity of offloading will be likely used much more often than marking an entity from offloaded to not offloaded. So even when a single packet arriving too late is equally unlikely, the difference in the number of packets does influence the chance of something going wrong between the two schemes. This is the reason why we have chosen not to use the same scheme as we do for scheduling user input events.

3.5. Offloading Decision

During run-time, the OF executes code to decide which of the existing OEs should be offloaded or unoffloaded. The frequency of these offloading decision moments can be adjusted according to the preferences of the developer or user.

To decide which and how many entities to offload, we have come up with several different policies we call *offloading strategies*. These strategies make their decisions based on the current state of the game and the goal of the user. Always offloading all entities for maximum performance may not always be desirable due to bandwidth usage and power consumption.

To cope with this problem, we have designed a few offloading strategies that reduce the amount of offloaded entities while maintaining some user defined goal. The first strategy is the *target FPS strategy*. This strategy monitors the current FPS of the game and attempts to offload just enough entities to maintain a user specified FPS. The *max bandwidth strategy* does something similar by offloading as many entities as possible within the user specified limits on download and upload rates. The *coarse-grained only* strategy only offloads entities that are very suitable for offloading in the sense that the offloaded functions are computationally heavy, require very little bandwidth to transmit results and also occur with low frequency. Due to the lack of any profiling tools of the framework, it is required by the developers themselves to give information about which offloadable entities (Section 3.3) are coarse-grained. Finally, we have the *offload all strategy*, which simply offloads all possible entities at any given time for maximum performance.

3.6. Limitations

One major limitation of our framework is that it can only be used for games that use discrete ticks to simulate their game world. Although this idea is general enough to be used for many games, some types of games with more continuous worlds, like first-person shooters and action games, tend to take more liberty with their tick lengths. Whether our framework can be modified to be used for these kinds of games is left as a future study.

Another major problem with our current framework is that it cannot recover from a desynchronized state caused by any factor. We have deliberately left this out of our current framework as this problem is complicated enough to warrant a study on its own.

Another thing we have left out is automatic server discovery. We think this problem is fairly easy to solve with existing solutions. Our framework is flexible enough to have a server discovery module build on top of it without much problem.

Our framework currently puts a lot of work in the hands of the game developers. They have to implement the OE interface as well as a game offloading controller. The OE interface requires a significant amount of work and thinking from the developers' part, but should not be a problem for developers who are comfortable with programming regular multiplayer games.

Furthermore, our approach can introduce subtle problems with the game loop locations at which offloaded events are executed. On the server side, offloaded events are executed at the moment they occur in the game loop. On the client side however, the results of all received offloading events are passed to the game at the point the game called the update function of the OF. If the game processes these events immediately, it may create desynchronizations due to the difference in the location in the game loop at which the same event is processed on the server and the client sides. We could not come up with a solution for this other than that the game developers should take this into account when creating OEs. The developers can also consider to temporarily store the received event result on the client side till its game loop reaches the same spot at which the server processed the event.

There is also a problem with how we estimate the TD. We have measured it in ticks, but this causes that the TD value is not only influenced by the ping, but also the TPS of the game. This generally is not a problem when the TPS does not make any wild swings between TD measurements, but can cause inaccuracies. Furthermore, the TPS of the client is not the same as the TPS of the server, so even when the network connection between them is parallel, the TD value might differ depending on which side is measuring it. However, the server likely has similar TPS as the client due to our lock-step scheme.

4

Experimental Setup

In this chapter we will present our experimental setup. In Section 4.1 we describe the game called *OpenTTD* that we have used to test our framework with and why we have chosen to use this game. Section 4.2 explains how we have modified OpenTTD to work with our offloading framework. In Section 4.3 we explain how we have setup various aspects of our experiments, namely the game parameters (Section 4.3.1), offloading parameters (Section 4.3.2), the client and server devices we have used (Section 4.3.3) and the offloading strategies (Section 4.3.4) we have used. In Section 4.3.5 we show what kind of data we record during our experiments and how we record it. Finally in Section 4.3.6 we will combine all the different experimental parameters into experiment sets that we will perform.

4.1. OpenTTD

To test whether our framework and fine-grained computation offloading works, we need an actual game to perform offloading on. For this we have chosen an open source real-time strategy game called *Open Transport Tycoon Deluxe* (OpenTTD) [28]. OpenTTD is a multiplayer simulation game where each player is part of a transport company playing on the same map. The goal of the game is to earn more money than other companies by building infastructure and vehicles to transport various types of goods like passengers, mail and oil.

Figure 4.1a shows a screenshot of what OpenTTD looks like in-game. This screenshot only shows a small portion of the entire game map. In the center is a city that provides opportunities to transport goods from and to it. The colored name labels are stations build by different companies in an attempt to make use of this opportunity. Some yellow colored busses can be seen driving around transporting passengers. The UI buttons are visible on both sides of the screen, which can be used to gain information on objects on the map as well as building and managing the player's own assets.

In OpenTTD, players cannot control entities like cities on the map directly, but instead can only influence them by means of transporting goods. These entities will change on their own as time passes. For example, a city with a lot of goods transported to and from it will grow in size. The player can more forcibly influence a city to a certain extent by changing its roads or demolishing buildings. The entities that the player does have control over are the things that they build themselves. These include the infrastructure, stations and vehicles. Vehicles however find a path and travel on their own after being given orders to transport goods from A to B and back. This results in the player being occupied with only building and managing assets on a fairly high level.

OpenTTD simulates its game world in ticks. The game progreses with the same amount with every tick no matter how much time has elapsed since the last tick. This fulfills the requirement of our framework. The set maximum number of ticks when running the game on regular speed is 33. The version of OpenTTD we use does not render a frame at every one of those ticks. Instead, it only renders a new frame when there is an animation change and the animation is also visible on screen. This results that the maximum frames per second (FPS) of OpenTTD is 33, but it is usually much lower. A result of this is that we cannot accurately measure the FPS of OpenTTD. Instead, we will measure the ticks per second (TPS) of the game, which gives a more consistent result that is independent of

(a) Screenshot of the Android port of OpenTTD build and running on Linux.

(b) Screenshot of the Android port of OpenTTD running on a Nexus 7 tablet computer.

Figure 4.1: Screenshots of the OpenTTD versions we have used.

coincidences of whether there was an animation on screen or not. For other games however, it would be better if we measured FPS instead, as that is actually what the player directly perceives.

OpenTTD is very suitable for our research purposes as it is open source, available on multiple platforms, including for mobile devices, and is also a very sophisticated game. The computational load of OpenTTD is divided among a large number of game objects which need to be constantly updated. These game objects are for example, map tiles, AI players, vehicles, stations, towns and factories. After

updating all objects, the screen also need to be rendered and displayed, which is also computationally heavy. These components provide a variety of offloading opportunities of varying granularity.

We have chosen to use an Android port of OpenTTD made by pelya [29]. In particular, we used the OpenTTD-1.4.4.36 version of pelya's port, which is a port of the regular OpenTTD 1.4.4 version. Figure 4.1b shows a screenshot of this version of OpenTTD running on a Nexus 7 device. This version of OpenTTD allowed us to build to both Android and Linux using the same code, which is very important for our framework to work. The Android version will be used as the client-side and the Linux version will be used as the server-side. It is possible for the Android version to play with the regular Linux or Windows versions, as the only difference between them is how the I/O is done.

4.2. Offloading OpenTTD

For our offloading framework, the server-side will not use the regular server mode of OpenTTD. Instead, it will run the game normally the same way as the Android client as if playing a single-player game. The only difference being that our offloading logic will be run on server mode on the server and client mode on the client. We does it this way because we want our framework to be as indepedent from the game as possible and using the network code of OpenTTD would create a dependency. As the game will be played from the mobile device and not the server, the server-side will run the game without visual and andio output. This significantly reduces the load on the server side, allowing our offloading framework to be used with lighter servers.

To implement our framework for OpenTTD, we have first implemented an offloading game controller that notifies the offloading framework of tick updates as well as providing functions to start, pause, resume and quit the game. As for the OEs, we have decided to let all road vehicle types and all companies to inherit from the OE interface. For road vehicles, we have partioned its logic so the collision detection and the path-planning parts can be offloaded. For companies, we have made sure that if they are controlled by AIs, the AI logic will be offloaded. We have defined the companies to be coarse-grained, as the AIs are likely computationally intensive, but do not send many actions.

A custom message is sent from the server to the client each time an offloaded event of a certain type occurs. For example, when the client offloads a roadvehicle, it will stop updating its collission detection and path-planning logic. Instead, when either code is run on the server side, the server will send a message to the client with the result of that call. The client will store the result till it arrives at the tick the call occurred before processing it.

We have also looked into offloading other parts of OpenTTD, including the path-planning of trains and the stocking of goods at stations. However, both of these functions had too many dependencies to be easily offloaded, hitting one of the problems of our framework. OpenTTD was not made with offloading in mind, so the separation of logic and its result is not always ideal.

We have not performed any profiling of OpenTTD to select the components to offload. Instead, we simply looked at the code looking for code blocks that look computationally intensive, are called often and also can be separated fairly easily. As the code of video games almost always consists of multiple classes with different tasks, this method can work fairly well to partition most video games for offloading given enough knowledge about the game itself.

To synchronize user input when offloading OpenTTD, we made sure that every time a game state changing event has been initiated by the user, a message is sent from the client to the server to synchronize the event. This means that certain kinds of user input are not synchronized, like user input that opens a game window to view information. These actions do not change the game state so do not need to be synchronized. OpenTTD is well structured in this regard as all game state changing events all go through the same function, making adjusting the code relatively easy and efficient.

For the server-side, we have turned off game features that are not neccessary due to the machine running it being used as a server. This means that we could turn off any I/O that were not neccessary without a player, like rendering, audio and polling for user input. We also let the server be able to run the game at more than 33 ticks per second. This allows the server to be able to more rapidly respond to client messages as well as to get into the next tick as soon as possible to reduce the chance of the client needing to wait. In particular, we set the maximum ticks per second to 120 for the server. This number is obtained by testing increasingly higher maximum ticks per second values on the server and looking at the measured TD value on the client. We noticed that if we ran the server at the same speed as the client with a maximum of 33 ticks per second, the measured TD value on the client would range from

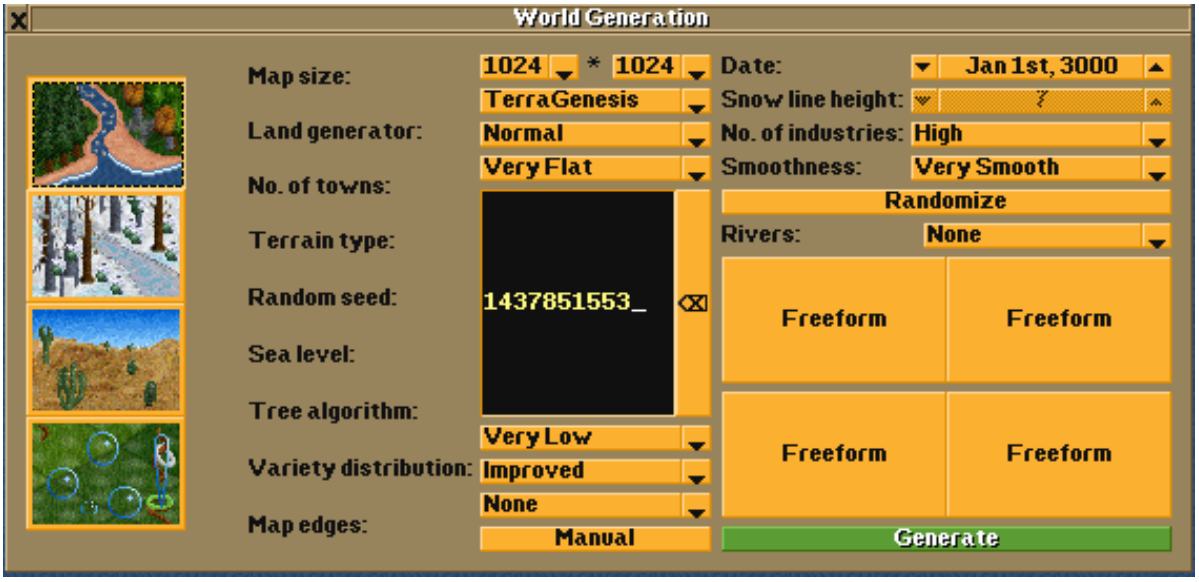


Figure 4.2: The settings used to generate the map used in our experiments as seen from in-game.

