A Peer-to-Peer mechanism to support advertisement auctions in smart cities

Alexander I. G. Stuart-Kregor

A dissertation submitted in partial fulfilment
of the requirements for the degree of
Master of Science
of the
University of Aberdeen.

Department of Computing Science

2015

Declaration

No portion of the work contained in this document has been submitted in support of an application for a degree or qualification of this or any other university or other institution of learning. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Signed:

Date: 2015

Abstract

The growing emphasis on smart cities and increasing availability of cheap technology is beginning to introduce technology to the city streets. Wireless-capable devices are being built into street furniture, allowing it to communicate with surrounding devices. These pieces of smart street furniture can be used to form a peer-to-peer network with other wireless devices. In order to create a peer-to-peer network however, users and devices are required to connect to one another.

Therefore, a platform is conceived that allows users to earn credits for participating in the network and sharing information around the it. The application will display adverts to the user, allowing the user to accrue credits they can redeem for rewards. This creates an incentive for the users to take part in the network.

In this project, the necessary framework will be developed to allow a simulation of this peer-to-peer platform. The platform will be run in the simulated environment and evaluated for suitability according to revenue and information propagation.

Acknowledgements

I would like to express my sincere thanks and gratitude to all those have helped with and contributed to this project. First of all, I would like to acknowledge the unfailing support, expertise and enthusiasm of my supervisor, Dr. Wamberto Vasconcelos. Dr. Vasconcelos has been instrumental in steering me through the process of creating the software, discussing pros and cons of various features as well as contributing a wealth of assistance with proof-reading and guidance while writing the documentation.

Secondly, I would like to thank my peer, and recent graduate Andrew Skinner, not only for his work which I am able to reuse, but also for completing the project to such a high standard. Andrew was also on-hand for any queries I had about the use of the simulator that was not covered in his documentation.

I would also like to acknowledge the efforts of my peer and flatmate Gregory Myers and personal friend Samuel Winter. Greg has frequently helped me to maintain motivation and occasionally steered my thinking in the right direction while I was confronted with a problem. Samuel was the unfortunate recipient of several proof-reading requests, and his accurate, well-targeted comments were very useful in highlighting shortcomings of the documentation.

Fourthly, I would like to thank the staff at Lifecycle Software Ltd., where I conducted my industrial placement. Without the experience I gained whilst in their employ, I doubt this project would have been as detailed and involved as it was.

Finally, I would like to acknowledge the efforts and experience of all the department staff, who have shaped my degree education while at the university. To them I am eternally grateful.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

As technology develops increasingly, we are growing more and more accustomed to being surrounded by electronics and gadgets. With the advent of popular wearable technology such as Fitbit[1] watches and wristbands, Jawbone[2] wristbands, and the Apple watch[3], technology is constantly around us.

Such is the prevalence of technology that even street furniture such as bins[4] are being improved or dual-purposed through the use of technology and creating smart street furniture. Thus towns and cities themselves are becoming enabled with technology through deploying smart street furniture.

This project explores the possibility of using these multi-purpose articles of street furniture and aims to create an information sharing platform, among them, making use of peer-to-peer technologies. Furthermore, an advertising framework is considered as a possible application of such an information sharing platform. This framework will allow the network and the feasibility of the application to be tested.

1.1 Motivation

Research has shown that it is possible for mobile devices to exchange messages even when they are moving in opposite directions towards each other [6]. Further research has demonstrated the viability of a peer-to-peer network formed by mobile devices within a city [19]. Thus there exists the possibility that a similar network could be created and utilised for various means. In order to further test this network and its the possible uses, this project aims to create an advertising marketplace within a smart city, making use of mobile devices and smart street furniture.

Using peer-to-peer technologies in this way allows information to be shared without the direct use of the Internet. In such situations where Internet connectivity is not available, these technologies are highly desirable and useful. For instance, even if conventional infrastructure fails, a peer-to-peer network created by local devices can restore connectivity between devices in the local area. This is highly advantageous in situations of emergency, when reliable infrastructure is most important. Having a local peer-to-peer network on standby could lessen the burden on mobile infrastructure during crises.

---

[1] `http://www.fitbit.com/uk`

[2] `https://jawbone.com/`

[3] `https://www.apple.com/uk/watch/`

[4] `http://www.bigbellysolar.co.uk/products/big-belly-telemetry-system`

1.2 Objectives

The project's objectives are as follows:

Create an advertising framework based on peer-to-peer networks

The main goal of the project is to create a mechanism for advertising in a smart city, making use of peer-to-peer technologies. This is, in addition, to test to see whether mobile peer-to-peer networks can be used to share information around, and support useful applications.

Advertising framework effectiveness

As a secondary goal, the effectiveness of the framework should be measured, to test whether it is worth researching further.

Standards and open software

To increase the system's compatibility with other tools and software, widely-accepted standards and open software should be used. Doing so will increase the utility of the software.

Modularity

In order to simplify any future development, the system should be created in a modular fashion, making it easy to customise different aspects of its behaviour. This allows the system to be easily extended for use in future testing.

Chapter 2

Background and Related Work

This chapter will cover background and work related to the topic of this project. The chapter covers general peer-to-peer mechanisms as well as auction types, especially those used in advertising. The chapter also covers important existing work before highlighting some relevant student honours projects.

2.1 Background

Beginning with smart cities, various peer-to-peer mechanisms and applications will be described, followed by advertising mechanisms, general purpose auctions and finally, online advertising auctions.

2.1.1 Smart cities

There are various working definitions of smart cities according to different organisations. A few working definitions have already been collected and summarised elsewhere [16, 7, 8]. IEEE[1] establishes that smart cities are comprised of various enhanced 'smart' components which are optimally designed to improve performance over their normal counterparts specifically designed to tackle the problems of growing and expanding cities, along with urbanisation. The combination of these smart components is the difference between typical cities and smart cities.

In 2008, at a conference, the CEO of IBM introduced IBM's Smarter Planet initiative[2]. Similarly, IBM is designing technologies and solutions to improve or smarten components of cities.

IBM is not the only organisation to be investigating smart cities and technology for use therein. In 2011, the European Union created the Smart Cities & Communities Industrial Initiative which then became the European Innovation Partnership for Smart Cities and Communities[3].

From the various sources mentioned introducing smart cities, there is a common underpinning: technology. Smart cities leverage technology to improve performance, this in turn means traditional components of cities and towns such as street furniture. Items like benches, lampposts[4], bus stops, bins[5] and so on, are modified to support newer infrastructure. Such devices are becoming equipped with wireless communication technologies like Bluetooth[6] and WiFi[7].

[1] `http://smartcities.ieee.org/about.html`
[2] `http://en.wikipedia.org/wiki/Smarter_Planet`
[3] `http://ec.europa.eu/eip/smartcities/timeline/index_en.htm`
[4] `http://hesalight.com/hesalink/`
[5] `http://www.bigbellysolar.co.uk/products/big-belly-telemetry-system`
[6] `http://www.bluetooth.com/`
[7] `http://www.wi-fi.org/`

Enabling such devices to communicate provides access to information the device holds. This could be environmental information such as temperature or humidity, but could also be information internal to the state of the furniture. BigBelly[8] Solar Compactor bins are a good example of smarter street furniture, as they are equipped with WiFi and enable control through a mobile app.

Given that these devices are able to communicate wirelessly and have the potential to store data, they can be used as part of a network to exchange information.

2.1.2 Peer-to-peer mechanisms

Peer-to-peer networking, loosely defined is a means of computer and device communication that offers an alternative to traditional architectures, such as the client-server model, attempting to solve some of the problems associated with them, particularly scalability.

In peer-to-peer networks, devices send messages to one another according to a predefined protocol which details the messages in the conversation. By following a protocol, devices can communicate with each other and create a conversation of sorts. There are variations among protocols, however there are common elements which transcend a lot of popular protocols. The first of these is that messages sent between peers are typically given a 'type'. By doing this, the recipient of the message knows how the message should be handled in regards to the protocol. The messages will also have numerous fields containing metadata such as the address of the sender. Another critical field is a Time To Live (TTL), which dictates how many peers a message can be sent to before it expires. Having such a field ensures that the networks are not flooded or over-burdened with old messages that are still in circulation.