1 to 3. As we increased the maximum ticks per second on the server we noticed that the measured TD value on the client would decrease, till it reached a steady value of 1, which is the smallest TD value possible. This happened around the value of 120, so that is the value we have chosen. The reason that the TD value is influenced by this parameter is that the server needs to run the game loop to respond to messages, even though it would not update the game afterwards because the client is lacking too far behind.

4.3. Setup

The main goal of this research is to find out whether fine-grained computation offloading can be a viable solution to increase the performance of mobile games. In this section, we will present how we have set up our experiments to answer this question. In Section 4.3.1 we will explain OpenTTD specific parameters that we have set and the save games we have created to act as experiment starting points. In Section 4.3.2 we will show the parameters that we have chosen that are related to the offloading framework we have created. Section 4.3.3 shows the client and server devices we have used. Section 4.3.4 the parameters we have chosen to test the offloading strategies we have created. Those sections conclude the different experimental parameters we have used. In Section 4.3.5 we show what data we record during our experiments and how we record it. Finally, in Section 4.3.6 we combine all the different experimental parameters into more concrete experiment sets that we will perform.

4.3.1. Game Settings and Save Games

A single game of OpenTTD lasts for many hours and can have different computational loads depending on the amount of entities in the game. To test our offloading framework for different computational loads of OpenTTD, we have decided to create several experiment starting points beforehand by using AI players to gradually fill the map. The idea is that the different starting points we create this way will have increasing computational loads, representing the way the game progresses over time.

To do this, we have first generated a random OpenTTD map using settings that would create a mostly flat map that is easy to play on. Figure 4.2 shows the exact settings used to generate the map. These settings were chosen to give the AI players a relatively easy time and enough space to fill the map with vehicles. The map size of 1024x1024 was chosen based on the number of AIs we are planning to use and our experience with how much space an AI would need. We have chosen a starting year far into the future so in-game technology progression would be eliminated.

After generating the map, we then start running the game and gradually populate it with AI players as in-game time passes. The AIs already in the game will already start filling the map with entities. At

Savegame Name	Road vehicles	AIs	Game Time (days)	Game Time (ticks)
Save 1	267	4	714	52836
Save 2	620	5	1263	93462
Save 3	1519	7	2656	196544
Save 4	2685	9	4212	311688
Save 5	4859	14	7326	542124
Save 6	6337	14	9532	705368

Table 4.1: The save games we have created as the experiment starting points with the numner of offloadable entities per type in them. Also included is the amount of time that has elapsed since the start of the game. One day in OpenTTD is equal to 74 ticks.

certain intervals, we save the game to create a potential experiment starting point we can use later. So all the save games we have created originated from running a single game from an initial map. Using this method, we have obtained a set of save games with increasing computational load, representing how far a single OpenTTD game has progressed.

From all the save games we have created, we have chosen six to experiment things with. As the game progresses and the computational load increases, so does the number of OEs. We have selected the save games based on the number of OEs on the map. Table 4.1 shows the initial number of OEs and the number of AIs in each of the selected save games. The relation between the actual computational load of the save and the number of OEs in it is likely not linear. However, as we have not done any profiling of OpenTTD, we will simply use the number of OEs as an indicator of game progress and computational load.

The OEs of our selected save games range from more than 200 to more than 6000. We have chosen to choose more save games in the lower OE ranges while having less save games in the higher OE ranges. This is to better experiment with a wide range of computational loads while minimizing the number of experiments required. The number of OEs may change slightly during a single experiment due to AI actions and collisions, but does not influence the overall number by much.

The types of AIs we have used are the *OtviAI* version 418 [30], the *SimpleAI* version 10 [31] and the *ChooChoo* AI version 418 [32]. These two AIs were chosen because from our experience, they can fill up a map quite well. The OtviAI and SimpleAI both use a variety of vehicles, but mainly road vehicles, while the ChooChoo AI mainly builds rails and trains. As our offloading of OpenTTD mainly focuses on road vehicles and the collission detection between road vehicles and trains, we think these three AIs together will create save games that are not only suitable to test our offloading framework with, but also represent a game of OpenTTD fairly well.

Due to the limited number of AI players that can be in the game at the same time, Save 5 and Save 6 have the same number of AIs. This is not a problem however, as the difference in the number of roadvehicles between the two saves is still very large. When all the AIs are active, we have used a total of 7 OtviAIs 4 ChooChoo AIs and 3 SimpleAIs. The save games with less than 14 AIs will have a lower number of AIs of each type.

For each save file, we have put the camera at the same position, overlooking the same city in a fairly zoomed out state. Figure 4.1b shows this position for Save 4. This position influences the performance of the game because OpenTTD will spend a lot of time rendering if a lot is visible and moving at that position. We think that the increase in activity at the set position reflects the increase in computational load due to game progress well.

4.3.2. Offloading Parameters

For the fastest response times and smoothness of the game, we have set $TUF = 1$. We have noticed that this does not incur too much of a bandwidth usage.

Due to the lack of any implementation to recover from desynchronization, we have decided to set LTD, UTD and ITD on the safe sides. In particular, we have set LTD as $2 \cdot TD$, UTD as $3 \cdot TD$ and ITD as $2 \cdot TD$. From our initial tests, this resulted in all packets arriving on time.

Furthermore, we run our offloading decision making process once every three seconds. This is more than enough to detect and offload new entities when they are created as well as giving offloading strategies enough opportunity to respond to the changing environment.

We also let the client measure its TD to the server by sending ping packets once every 80 ticks. We noticed that the Internet connection we used did not fluctuate much. This meant that the frequency of these latency measurements does not influence our framework much.

4.3.3. Devices

Device	CPU	GPU	RAM	Resolution
Nexus 3 (Galaxy Nexus)	ARM Cortex-A9 (1.2 GHz, dual-core)	PowerVR SGX540	1 GB	720x1280
Nexus 6	Krait 450 (2.7 GHz, quad-core)	Adreno 420	3 GB	2560x1440
Nexus 7 (2013)	Krait 300 (1.5 GHz, quad-core)	Adreno 320	2 GB	1920x1200

Table 4.2: Specifications of the three client devices we have used for our experiments.

Table 4.2 shows all the client devices we have used for our experiments. The Nexus 3 and Nexus 6 are both smartphones while the Nexus 7 is a tablet computer. Despite its numbering, the Nexus 6 is actually a newer model than the Nexus 7 and boasts better hardware.

We have chosen these devices to test a range of different devices with different specs. The Nexus 3 represents a somewhat older device while the Nexus 6 is a more up-to-date device, with the Nexus 7 somewhere in between. It is important to note that the screen resolution is also of importance here. A higher screen resolution means that more pixels must be rendered. This has significant impact on the performance of the device.

All of the above devices have trouble running OpenTTD at its maximum tick rate when the number of OEs is high. This means that any increase in performance should be easily visible by looking at the average number of ticks per second that the device can simulate with and without offloading.

The clients will connect through the Internet using a wireless Internet connection. The experiments are all performed at the TU Delft. The wireless connection we used is the *eduroam* connection. This is a wireless connection that is only accessible for university staff and students. It is a fairly stable network connection if the signal is strong enough, but can also fluctuate every now and then.

Device	CPU	GPU	Memory	Ping (avg/std dev)
Samsung Q330	Intel i3 M350	GeForce 310M	4 GB	12.129/8.395
Amazon EC2 t2.micro (Frankfurt)	Intel Xeon Processor E5-2676 v3 (1 vCPU)	None	1 GB	23.515/27.437
Amazon EC2 t2.normal (Frankfurt)	Intel Xeon Processor E5-2676 v3 (2 vCPUs)	None	4 GB	20.207/5.395
TU Delft DAS4	Intel Xeon Processor X5650	None	48 GB	15.595/46.646

Table 4.3: Specifications of the four server devices we have used during our experiments.

Table 4.3 shows the server devices. Our experiments only involve a server providing offloading services for only one mobile client at the same time. This means any ordinary computer that can run OpenTTD can be used. It is important however that the server can run OpenTTD much faster than the client can. This is generally easy as even the most modern consumer level mobile devices are still significantly slower than computers of a few years old. Moreover, the server will be running the game without user related IO like rendering and without audio output, significantly decreasing the computational load.

The Samsung Q330 laptop will be used to see how well a non-optimized home computer performs as the server. It is a simple home laptop with very mediocre specs. The two Amazon EC2 servers will be used as the more realistic cloud solution. We use two different Amazon servers to see whether having over capacity has any influence on the offloading process. We have chosen to set up both Amazon servers in the Frankfurt area, as that is the closest area that is available. We also think that in the situation of an actual deployment of our framework, this distance is fairly realistic. Finally, we have the TU Delft's DAS4 system, which is a powerful distributed computation cluster. Although our experiments will not be using the distributed side of it, we will make use of its high computational power.

One problem of the TU Delft DAS4 server is that it is not accessible directly from the Internet. To be able to establish a connection to the DAS4 server, we used a combination of VPN and SSH tunneling on the client device. First, we must connect to the TU Delft *luchtbrug* VPN. This allows us to access the DAS4 through the eduroam network. However, at this point we still cannot connect to the DAS4 using a custom port. We then use an SSH tunnel with the DAS4 to achieve this. We have used third-party

apps to perform both types of connections. We used Cisco's *AnyConnect* version 4.0.05016 to connect to the VPN server and *SSH Autotunnel* version 1.4.7 by Europe Dev. Group to achieve the SSH tunnel. Although we could have just used SSH on the Android through the terminal, we used an app to more conveniently setup each experiment. This roundabout way of connecting might have some impact on the performance, but in Chapter 5 we will see that this difference is very small.

The ping values to the different servers were measured by pinging each of the servers using the Nexus 7 device and the same Wi-Fi connection we will be using during our actual experiments. For each server we sent a total of 240 ping packets with one second between each of them. This ping measurement duration was chosen to more or less correspond to the planned duration of a single experiment, which will be explained in Section 4.3.6.

4.3.4. Offloading Strategies

In Section 3.5 we have presented four different types of offloading strategies, which are offload all, target TPS, max bandwidth and coarse-grained only strategies. For the control tests we disable the offloading framework part entirely to simulate a situation with no offloading at all.

Several parameters must be set for the target TPS strategy and the max bandwidth strategy. A common parameter between the two of them is when they should offload more entities. After some testing around, we have set the following parameters for offloading decision and strategies that give a fairly stable result but also converges quickly to the right amount of offloaded OEs.

As we have mentioned in Section 4.3.2, we run our offloading decision code every three seconds. The offloading strategy is part of this code. Each time the decision code of an offloading strategy is called, it checks whether its conditions have been satisfied to offload more entities. For example, for the target TPS strategy, this happens when the currently measured TPS does not exceed the given target TPS. Only after five successive measurements where this condition has been satisfied will the strategy offload more entities. It will then do so according to the latest measured TPS difference with the maximum TPS given. The larger the difference, the more entities will be offloaded each time. We have set that the target TPS strategy will offload 200 entities for every tick below the target TPS and the maximum bandwidth strategy will offload 90 entities for every 250 B/s download rate below maximum. These parameters were chosen by some trial-and-error to make sure that the strategy does not overshoot the maximum by too much while still converging fast to the right amount of offloaded entities.

What remains for these two strategies is to select a target TPS and a maximum download rate. It is not easy to choose a reasonable target TPS for OpenTTD. Although there has not been extensive user research on the subject of the impact of FPS on the gameplay experience, it is generally accepted that the FPS requirements of a game depends on the amount real-time action in it. For example, a first-person shooter game requires fast and accurate input from the player, so it generally has a high FPS requirement. Claypool et al. [33] showed that an FPS of 30 is generally enough for these kinds of games. OpenTTD on the other hand is a much slower game, real-time strategy game. So we think that a playable FPS for OpenTTD can be lower.

As we have stated in Section 4.1, OpenTTD's has a maximum FPS of 33 and more often than not, will be below that number on purpose. This is caused by OpenTTD not having continuous movements in-game, thus no need to render another frame when nothing has changed. Moreover, the in-game animations are only updated during a regular game tick. As OpenTTD's maximum FPS is limited by its TPS in this way, for the target TPS stretegy, we have decided that a TPS of 20 is a good target in OpenTTD's case. From our experience, OpenTTD is still fairly playable at this speed and the animations are still somewhat smooth.

As for the maximum download rate strategy, we have looked at the bandwidth usage of other RTS games like *StarCraft* and some player reports of the bandwidth usage of *League of Legends*, as well as using our own experience of bandwidhth usage of online games. We have decided that a maximum download rate of 5000 B/s is the limit our framework should be using on a mobile device. As the upload rate of our framework is incredibly low (less than 2000 B/s), we will only consider download rate here.

The offload all strategy and coarse-grained only strategy both do not need any parameters. The former will simply offload all entities at all times and the latter all coarse-grained entities from the start.

4.3.5. Data Recording

During the experiments, we will record a variety of data. Most of the data will be recorded and written to a file by the client itself. For each run, a data point is created every two seconds. Each of these data points contains the following data:

- The number of ticks that has passed in the last second (TPS)
- The last measured transmission delay (TD)
- The total number of times up till now that the client had to wait for server tick updates
- The moving average amount of time each wait lasted
- The moving average download and upload rates.

For each run of the experiment, we also measure the CPU usage of the client. This is done by reading the value of the fourteenth column of the /proc/<pid>/stat file of OpenTTD on the client device. This column contains the amount of CPU time used by a certain process during its run in jiffies, excluding the CPU time used for system calls. For all the client devices we use, one second contains 100 jiffies. We normalized the number obtained from the file by dividing it with the number of seconds OpenTTD was run.

To automate and simplify the extraction of this data, we use an automated script that pulls the CPU usage from the file at the end of a run of an experiment before OpenTTD terminates. This requires that the device is connected through an USB connection and the Android Debug Bridge with the machine that the script will be run from.

Another type of data that will be recorded separately is the power consumption. The equipment to measure this data is only available for the Nexus 3 device. This is because using the equipment requires a modified battery and therefore, requires adjustments to the device itself. The equipment we use to do the actual power measuring is the *Power Monitor* by Monsoon Solutions Inc. [34]. It allows us to directly measure the power draw of the device with a sampling rate of around 5000 Hz. The input voltage we have used for these experiments was 4.6 V, which corresponded to a fully charged battery on the Nexus 3.

4.3.6. Experiment Sets

Set Nr	Client	Server	Strategy	Save	Goals
1	All	None	No offload	All	Baseline
2	All	All	Offload All	All	Performance
3	All	Amazon EC2 t2.micro	TPS + Bandwidth + Coarse	All	Strategies
4	Nexus 3	Amazon EC2 t2.micro	No offload + Offload All + Coarse	All	Power consumption
5	Nexus 3	Amazon EC2 t2.micro	Offload All + Coarse	All	Power consumption with limited ticks

Table 4.4: Description of all the experiment sets and the parameter combinations that they consist of.

Table 4.4 shows the conducted experiments by combining the different parameters mentioned in previous subsections. The experiments are organized into different sets to better differentiate between them. Each combination of parameter will be tested individually by running the game for a short while. Each run of an experiment lasted three to four minutes. Each combination of the Set 1 and Set 2 experiment parameters were run three times each while Set 3, Set 4 and Set 5 experiments were each run only once. We observed from some initial test runs of longer duration that the data we record do not change much over time, as long as it is not too long to give the AIs enough time to build a significant number of additional vehicles. A three to four minute run gives us stable data for a single run while still staying close to the computational load at the start of each save game. However, we also observed that the result of each individual run can differ a bit from each other. This is likely caused by network flunctuations and maybe even other apps running on the client device. We would have liked to have run more than three times for each parameter combination, but due to time constraints we have settled with just three.

To illustrate the idea of the table, the Set 2 experiments consist of the combination of the three different client devices, the four different server devices, the single type of strategy and the six different

saves for a total of 72 different combinations. Each combination will be run three times and each run will last three to four minutes. The combinations of experiments of other sets can be obtained in the same way.

For the Set 3, Set 4 and Set 5 we have chosen to use the Amazon EC2 t2.micro server, as that server is considered not only computationally sufficient, but also realistic as it is in the cloud. This is based on our analysis of the Set 1 and Set 2 results, which were performed first.

Both the Set 4 and Set 5 experiments were to measure the power consumption of offloading. We initially only had the Set 4 experiments, but after analyzing the results from the Set 4 experiments, we have concluded that our setup was slightly flawed. This was because our offloading framework increased the performance of OpenTTD, but OpenTTD would use the saved CPU time to perform more simulation ticks, effectively increasing the amount of work that needs to be done every second compared to not using offloading, thus also effecting the power consumption measurements. You can find a more detailed analysis of this result in Section 5.6. To solve this problem, we have designed the Set 5 experiments, where the experiments of Set 4 were performed again, except that we lowered the maximum number of ticks per second of OpenTTD to the number of ticks per second the Nexus 3 could achieve for each save game without offloading. Normally, OpenTTD has a maximum number of ticks per second of 33, which is achieved by setting the minimum duration of a tick to 30 ms. Table 4.5 shows the ticks per second limits we have set for the Set 5 experiments.

Save	Ticks Per Second	Milliseconds Per Tick
1	26	37
2	26	37
3	21	46
4	15	70
5	9	109
6	7	141

Table 4.5: The ticks per second limitations of the Set 5 experiments. The Ticks Per Second column shows the which ticks per second we tried to achieve while the Milliseconds Per Tick column shows the actual minimum duration of a tick we have set for OpenTTD.

5

Results

In this chapter we analyze and discuss our experimental results to see whether our framework for fine-grained computation offloading is a viable solution to increase the performance of sophisticated mobile games. For this to be true we need to see a significant increase in performance (Section 5.1), without too much of a decrease in game smoothness (Section 5.2) and responsiveness (Section 5.3). We also look at the influence of offloading on the bandwidth usage (Section 5.4) and overall power usage (Section 5.6) of the client. Finally, in Section 5.7 we give a few comments on the consistency of the game state during our experiments.

5.1. Performance

Figure 5.1 shows the average ticks per second results and the CPU usage results from our Set 1 and Set 2 experiments. These two types of data are relevant for our performance analysis.

From the average TPS results we see that our offloading method significantly increases the performance of OpenTTD on all computational loads on all devices. The TPS results of the baseline tests were mostly beneath the acceptable TPS threshold we have determined in Section 4.3.4, while with offloading, it is mostly above this threshold. From this point of view, offloading can make an unplayble slow game acceptable and sometimes even smooth again. The difference between no offloading and offloading becomes increasingly larger as the computational loads increase, which means our solution scales very well. The difference is more than an 100% increase for all devices, although this is only possible when a significant portion of the game can still be offloaded as the game progresses. The game progress of OpenTTD consists mostly of increase in the number of vehicles, stations, infrastructure and cities. As we have offloaded a significant portion of the vehicle code, this seems to have resulted in good scalability.

The impact on performance between the four different servers has been minimal. The slight differences are small enough to be natural tick fluctuations of the device and the network. However, the difference between the DAS4 server and the other servers during the experiments with Nexus 3 has been larger. We think that something went wrong during those experiments that resulted in the lower performance increase. We are planning to redo those experiments to see whether the numbers are correct.

That there is no difference in performance between the servers is good news. This means that a user of a game using our framework can setup and rely on a personal home computer to perform the offloading for a single client. This allows for more flexible use of this technology as well as reducing the ping between the client and the server. This ability fits well with the vision of [35] of using *cloudlet* machines for offloading, which perform services just like real cloud machines, but which are physically in close proximity with the client.

As we have not tested multiple clients connecting with a single server, we do not know what the influence is on the performance when the server is overloaded. Our current research mainly focuses on the client-side effects of offloading. Looking into the server-side is left for future work.

Looking at the CPU usage, we see that the CPU usage of not offloading and offloading is more or less the same. We think that this is not because our offloading setup does not save CPU time, but

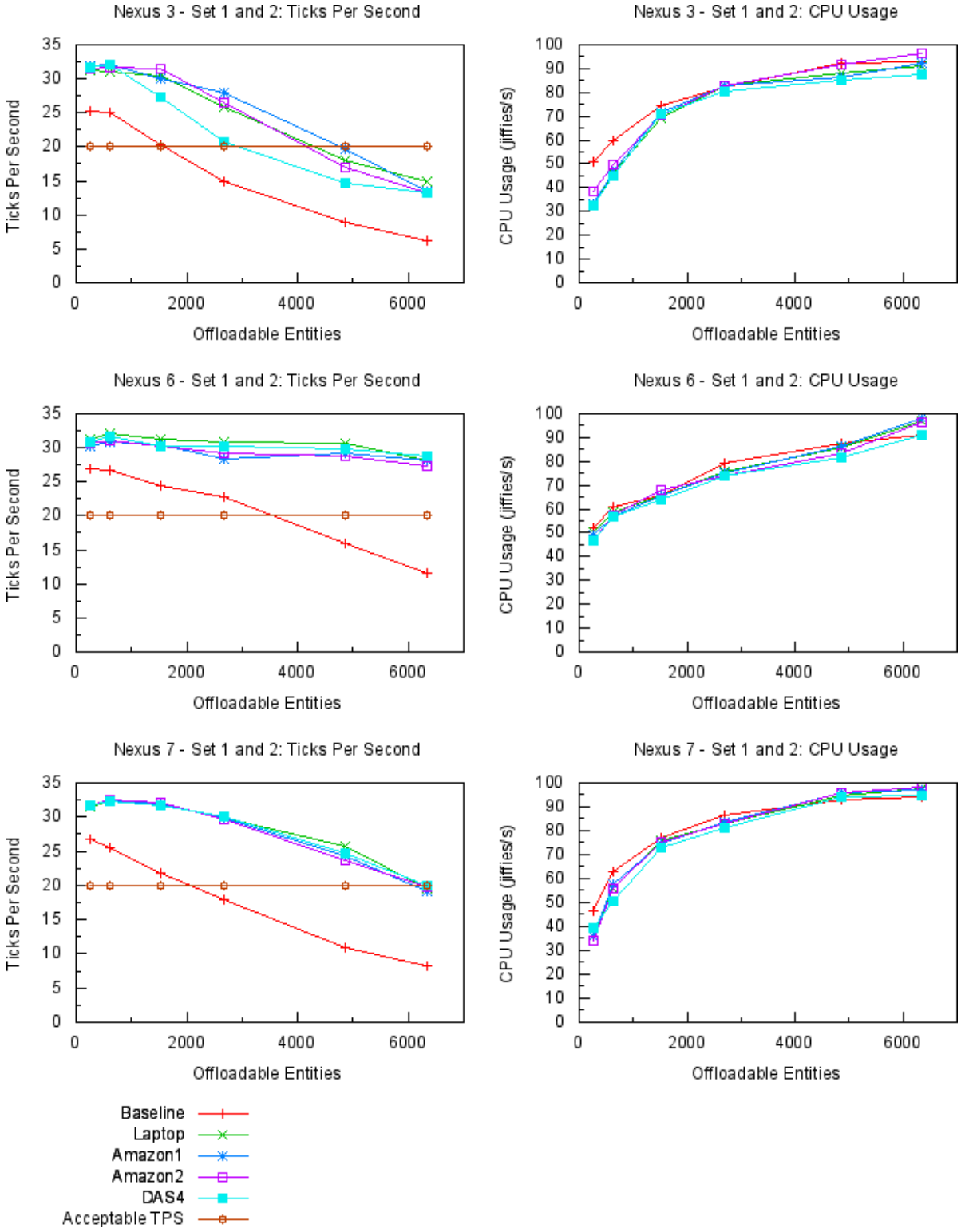


Figure 5.1: The average ticks per second and CPU usage results of the Set 1 (baseline) and Set 2 experiments.

rather that the saved CPU time has been used again by increasing the TPS, effectively increasing the amount of work that needs to be done in the same time frame.