There are various routing protocols and topologies within peer-to-peer networks, each with their own merits and drawbacks. Napster could be considered a significant origin of peer-to-peer systems that made use of centralised components as it makes use of a brokered architecture [20, 21]. An early service designed for music sharing, Napster is no longer running since it enabled its users to infringe upon copyright held on the files being shared. Napster is an example of a peer-to-peer topology . Users would first connect to the Napster service, which held a directory of which peers were sharing which files. Upon requesting a file, the service would then supply details to the software of the peer where the file could be found, and the file would download directly from the other peer.

The next prominent architecture is Gnutella[9], which utilises a fully decentralised model, with the option of centralising some components. Although it is possible for peer-to-peer networks to form and run without any centralisation as in Gnutella, creating a hybrid topology has its uses. Many topologies including instances of Napster and Gnutella would be considered hybrids.

Some hybrid architectures follow a centralised-decentralised model, meaning that overall, the network is mostly decentralised, with some locally centralised components. These locally centralised peers are known as 'supernodes' or 'superpeers'. A supernode will connect to multiple other normal peers, but will aggregate requests from them and communicate with other neighbouring supernodes as necessary. Such an approach will reduce the bandwidth necessary when performing searches since the majority of messages are passed between supernodes. This topology is demonstrated in Figure 2.1. The supernodes (labelled) connect to each other enabling the

[8] `http://www.bigbellysolar.co.uk/`
[9] `http://rfc-gnutella.sourceforge.net/`

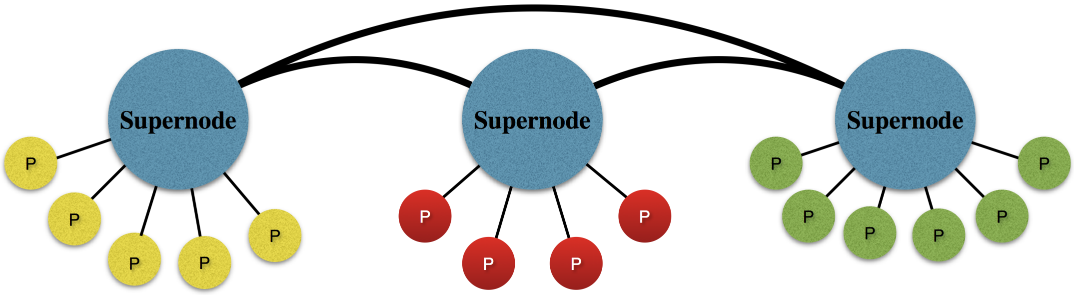

Figure 2.1: Centralised-decentralised topology

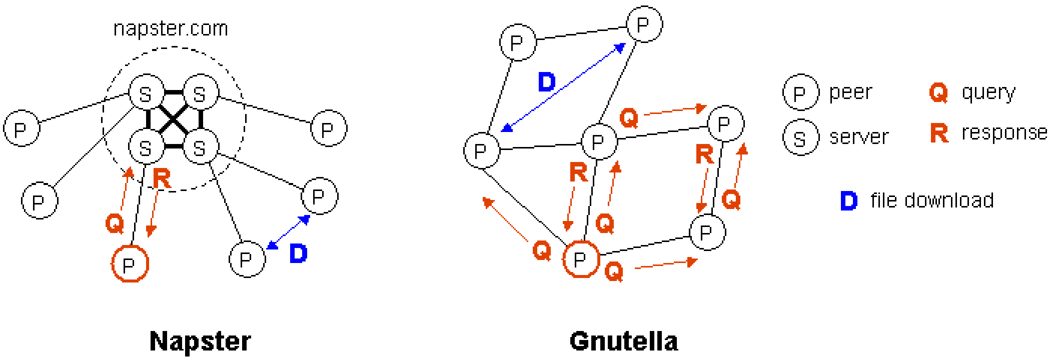

Figure 2.2: Differences between Napster and Gnutella.
Source: Microsoft research[10]

other peers (labelled 'P') to communicate by proxy.