We also see that for the first two computational loads, the CPU usage of the experiments with offloading of both the Nexus 3 and the Nexus 7 devices are lower than the experiments without offloading. We think this is caused because the TPS of those computational loads without offloading is already high. Adding offloading on top increases the TPS to the maximum of 33. The rest of the time saved by using offloading is not further consumed, unlike in the experiments with the higher computational loads where the TPS cannot reach 33 even with offloading. However, we do not know why we do not see the same results that clearly from the results of the experiments with the Nexus 6 device, which even has the highest TPS without offloading.

The question remains why the CPU usage is not 100% when the TPS has not reached the maximum of 33 yet. A 100% CPU on a single core usage should be 100 jiffies a second. None of the devices could reach the maximum TPS of 33 of OpenTTD even at the lower computational loads, despite not reaching 100% CPU utilization either. One reason that we do not see a 100% CPU utilization may be because we only read the application level CPU usage of the proc/<pid>/stat file that did not include the CPU usage of system calls by that application. This means that even when all the CPUs of the mobile client were fully utilized, the value we read could still be lower than 100% of a single core. Our framework does some logging to the flash memory of the device, which are system calls that take some CPU time. However, when we were doing the baseline tests this logging was turned off, so the reason that the CPU usage of the baseline tests did not reach 100% must be due to other I/O operations like rendering. This also explains why the CPU usage increases as the computational load increases; the device spends most of the time simulating a tick, which uses CPU time at the application level and so it does less other types of I/O in the same amount of time.

Works like Li et al. 2001 [21], Chu et al. 2004 [22], Kemp et al. 2010 [5] and Cuervo et al. 2010 [23] were also able to achieve similar performance increase results, although the amount thereof significantly depended on the game and the amount of functions that were offloaded. Unlike these frameworks however, our framework does not need to send an offloading request and wait for its result at every call of an offloaded function. The many benefits of this was explained in detail in Section 3.1. One of the benefits is that we can do fine-grained offloading with this, because even functions that the client could do very fast itself are worth offloading. This is because we do not need to take into account the waiting time and the overhead required to send an offloading request for a call to a single offloadable function.

5.2. Game Smoothness

To analyze the smoothness of the game, we look at the average number of times the client had to wait for the server and the average duration of those waits. Both results are shown in Figure 5.2. A smooth game experience either has a very low number of waits or if it has a high number of waits, we imagine that the game can still be perceived as smooth if the average wait time is consistent and low.

We expected that the number of waits would increase when using a faster client device, due to it being able to simulate more ticks in a shorter time, increasing the likelihood of server tick updates not arriving on time. Although this seems to be the case when comparing the results for the Nexus 6 and Nexus 7, it is not clear why the results for the Nexus 3 show that it has waited more times than both other devices, while being the slowest of the three. This result could be a coincidence caused by the unreliability of network connections and the low number of experimental runs.

Similarly, we also expected that the number of waits would decrease as the computational load increases, because at high computational loads, the client would not be able to simulate very fast. This hypothesis does not seem to be true either as the results show that the number of waits is not really influenced by the amount of computational load at all. Instead, it will mostly stay at the same level for a single client and server combination with random spikes here and there. Although the results of the experiments with the Nexus 3 does seem to show a general downward trend, we do not think that alone is enough to prove the hypothesis.

Lastly, we expected that the number of waits would increase as the distance to the server becomes larger. A larger distance means that packages have a higher chance to become lost or unexpectedly delayed. The results show that there is no relation at all between this distance and the number of waits. This may be caused by the relatively low ping between the client and all of the different servers

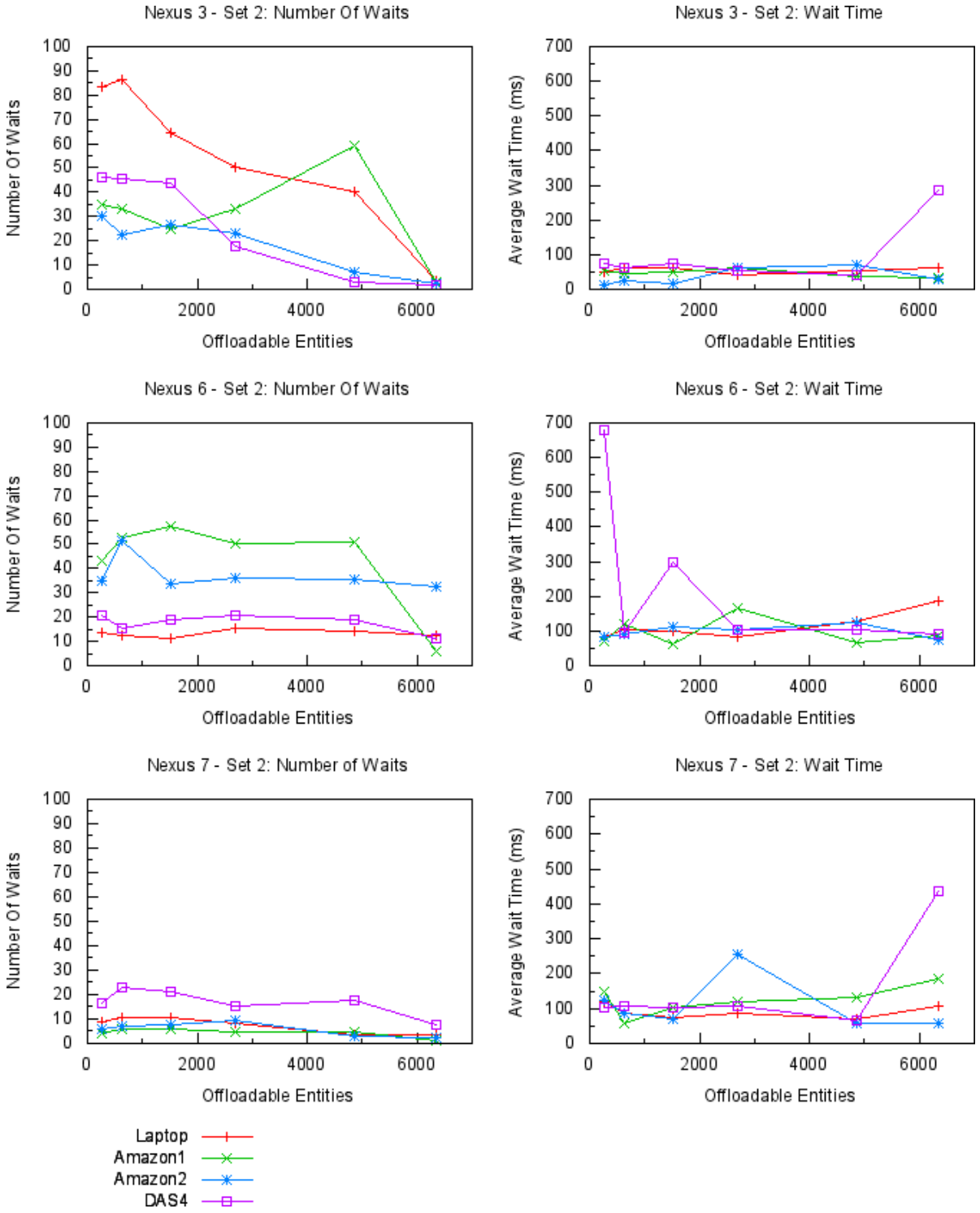


Figure 5.2: The average number of times the client had to wait for the server and the average duration of those wait results of the Set 2 experiments.

we have used. The small diffferences in the ping between these servers may be too small to make a significant difference, especially considering the update speed of OpenTTD is at 30 ms a tick at max, which is slow compared to most modern games which run at 60 FPS, which means around 16 ms a tick.

From the above results, it seems that the number of waits is mainly influenced by the condition of the network connection at that moment. If we look at the results of every individual run of a parameter combination instead of looking at the average of three runs, we also see large variances in results. This means that under the same conditions, the player experience of using offloading can vary wildly. To properly test our hypotheses however, a much larger number of runs must be performed to eliminate these variances.

The number of waits mostly varied from 5 to 60 waits per run. Each run of the experiment lasted nearly four minutes. This means that on average the client had to wait for the server around every 4-50 seconds. We think that a wait every 4 seconds is still an acceptable value, but will also depend on the duration of those waits.

Looking at the average wait time, we see that this data is more consistent than the number of waits, although some spikes are also visible here. Again, there does not seem to be a clear relation between the the average wait time and the client device used, the server device used and the amount of computational load.

With the average wait time, we expected that the computational power of the device itself and the amount of computational load would not influence the average wait time. This is because during a wait, the device will stop simulating the game world, allowing it to poll the network interface at a fast rate for server tick updates. All of the devices would be able to poll at the same rate in a situation with such low computational loads. This hypothesis seems to be confirmed by the results, as the average wait time for all three devices is in the 50-100 ms range.

We did expect that the difference in distance to the different servers would effect the average wait time. In particular, we expected that as the distance to the server increased, the chance of sudden packet delays would be larger. Like the number of waits, the results show that there is no relation between the server we use and the average wait time. Again, we think that this is caused by the minimal difference in distance between the different servers.

Combining the two types of data and thinking from the players' perspective, we think that the waits will certainly be noticeable by the player. Some of the waits can even be fairly annoying as they can be frequent and long enough to be noticed. However, we also think that the average wait time is fairly low so most waits will not really be noticeable. Moreover, we think that in games that have a higher simulation speed than OpenTTD, which is quite normal, the impact of the waits will be smaller. This is because that despite that in such a situation the number of waits might increase, the duration of those waits will decrease, making each wait having less of an impact on the game smoothness and might not even be noticed at all by the player.

5.3. Game Responsiveness

The responsiveness of a game using our framework depends on the chosen LTD, UTD and ITD values. Whatever the values chosen, the minimum delay is directly related to the measured TD value, which in turn depends on the ping between the client and the server and also the processing speed of both machines.

As we already mentioned in Section 4.3.2, for our experiments we have set LTD to $2 \cdot TD$, UTD to $3 \cdot TD$ and ITD to $2 \cdot TD$. From the results of the Set 2 experiments, we have observed that for all our different servers we measured a TD value of mostly 1, sometimes 2. Using these values we can calculate the time in milliseconds the player has to wait to see the result of an input.

In the best case scenario where $TD = 1$ and the client tick is as close to the server tick as possible at LTD, the client will schedule a user input event $4 \cdot 1 = 4$ ticks in the future. Assuming a simulation speed of around 30 TPS for the client, the resulting perceived delay by the player will be at 132 ms. A similar calculation of the worst case scenario with $TD = 2$ and the client UTD number of ticks behind the server, we get $5 \cdot 2 = 10$ ticks input delay. Which results in 330 ms perceived delay. In other words, for our offloading setup of OpenTTD, the perceived delay of the player can range from 132-330 ms.

Video game players start to notice delays above 100 ms [36], which our method greatly exceeds. These response times of our method are worse than those of existing cloud gaming systems [37]. A delay of 330 ms is unacceptable in fast paced games, but for some genres of games like RTS games,

players may be able to adjust to the delay [38]. The user input response time delays created by our framework are generally too high to create a good game experience.

Our delays could be decreased if we chose the margins tighter. Although we have not tested it, we think that in our network conditions, we could have chosen the LTD, UTD and ITD values tighter. However, the LTD value has a minimum value of TD, which is the minimum number of ticks required on the client-side before the client receives a message sent by the server.

Another potential way to increase the responsiveness of our framework is as follows. We explained in Section 3.2 that it might have been a better idea to measure TD in milliseconds in a separate thread other than the game loop of the game. This eliminated the inaccuracy caused by the dependency of TD with the speed of the game. Furthermore, we explained that we could have the server tell the client of its current TPS, so we can then use this value to better predict how far ahead we can schedule user input events. Combining these two ideas results that we can convert the transmission time in milliseconds to the number of ticks that the server can do in that time. This way we can more accurately schedule user input events, which will likely result in better game responsiveness.

All in all, keeping the ping between the client and the server as low as possible will become one of the most important factors when increasing game responsiveness. The possibility of our framework to be able to use home computers as servers is very beneficial in this situation, as those machines are almost always relatively nearby, even when the user is not at home.

5.4. Bandwidth Usage

Figure 5.3 shows the average download and upload rate results of the Set 2 experiments. The data shows that the download rate remains very consistent throughout the different servers and computational loads, but that the upload rate using the Amazon EC2 servers suddenly becomes very unstable as the computational load increases. Looking at the logs of those runs it seems that this is caused by a bug that causes the client to resend offloading requests of entities that are already marked as offloaded. We could not find the cause of this bug nor explain why this does not always happen and if it happens it is with the Amazon EC2 servers. When analyzing the upload rate, we will thus only look at the upload rate results of the experiments using the laptop and DAS4 servers during our analysis.

Claypool et al. [39] have measured the upload rate of each player for the first *StarCraft* game. For eight players in Starcraft, Claypool et al. have measured an upload rate of 8000 B/s for each player. The upload rate of our offloading framework for OpenTTD is only around 300 B/s, which we think is at a very acceptable level. Claypool et al. have not measured the download rate of each player in Starcraft, but we can try to estimate this value. Starcraft is a peer-to-peer game, which means every message of each player should have been sent to all other players. With eight players and an upload rate of 8000 B/s for each of them, this means that every player sends around 1143 B/s to the seven other players. So every player will also receive 1143 B/s from seven players, putting the download rate of every player to 8000 B/s too.

Although we do not know the number of units that were in the StarCraft games of Claypool et al. [39], based on our knowledge of StarCraft, a game with eight players likely does not have more than 1000 units on the map at the same time. Our experiments have far more entities that require network communication. The download rate we measured for our framework when offloading this high number of entities is 12000 B/s. We think that the download and upload rates of our framework are both at very reasonable levels.

Another thing to consider is that users of our offloading framework may often not be using a Wi-Fi connection. Instead, they will be relying on mobile connections like the 3G network. Our bandwidth usages are well within the capabilities of 3G networks. With the advance of 4G networks, these levels of bandwidth usage should not be a problem at all. The increase in the number of free Wi-Fi hotspots in public areas is also beneficial for future use of offloading.

Looking at the upload rate results, we further see that the upload rate goes down as the computational load increases. This is because the upload rates of the client in the Set 2 experiments were mainly dominated by client tick update messages. A higher computational load means a lower simulation speed, which results in less tick update messages being sent, causing the average upload rate to slightly decrease.

Although a lower simulation speed also decreases the rate at which offloaded events happen and thus the required download rate on the client-side, we still see an increase in download rate as the

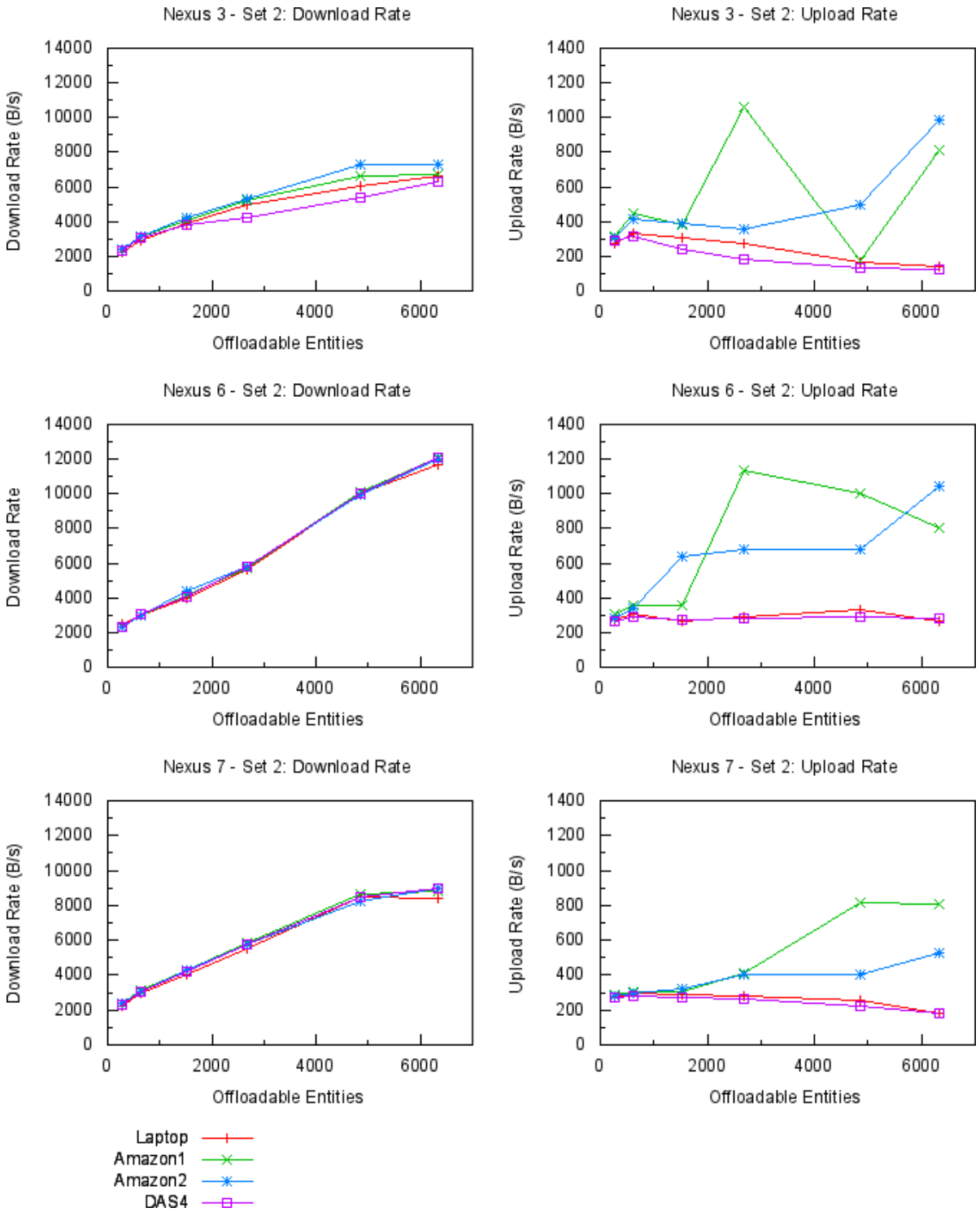


Figure 5.3: The average download and upload rate results of the Set 2 experiments.

computational load increases. This is likely because the extra download rate required to synchronize more OEs outweighed the decrease in download rate due to a slower simulation speed. This is more apparent in the experiments with the Nexus 6 device, where the overall TPS does not decrease by much even at the highest computational load. For the experiments with the Nexus 3 and Nexus 6 devices, we see that the download rate barely increases when going from the computational load of Save 5 to Save 6.

In the end, the required download rate of our offloading framework mainly depends on the size and frequency of the offloaded event messages. This is turn depends on the game itself and how optimized the network code in with respect to message sizes. The download and upload rates will also slightly depend on the selected tick update interval. OpenTTD only runs at 33 TPS. When offloading a game that runs at 60 TPS, which is pretty normal for modern games, the bandwidth usage caused by the tick update messages will increase significantly. We think that even in this case, setting the tick update inteval to 1 will still result in a very manageable bandwidth usage.

5.5. Offloading Strategies

The Set 3 experiments were set up to test the three different offloading strategies we have created. Section 4.3.4 goes into detail about the parameters we have chosen for each strategy. Basically, we have set the target TPS of the target TPS strategy to 20 and the max download rate of the max bandwidth strategy to 5000 B/s.

Figure 5.4 shows the results of the Set 3 experiments along with the results from the Set 1 experiments with the control (no offload) and the offload all strategies. Despite the different lines in the graph looking chaotic at first and some of the data being slightly unstable due to the low sample size, the results do make sense at closer look. During the calculation of the average ticks per second of the Set 3 experiments, we have only taken into account the data points after the corresponding strategy has converged to a stable number of offloaded entities. The time the strategies need to converge range from 30 to 120 seconds depending on the save game. We think that this convergance time is acceptable, as a playing session of a sophisticated game tend to be much longer than a simple game. That said the convergence time can be more optimized in the future by automatic profiling of the program to get a better estimation of the effects of offloading a certain entity.

We have decided to only look at the average ticks per second and the download rate to compare the strategies. This is because we think that in a realistic scenario, the player would be making the tradeoff between these two parameters. On the one had the player wants the best gaming experience with the highest ticks per second possible but on the other hand might not want to use too much bandwidth as it may come with additional fees. Ideally, one would also consider the power consumption, but we could not develop a strategy for this as not every device has an easy way to accurately measure power consumption.

The results of the target TPS and the max bandwidth strategy both show that they do what they are supposed to do. Both of them only offloads entities when their respective conditions are not met and both of them tries to maintain just enough offloaded entities to satisfy their conditions. From the results we see that the target TPS strategy is slightly better at keeping the average ticks per second close to the target than the max bandwidth strategy keeping the download rate close to the max. However, both of them manage to maintain it fairly well. Which one is better mostly depends what the player values the most; a high ticks per second or a low download rate. The results show that these two strategies are customizable enough for the player to use the mand making the tradeoff.

The results of the coarse-only strategy is interesting, as it shows that in the early save games, the effects of this strategy in increasing the performance is very good, almost the same as offloading all entities in fact. But as the number of entities increase, the effects of offloading only the coarse entities drops rapidly, almost to the point that there is barely any difference between offloading them and not offloading them. We think that this results is an OpenTTD specific phenomenon. It says something about the ratio between how much of the computation is cause by coarse entities and how much by regular entities. How well this strategy performs will thus depend on the game itself. That said, we think that in a practical scenario, this strategy will do a very good job at minimizing the bandwidth usage while maximizing the ticks per second.