Figure 2.2 illustrates the differences between the protocols used by Napster and the Gnutella protocol. The image clearly shows the common element between the Napster and Gnutella protocols, which is that files are downloaded directly from other peers, marked by the blue 'D'. The differences in the architecture are also shown, and in the Napster protocol, the peer can be seen querying the server ('S'), rather than its neighbouring peers ('P') as in the Gnutella protocol. An important and influential peer-to-peer protocol is Document Routing, which is used by Freenet[11]. Freenet's protocol is different to that of Gnutella or Napster since it considers file routing and distribution, along with the location of the file itself. Similarly, searches are intelligently routed in conjunction with files. The basic premise is that peers and files are identifiable with keys, and when a file is pushed into the network, a key for it is created. The file is then routed through the network of peers until it reaches the peer with the most similar key to the file. A peer determines the best match by consulting a 'routing table', which contains keys for neighbouring peers. This is one of the noteworthy differences between Freenet and Gnutella; message paths vary depending upon a distance metric.

The basic protocol used by Freenet is depicted in Figure 2.3. A search is initiated by peer *A* and travels through the network from neighbour to neighbour until a match is found. Peers check their neighbours sequentially, as can be seen between peers *B*, *E* and *F* with messages 4 - 8. The request is forwarded until a reply is received, which can be either success or failure. If a failure is

[10] `http://research.microsoft.com/en-us/um/people/ssaroiu/publications/mmcn/2002/mmcn.html`
[11] `http://www.freenetproject.org`

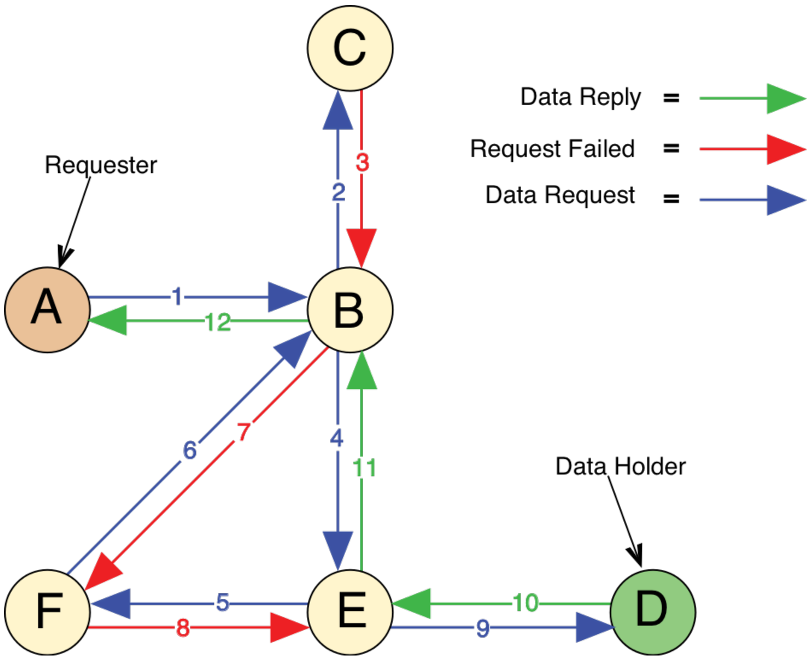

Figure 2.3: Freenet protocol

received, the peer tries the 'next' neighbour.

2.1.3 Applications of peer-to-peer networks

Peer-to-peer networks are typically used to enable users to share files over the network, the most famous example being a protocol and corresponding application developed by BitTorrent[12]. Many other desktop applications have been written to use the BitTorrent protocol. However, peer-to-peer protocols are used for other applications as well.

Skype[13] originally used a peer-to-peer protocol to enable communication between its users [5]. Although reportedly[14], the protocol has changed significantly since it was created. The peer-to-peer protocol was used for user search predominantly, with calls using direct connections between peers.

Aside from sharing files, peer-to-peer technologies have also been used for content delivery. Until a few years ago [18], BBC's iPlayer[15] desktop application used a peer-to-peer network to lessen the load on their own infrastructure and reduce costs spent on bandwidth. The content delivery used by iPlayer wasn't entirely peer-to-peer, and their own infrastructure did also serve files in part, in order to distribute pieces of them across users' computers.