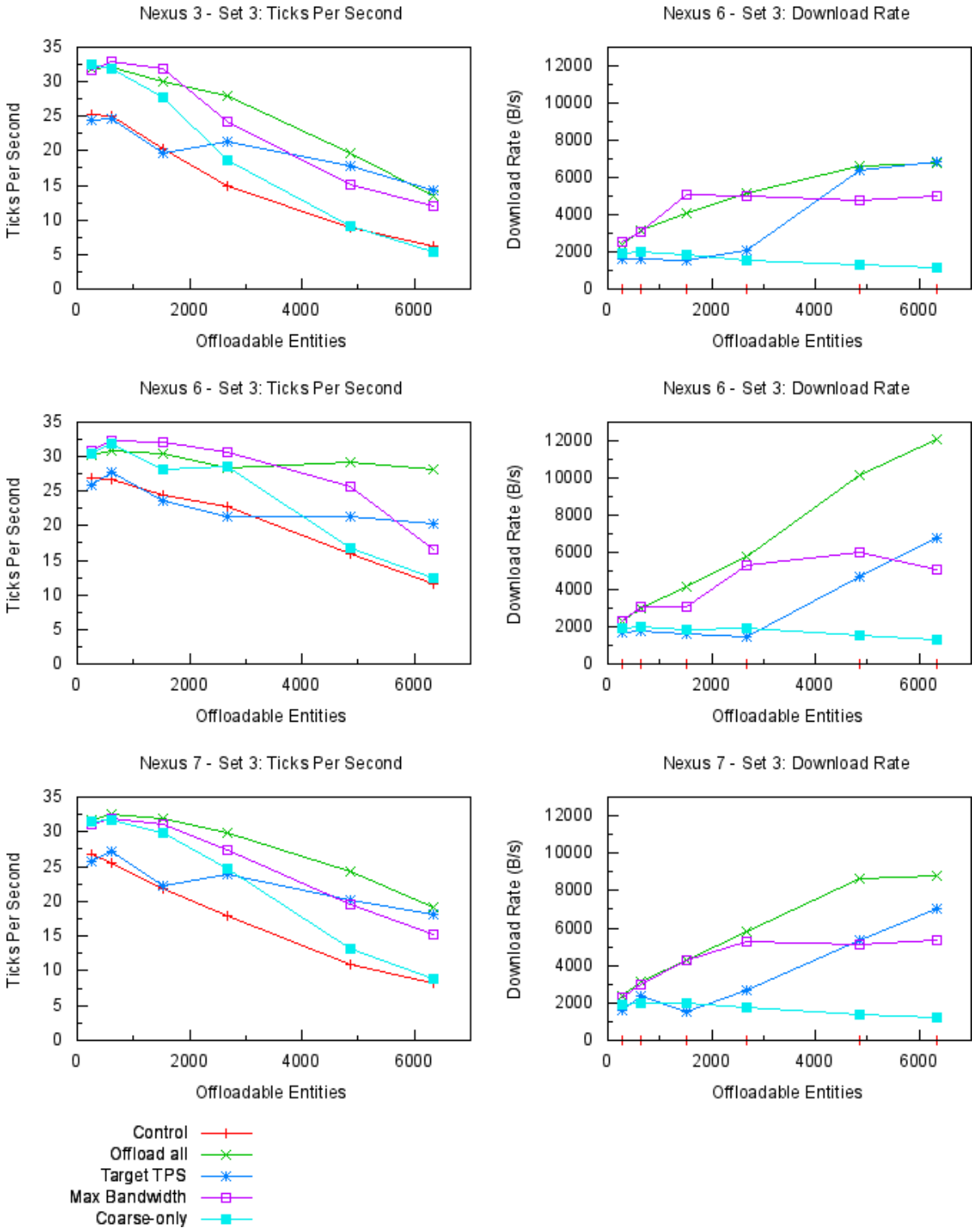


Figure 5.4: The ticks per second and download rate results of the Set 3 experiments using the target TPS, max bandwidth and coarse-only strategies. The baseline results from Set 1 and the results with the offload all strategy of Set 2 are also included.

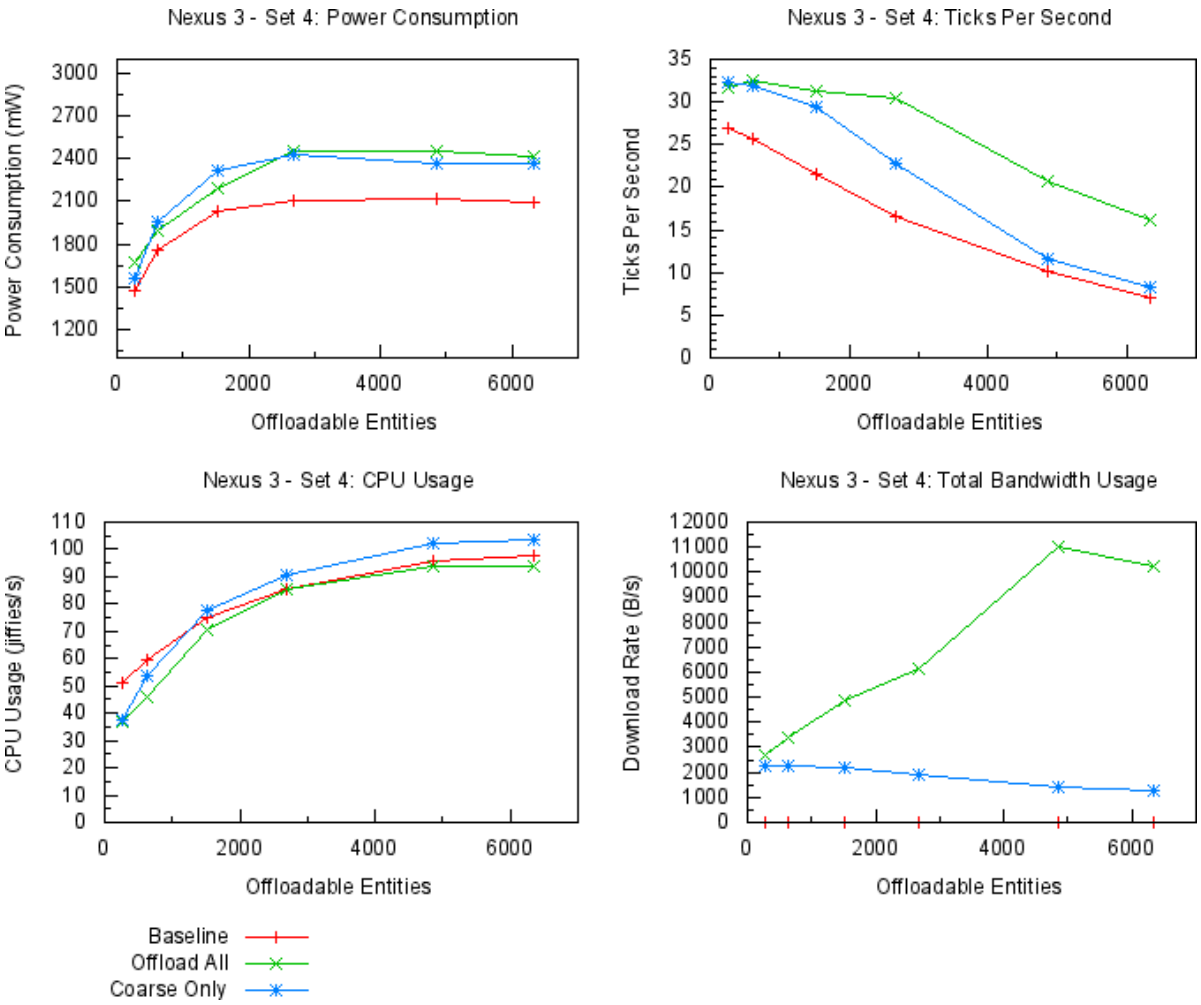


Figure 5.5: The various results from the Set 4 experiments measuring the power consumption of the Nexus 3 device with offloading.

5.6. Power Consumption

Figure 5.5 shows the results from the Set 3 experiments measuring the power consumption of the Nexus 3 device with offloading. Aside from the power consumption itself, we have also included the ticks per second, CPU usage and the download rate data from the experiments. These will be analyzed to see how they relate to the power consumption itself.

From the results, we see that using offloading has significantly increased the power consumption of the device by around 20%. This is most certainly caused by the additional usage of the network hardware of the device when using offloading compared to not using offloading.

However, we expected that offloading would decrease the overall power consumption as it was the case for other frameworks [21, 23]. We thought that if offloading could decrease the CPU usage, the power saved by using less CPU might outweight the extra power used by using the networking hardware. From the results in Figure 5.5 we see that the CPU usage when using offloading is only lower than the control for the first two computational loads, which is in line with what we have seen from the Set 1 experiments results in Section 5.1. In that Section we also explained that the cause of this is that OpenTTD uses the saved CPU time from offloading to perform more ticks per second, effectively increasing the overall workload each second.

We thought that if we forced the ticks per second of the experiments of using offloading to stay the same as in the control experiments, we would get a more accurate measurement of whether offloading can save power. To test this, we have set up the Set 5 experiments as explained in Section 4.3.6. Figure 5.6 shows the results of these experiments.

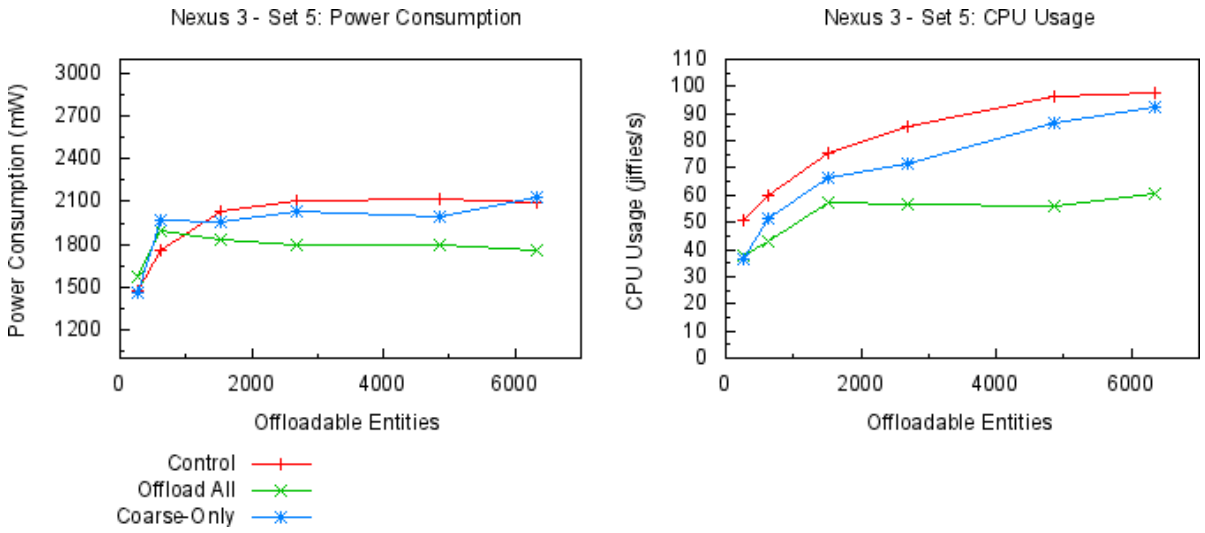


Figure 5.6: The results from the Set 5 experiments measuring the power consumption of the Nexus 3 device with offloading while limiting the maximum number of ticks of OpenTTD.

From these results, it is clear that offloading can save a significant amount of batter power when the workload remains the same. This also shows that it even offloading fairly fine-grained functions like the path-planning and collission detection of vehicles in OpenTTD is benificial to the power consumption of the device. This means that with our framework, it is very easy to find functions that would benefit from offloading

However, the results of the first two save games show that there needs to be enough computation to be offloaded before offloading can save power. In our experiments, the number of offloadable entities in the first two save games are relatively low. The amount of computation that can be offloaded through them compared to the rest of the game that cannot be offloaded is also low. This has resulted that the power consumption of using the Wi-Fi of the device even at low bandwidth usages exceeds the amount of power saved by reducing CPU usage. This does not mean one should never offload in these situations though, as we have seen in 5.1, offloading will still significantly increase performance in these situations.

Another observation from the results is is that the power consumption of the Wi-Fi interface likely uses significant amount of power by just turning it on, while actually using it with increasing download and upload rates only marginally increases its power consumption. This is based on the increased power consumption of offloading at the first two save games, the decreased power consumption in the latter save games and the difference in power consumption of the offload all and coarse-only strategies. If this is true it would mean that it will often be very beneficial to offload as much as possible. Further research is needed to determine how much power different network interfaces of a mobile device consume at different download and upload rates.

Combining the results of the Set 4 and Set 5 experiments, we can conclude that offloading for mobile games will only save power when the game does not add more work because of the increased performance due to offloading. This is only true when the device can run the game at or almost at the maximum number of ticks set for the game. This means that our framework either increases performance by increasing the number of ticks per second of the game while consuming more power, or the performance of the game will more or less stay the same but using less CPU and thus power.

5.7. Game Consistency

As we have previously stated, our current implementation of the offloading framework cannot recover from desynchronizations caused by late or faulty messages. To be able to use the framework despite these shortcomings, we have chosen relatively large margins for LTD and UTD. For each arrived message we check whether it has arrived too late and if so, record it in the data. From all our experiments, we have not detected any such messages. Most desynchronizations in OpenTTD would also result in a crash rather soon, which would have been noticed immediately during the experiments.

Our reasoning with the TD are based on a fairly pessimistic expectation of the network connection. More research is needed to see whether the margins can be chosen tighter.

6

Conclusions and Future Work

In this chapter, we will first summarize the work we have done and the results we have gotten in this thesis in Section 6.1. We will then discuss potential future research directions for offloading for mobile games in Section 6.2.

6.1. Conclusions

In this thesis, we have investigated the viability of performing fine-grained offloading to increase the performance of sophisticated mobile games. We have done so by first establishing what is required of such framework and analyzing the current state of the art in offloading. We have discovered that existing offloading frameworks do not meet the requirements that comes with fine-grained offloading for games. We then created our own offoading framework that uses a combination of offline and online partioning to achieve dynamic and fine-grained control of the offloading process. The framework automates the synchronization process by using our own variation of the lockstep system. The framework also defines class interfaces to abstract the framework from the actual game and to guide and simplify the implementation process for game developers. Finally, our framework supports different offloading strategies that enables smart guidance of the dynamic offloading process depending on the situation.

Our experiments were performed by implementing our framework to an existing game called OpenTTD. We have done this by implementing the interfaces defined by our framework. We partioned OpenTTD by making offloadable entities out of OpenTTD's roadvehicles and AI companies. For roadvehicles we were able to offload its collission detection and path-finding functions and for AI companies, we could offload its whole decision making process.

We have performed experiments using three different mobile devices acting as clients, four different machines acting as servers, six different save games, representing increasing degrees of progress of an OpenTTD game with increasingly higher computational loads, and four different offloading strategies. During these experiments we have measured a vareity of data that we could use to analyse the performance of our framework. These types of data include the number of simulation ticks per second, the CPU usage, the download and upload rates and the number and duration of the time the client had to wait for the server. Additionaly, we have also measured the influence of offloading for the power consumption of the device. We could only do this for one of our devices, namely the Nexus 3 device, as it was the only device we had a setup for.

The results from the experiments show that our framework performing fine-grained offloading significantly increases the performance of OpenTTD across all computational loads. The increase in performance is much larger when the computational load is high, showing that our framework scales up pretty well. The number of ticks per second with offloading compared to not using offloading can become as high as 250%, which significantly increases the playability of the game when either the device is not as powerful or the computational load is exceptionally high or both. The results also show the computational power of the server is not of that importance when it is only providing offloading services for a single client. The only requirement for the server is that it should be able to simulate the game faster than the mobile client, which is easily achieved as even old computers can outperform the most modern commercial smartphones and tablet computers. The results show that even a simple

home laptop can be easily setup and used as an offloading server with equally good results compared to using more powerful machines. Using computers nearby the user also has the advantage of lowering the ping between the client and the server.

Our framework also allows guided control over which entities to offload and how many using offloading strategies. The offload all, target TPS, max bandwidth and coarse-only strategies we have created show that it is possible for the user to tune these strategies to his liking and can choose between them to make a prefered trade-off between performance and bandwidth usage.

We also see that using our offloading framework can also decrease the power consumption of the device. However, this is only possible when the game itself does not use the saved CPU time from offloading to actually increase the simulation speed of the game. This situation is possible when the device can already run the game at nearly its maximum simulation speed without offloading. When the device cannot do this however, performing offloading both increases the simulation of the game as well as the power consumption of the device. This is caused by that the CPU usage remains the same while the device also needs to make use of its network hardware to perform offloading.

Offloading makes a distributed system out of a game and also requires the use of Internet. This does come with a few drawbacks. Our framework is no exception. The bandwidth usage of our framework is fairly manageable even when the number of offloaded entities is very high and should be even less a problem as mobile networks keeps developing in the future. Due to the design of our offloading framework, it is possible that the client must sometimes pause the game and wait for the server. This can create short hiccups in the game that can be fairly annoying to the player. Our results show that these hiccups exist in our framework and are sometimes long enough to be noticeable by the player. However, we also think that the average wait time is low enough that most of these waits will not annoy the player. A more serious problem is the long responsive times for user input caused by our framework, which can be as high as 330 ms for our current implementation with OpenTTD. Although we think that both wait times and input response times can be improved in the future by more accurately prediction of the network condition. The possibility of using home computers as offloading servers also helps minimizing the ping to the server. Moreover, we think that using a game that has a higher simulation speed than OpenTTD, which should be pretty common nowadays, will also significantly alleviate these problems.

Despite these drawbacks, we think that performing fine-grained offloading for sophisticated mobile games has significant potential to be used in the future. Our current implementation of the framework is very experimental and only includes the core features, but even it can drastically improve the performance of the game. Our current framework can easily be improved to be less experimental and more user-friendly by adding features like automatic sending of save games to the server, being able to input the server address in-game, being able to change offloading parameters in-game and creating a server side program that relaunches after an offloading session. These changes are simply implementation problems and do not require further scientific research.

That said, even after our experiments and analysis, there are still many open questions regarding the optimization of the core of the framework as well as the influence of offloading on the client.

6.2. Future Work

The first potential research direction is whether the margins of our lockstep scheme can be chosen tighter and more smartly. Our current implementation uses a rather pessimistic view of the Internet, especially when scheduling user input events. Further research is needed to determine how many ticks the known server tick of the client is behind the actual server tick in practice. Furthermore, more investigation can be done to see how accurate our measurements of TD actually are to the actual time needed to send and receive a packet. We also concluded that the accuracy of our TD measurements is influenced by the simulation speed of the, but also does not consistently increase as simulation speed increases as there can be a point where a measurement package arrives just after a single tick. Further research is needed to determine what the influence is of the simulation speed on the accuracy of TD measurement and whether there is an alternative way to predict message traveling times so it can be used to schedule in-game events. Furthermore, our TD measurement is done at the client-side, which does not give us a good estimate of how many ticks ahead we should schedule user input events at the server-side. This could potentially be solved by letting the server also calculate its TD and sending the value to the client for use.

The framework also needs to be thoroughly tested under different network conditions. Although our framework is designed for mobile use, we currently have only tested it with relatively stable Wi-Fi connections. During our experiments, we noticed that the stability of this connection significantly impacts the performance of our framework. It is interesting to see how our framework performs under other mobile networks like 3G and 4G, which most users of the framework will eventually use. Research that have compared Wi-Fi and 3G networks have discovered that 3G uses more power than Wi-Fi [40]. Moreover, works that measure the power consumption of mobile network interfaces like Balasubramanian et al. [40] and Perruci et al. 2011 [41] only measure the power consumption of the interface when downloading or uploading a static amount of data at a certain transferring speed. The situation of gaming is different as increasing the transferring speed does not shorten the time needed to finish the job. Therefore, reserach is needed to see what the power consumption is of different network technologies when transferring data at a certain speed for a static amount of time. Both of the above aspects have implications for the point when offloading will either increase or decrease power consumption. Experiments to test the framework under network conditions can also be useful to determine what kind of networks can support a playable offloading experience and which ones cannot. These experiments can also determine what the maximum ping is of a playable session and thus how close the server must be to the client. This will have influence on the deployment of the service by potential game developers that want to use offloading.

The framework must also be made robust against temporary network outages and desynchronizations to be of any practical use. Currently, a disconnection will pause the game for both the client and the server without a possibility to recover. This is not acceptable in a practical case. The client should at least try to reconnect to the server after reobtaining a network connection. If the network outage lasts too long, the client should give up entirely on the current session, create a new save game of the current game state to try again later. The server should also quit its session after a certain time has elapsed.

A more difficult question is how to cope with desynchronizations. Desynchronizations can be caused by late or lost packets or wrongly implemented offloadable entity interfaces. The result of a desynchronization is also very game specific, so it is very hard to solve the problem in a very general way. The question here is not only how to handle a desynchronization, but also how to detect it in the first place. Of course, desynchronization from a late packet is easy to detect, but desynchronization caused by lost or faulty implementation are less trivial.

It might be possible to let the both the client and the server create save games at certain intervals and comparing them either directly or through hash. The question is whether all relevant state can be captured in a regular save game or not and if not, how much extra work does the developer need to do to incorporate the missing state in the save game. Another question is how much of a performance impact regularly making save games will have on the client. Creating a save game is a pretty heavy operation for a game and will create a significant slowdown each time the operation is performed. It is possible to do this in parallel, but this will require even more work from the game developers.

Even when you can create save games without disturbing the regular gameplay, the next question is what to do after detecting a desynchronization. The good news is that the same save games could be used as rollback points. After detecting a desynchronization, both the client and the server will then rollback to the last save game that was still synchronized. However, this will have a huge impact on the gameplay experience of the user, as not only must the regular gameplay be interrupted to perform the rollback, the player will also lose some progression. How seamless the rollback can be performed will unfortunately again fall into the hands of the game developer, as the framework does not directly control the state of the game. The interval of the save games must also be carefully chosen to minimize the loss of player's progress when rolling back after a desynchronization while keeping the performance cost under control.

Our current framework gives game developers relatively a lot of freedom to structure their game however they want. However, as we have seen above, to generalize solutions it might be neccessary for the offloading framework to have more control over the structure of the game. This way, it will have more control over the state of the game and be able to generalize solutions for creating save games and coping with desynchronizations. Therefore, it might be interesting in the future to try to come up with a more complete framework. One that not only controls the offloading part, but also controls the whole gameloop and the game state. It will be interesting and challenge to see whether

such a framework can be designed in such a way to still give developers plenty of freedom to create their own games.

A solution must also be found for how to support multiplayer games. OpenTTD is actually a multiplayer game, but our framework only works for single player games. The challenge of offloading in a multiplayer session is how to schedule user input events in the future so all parties, including the offloading servers, is able to receive the messages on time to perform them at the same tick. This requires a complicated communication scheme where all parties must know the ticks of all other parties as well as the transmission delay of each party. It will be interesting to see whether the current core of our framework can be adjusted to enable multiplayer sessions with offloading and whether it is still worth doing offloading with the extra synchronization overhead.