Similar to the BBC's iPlayer platform, Spotify[16] also used a peer-to-peer mechanism to distribute content [9]. Since Spotify has expanded, the peer-to-peer mechanism has been removed.

Both BBC iPlayer and Spotify stopped using peer-to-peer once they were able to support and provide their own infrastructure.

[12] `http://www.bittorrent.com/`
[13] `http://www.skype.com`
[14] ArsTechnica article
[15] `http://www.bbc.co.uk/iplayer`
[16] `http://www.spotify.com`

2.1.4 Advertising mechanisms

It is important to gain some understanding of conventional advertising as it occurs in the modern world. Companies such as ClearChannel[17] and JC Decaux[18] specialise in delivering predominantly outdoor advertising space to companies via billboards, bus stops, digital displays and more. This shall be regarded as 'conventional advertising' from the perspective of this project.

It is interesting to note how this advertising space is sold and procured. For smaller businesses, it is possible to directly work with advertising companies to create campaigns [19].

From searching various resources, it has not been possible to determine exactly how these traditional forms of advertising are sold with the exception of smaller businesses dealing directly. It seems like larger companies would approach an advertising agency to develop their campaign, who would then work directly with companies like those previously mentioned to publish the campaign and make use of the advertising space.

The reason this is interesting to note is that the supposed mechanism of interaction is directly business to business, with an assumed large amount of time and effort being spent to construct a campaign with an agency. This differs greatly to the next mechanism to be discussed, sponsored search advertising and AdSense.

2.1.5 Online advertising

Advertising on the internet, specifically sponsored search advertising and Google AdSense[20] differ quite strongly from more traditional ways as discussed in Section 2.1.4.

The origins of sponsored keyword advertising are not exactly clear, but it is possible that it dates back from as early as 1996 [12].

The idea behind sponsored search, is that relevant relevant adverts will be placed on the page, near the search results. These adverts will contain one or more of the search keywords supplied to the search engine in order to maintain relevance. Google AdSense works similarly, in that it produces relevant advertisements according to the content on the participating site.

It is interesting to mark the differences between conventional e-commerce, more traditional advertising, and online search advertising. Typically, when a sale is completed or a bid is placed, it is for something permanent such as a company service or product. Unlike the more traditional advertising mechanisms mentioned in section 2.1.4, search advertising differs in that the advertising slots are created as they are required, per search [11]. As such, there is no concrete concept of a limited amount of advertising space. This difference means that alternative mechanisms were required, such as those discussed in Section 2.2.3.

In most cases, systems like Google's AdSense and various sponsored search mechanisms will encourage advertisers to sign-up to their platform and allocate budget information as well as providing keywords where necessary. Automatic auctions will then be held to determine which adverts should appear and where. These will be discussed in the following section.

[17] `http://www.clearchannel.co.uk/`
[18] `http://www.jcdecaux.co.uk/`
[19] `http://www.clearchannel.co.uk/direct-sales/`, `http://www.jcdecaux.co.uk/contact`
[20] `http://www.google.com/adsense/start/`

2.2 Related Work

This section, will cover important related topics including mobile ad-hoc networks, as well as auctions used in agent-based systems and auctions for advertising. Finally, previous undergraduate projects will be discussed, as well as simulators and agent development frameworks. These topics provide necessary context to decisions made during later stages of the project, with specific regard to auctions and simulators.

2.2.1 MANETs

Besides conventional peer-to-peer protocols, another related research area addresses ad-hoc peer-to-peer networks. Of particular interest within this area are Mobile Ad-hoc NETworks (MANETs), which bear similarities to the system under development. A key difference between traditional peer-to-peer networks, is that the network to connect the peers together already exists, and the protocol does not consider exactly how the peers are connected. However, with MANETs, the underlying network is created and adapts on demand [13]. Moreover, unlike classical peer-to-peer networks, where peers can connect directly to one another, in MANETs, if a peer is not in range, it must be reached through neighbouring devices [13].

2.2.2 Agents and auctions

There are various types of auctions used in the real world which are also used in agent-based systems. This includes open cry auctions such as the English and Dutch auctions, along with sealed bid auctions such as First Price sealed bid and Vickrey auctions.