Our current research has mainly focused on the effects of fine-grained offloading on the client side. For commercial use, it will also be neccessary to see what the requirements are on the server side. Specifically, it will be interesting to see how many clients a single server can offload for and what happens when the server is overloaded. Such experiments might be hard to perform with real mobile devices. It might then be neccessary to use virtual machines to simulate a high and varying number of mobile clients. As the focus of such experiment will be on the server side, such a method will probably still give valid results. However, it would also be interesting to see what the effects are on the client side when using an overloaded server. These aspects all need to be investigated before the framework can be deployed for commercial use.

Other research directions include looking into the possibility of doing more than just computation offloading. Other resources that could potentially be offloaded are graphics, memory, disk and network. Out of these, we think that the most promising and most beneifical resources to offload for mobile games are graphics and network offloading. Graphics is similar to computation offloading, but involves reducing the GPU usage instead of the CPU usage. As the computational power of the graphics hardware on a mobile device is also fairly limited, we think that being able to properly offloading graphics will significantly increase the amount of games mobile devices can play. Network offloading involves reducing the bandwidth usage of the mobile device by gathering and preparing data for it. From a gaming perspective, this can be useful in peer-to-peer games where the mobile client is only connected to and uses its network offloading server as a proxy as a peer in the network. This way, it might be possible for the server to preprocess, filter and reduce the size of all the incoming messages before passing them to the moblie client. The other way around, the offloading server can help multicasting messages the mobile clients wants to send to its peers, so the client itself does not need to send them to its peers one at a time.

There is also an interesting opportunity to extend the game running on the server side with addtional features, like logging and the detection of in-game achievements. These types of features can benefit from the fact that the game state at the server represents the game state at the client. Such an extension can in principle be added automatically, for example, through load-time instrumentation. The benefit of doing this is that it does not cost extra resources at the client side. A feature such as logging can be useful for diagnosis purposes, or for continuous model inference as in Vos et al. 2014 [42] for the purpose of mining test cases.

Our current framework requires that the game using it simulates its state in ticks of equal length. In these kinds of games, the state may not change between two ticks and the delta-time of each tick is the same no matter how much time has past since the last one. Most modern games however, use a more dynamic game loop. These games change their delta-time according to the time elapsed since the last tick. This allows the state to be changed in a more continuous way, allowing slower machines to simulate the game at the same speed as a faster one, although with a lower frame rate. Supporting offloading for these types of games will be a big step towards having a framework that can work for all types of games.

Bibliography

[1] M. Claypool and K. T. Claypool, *Latency and player actions in online games,* Commun. ACM **49**, 40 (2006).

[2] M. Haggerty, *The state of mobile game development,* http://gameindustry.biz (2012).

[3] N. Fernando, S. W. Loke, and W. Rahayu, *Mobile cloud computing: A survey,* Future Generation Comp. Syst. **29**, 84 (2013).

[4] K. Kumar, J. Liu, Y. Lu, and B. K. Bhargava, *A survey of computation offloading for mobile systems,* MONET **18**, 129 (2013).

[5] R. Kemp, N. Palmer, T. Kielmann, and H. E. Bal, *Cuckoo: A computation offloading framework for smartphones,* in *Mobile Computing, Applications, and Services - Second International ICST Conference, MobiCASE 2010, Santa Clara, CA, USA, October 25-28, 2010, Revised Selected Papers* (2010) pp. 59–79.

[6] R. Kemp, N. Palmer, T. Kielmann, F. J. Seinstra, N. Drost, J. Maassen, and H. E. Bal, *eyedentify: Multimedia cyber foraging from a smartphone,* in *ISM 2009, 11th IEEE International Symposium on Multimedia, San Diego, California, USA, December 14-16, 2009* (2009) pp. 392–399.

[7] M. Weiser, *The computer for the 21st century,* Scientific american **265**, 94 (1991).

[8] M. Satyanarayanan, *Pervasive computing: vision and challenges,* IEEE Personal Commun. **8**, 10 (2001).

[9] M. Satyanarayanan, *Fundamental challenges in mobile computing,* in *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996* (1996) pp. 1–7.

[10] R. K. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H. Yang, *The case for cyber foraging,* in *Proceedings of the 10th ACM SIGOPS European Workshop, Saint-Emilion, France, July 1, 2002* (2002) pp. 87–92.

[11] X. Gu, A. Messer, I. Greenberg, D. S. Milojicic, and K. Nahrstedt, *Adaptive offloading for pervasive computing,* IEEE Pervasive Computing **3**, 66 (2004).

[12] S. Goyal and J. Carter, *A lightweight secure cyber foraging infrastructure for resource-constrained devices,* in *6th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2004), 2-10 December 2004, Lake District National Park, UK* (IEEE Computer Society, 2004) pp. 186–195.

[13] R. K. Balan, D. Gergle, M. Satyanarayanan, and J. D. Herbsleb, *Simplifying cyber foraging for mobile devices,* in *Proceedings of the 5th International Conference on Mobile Systems, Applications, and Services (MobiSys 2007), San Juan, Puerto Rico, June 11-13, 2007*, edited by E. W. Knightly, G. Borriello, and R. Cáceres (ACM, 2007) pp. 272–285.

[14] K. Yang, S. Ou, and H.-H. Chen, *On effective offloading services for resource-constrained mobile devices running heavier mobile internet applications,* Communications Magazine, IEEE **46**, 56 (2008).

[15] A.-C. Olteanu and N. Ţăpuş, *Offloading for mobile devices: A survey,* UPB Scientific Bulletin (2014).

[16] M. A. Khan, *A survey of computation offloading strategies for performance improvement of applications running on mobile devices,* J. Network and Computer Applications **56**, 28 (2015).

[17] M. A. Hassan and S. Chen, *Mobile mapreduce: Minimizing response time of computing intensive mobile applications,* in *Mobile Computing, Applications, and Services - Third International Conference, MobiCASE 2011, Los Angeles, CA, USA, October 24-27, 2011. Revised Selected Papers* (2011) pp. 41–59.

[18] H. Flores and S. Srirama, *Adaptive code offloading for mobile cloud applications: Exploiting fuzzy sets and evidence-based learning,* in *Proceeding of the fourth ACM workshop on Mobile cloud computing and services* (ACM, 2013) pp. 9–16.

[19] G. Huerta-Canepa and D. Lee, *A virtual cloud computing provider for mobile devices,* in *Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond* (ACM, 2010) p. 6.

[20] X. Zhang, S. Jeong, A. Kunjithapatham, and S. Gibbs, *Towards an elastic application model for augmenting computing capabilities of mobile platforms,* in *Mobile Wireless Middleware, Operating Systems, and Applications - Third International Conference, Mobilware 2010, Chicago, IL, USA, June 30 - July 2, 2010. Revised Selected Papers* (2010) pp. 161–174.

[21] Z. Li, C. Wang, and R. Xu, *Computation offloading to save energy on handheld devices: a partition scheme,* in *Proceedings of the 2001 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES 2001, Atlanta, Georgia, USA, November 16-17, 2001* (2001) pp. 238–246.

[22] H. Chu, H. Song, C. Wong, S. Kurakake, and M. Katagiri, *Roam, a seamless application framework,* Journal of Systems and Software **69**, 209 (2004).

[23] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, *Maui: making smartphones last longer with code offload,* in *Proceedings of the 8th international conference on Mobile systems, applications, and services* (ACM, 2010) pp. 49–62.

[24] C. Huang, K. Chen, D. Chen, H. Hsu, and C. Hsu, *Gaminganywhere: The first open source cloud gaming system,* TOMCCAP **10**, 10 (2014).

[25] O. Soliman, A. Rezgui, H. Soliman, and N. Manea, *Mobile cloud gaming: Issues and challenges,* in *Mobile Web and Information Systems - 10th International Conference, MobiWIS 2013, Paphos, Cyprus, August 26-29, 2013. Proceedings*, Lecture Notes in Computer Science, Vol. 8093, edited by F. Daniel, G. A. Papadopoulos, and P. Thiran (Springer, 2013) pp. 121–128.

[26] Y. Lee, K. Chen, H. Su, and C. Lei, *Are all games equally cloud-gaming-friendly? an electromyographic approach,* in *11th Annual Workshop on Network and Systems Support for Games, NetGames 2012, Venice, Italy, November 22-23, 2012* (2012) pp. 1–6.

[27] P. Bettner and M. Terrano, *1500 archers on a 28.8: Network programming in Age of Empires and beyond,* Presented at GDC2001 **2**, 30p (2001).

[28] Various, *OpenTTD,* www.openttd.org/ (2016).

[29] S. Pylypenko, *OpenTTD Android port,* www.wiki.openttd.org/Compiling_and_installing_the_unofficial_Android_port (2014).

[30] Mainthebox, *OtviAI (version 415),* www.tt-forums.net/viewtopic.php?t=39707 (2013).

[31] Brumi, *SimpleAI v10 - trying to remake the old AI,* www.tt-forums.net/viewtopic.php?t=44809 (2015).

[32] Michiel, *ChooChoo, a train network AI,* www.tt-forums.net/viewtopic.php?t=44225 (2013).

[33] M. Claypool, K. Claypool, and F. Damaa, *The effects of frame rate and resolution on users playing first person shooter games,* in *Electronic Imaging 2006* (International Society for Optics and Photonics, 2006) pp. 607101–607101.

[34] Monsoon Power Inc., *Power Monitor,* www.msoon.com/LabEquipment/PowerMonitor/ (2016).

[35] M. Satyanarayanan, P. Bahl, R. Cáceres, and N. Davies, *The case for vm-based cloudlets in mobile computing,* IEEE Pervasive Computing **8**, 14 (2009).

[36] S. Choy, B. Wong, G. Simon, and C. Rosenberg, *The brewing storm in cloud gaming: A measurement study on cloud to end-user latency,* in *11th Annual Workshop on Network and Systems Support for Games, NetGames 2012, Venice, Italy, November 22-23, 2012* (2012) pp. 1–6.

[37] K. Chen, Y. Chang, P. Tseng, C. Huang, and C. Lei, *Measuring the latency of cloud gaming systems,* in *Proceedings of the 19th International Conference on Multimedia 2011, Scottsdale, AZ, USA, November 28 - December 1, 2011* (2011) pp. 1269–1272.

[38] N. Sheldon, E. Girard, S. Borg, M. Claypool, and E. Agu, *The effect of latency on user performance in Warcraft III,* in *Proceedings of the 2nd Workshop on Network and System Support for Games, NETGAMES 2003, Redwood City, California, USA, May 22-23, 2003* (2003) pp. 3–14.

[39] M. Claypool, D. LaPoint, and J. Winslow, *Network analysis of Counter-strike and StarCraft,* in *Proceedings of the 2003 IEEE International Conference on Performance, Computing and Communications* (2003) pp. 261–268.

[40] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani, *Energy consumption in mobile phones: a measurement study and implications for network applications,* in *Proceedings of the 9th ACM SIGCOMM Internet Measurement Conference, IMC 2009, Chicago, Illinois, USA, November 4-6, 2009* (2009) pp. 280–293.

[41] G. P. Perrucci, F. H. P. Fitzek, and J. Widmer, *Survey on energy consumption entities on the smartphone platform,* in *Proceedings of the 73rd IEEE Vehicular Technology Conference, VTC Spring 2011, 15-18 May 2011, Budapest, Hungary* (2011) pp. 1–6.

[42] T. Vos, P. Tonella, I. Prasetya, P. Kruse, A. Bagnato, M. Harman, and O. Shehory, *FITTEST: A new continuous and automated testing process for future internet applications,* in *Conf. on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)* (IEEE, 2014).