The first important distinction between these four auction types is the environment in which the bidding takes place. Open cry auctions are arranged such that each bidder knows how many bids have been placed, and the current value of the lot, since the bids are typically called out by an auctioneer [14]. Sealed bid auctions however are conducted in way such that the value of a bid is only known by the original bidder and the auctioneer, instead of all bidders in the auction.

Another important distinction between these auctions is how the value of the lot is decided. In the English auction, the auctioneer will open the bidding at a starting or reserve price [14]. Bidders will then *cry* out increasingly higher bids to the auctioneer until a winning bid is reached.

In the First Price sealed bid and Vickrey auctions, the auction can be opened in a similar way (with a starting price) and the bids are collected in sealed envelops or similar. Unlike most English auctions, these sealed bid auctions place a time limit on bidding. Once this time is over, the bids are collected and the winning bid is chosen by the auctioneer.

The Dutch auction is somewhat different, in that although the auction is opened with a starting price, the starting price is very high [14]. The auctioneer then gradually lowers the price of the lot until one or more bidders place a bid. If more than one person bids for the same price, the auctioneer will then start increasing the price, using smaller increments until a winning bid is found.

The Vickrey auction is the most different to the other auction types mentioned, in that it is a second price auction [14]. This is the key difference between the Vickrey auction and the First Price sealed bid auction. The Vickrey auction proceeds just as a First Price sealed bid auction would, with the exception that when the bidding has finished, the auctioneer picks the winning bid but only charges the winner the value of the second highest bid, not what the winner actually bid.

This has the side-effect that bidders are encouraged to bid what they believe is the true value of the item and is the only auction of the four discussed that is not susceptible to the “Winner’s curse”.

The Winner’s curse describes the effects of a winning bid in most first price auctions. During an auction, bidders usually have their own estimate of the value of the lot for auction, which will most likely differ from the estimates of other bidders. Bids can then be placed based on this estimated value, causing some bidders to bid potentially more than the actual value of the lot, aside from also bidding more than anyone else is willing to pay [15]. This is the Winner’s curse, since the winning bidder might lose out if they paid more than necessary, or what the item is worth.

2.2.3 Advertising auctions

The auctions used by various search engines such as Google and Yahoo to power their advertising platforms have been studied relatively well [1, 17]. Such auctions and their implications are key, since search engine companies such as Google make vast amounts of their income from advertising. For instance, in 2014, Google made over $59 billion in advertising revenue[21]. Therefore, it makes sense to carefully study the processes involved given the amount of money available.

The Generalised Second Price (GSP) auction is commonly used for selecting which adverts should appear alongside search results [11]. The GSP auction is very similar to the Vickrey second price auction [22], however modifications are made to tailor the auction to sponsored searches. One of the reasons for this is the potential for multiple winning positions, making it difficult to calculate the harm done to other bidders through the auction, therefore the auction is generalised [11]. This means that the harm caused between bidders is not considered.

Another reason is that the advert selection process can be quite complex. For instance, there are three interested parties involved in sponsored searches: the advertiser, the company providing the advertising platform, and the user [12].

Furthermore, unlike the Vickrey auction, bidding honestly is not always a dominant strategy in GSP auctions [11]. It is worth noting that in the case of Google, the plain GSP auction is modified to consider other factors such as Click-Through Rates (CTR) [11]. These CTRs given an indication of the advert’s quality and relevance to the search terms used by the user, and provides a weight in the auction [12]. Such factors help to stabilise the auction and prevent bidders constantly changing their bids to gain a better position over the others. Along with the CTR, the position of the advert on the page is also taken into consideration when calculating such weights [2].

Other mechanisms and auction types have been discussed in literature and the general process is well documented [3].

The GSP auction itself has been explored in many ways, including various assumptions about bidder independence, bidder budgets [4] and also CTRs along with bidder and slot matching [2, 4].

2.2.4 Student Projects

Some work covering mobile peer-to-peer networks has already been done by fellow students and graduates of the university. The first is a project which explored the possibility of using Bluetooth to form a peer-to-peer network using mobile devices [6]. This project provided an indication that such a network would be feasible.

[21] `http://investor.google.com/financial/tables.html`

The next project is a simulator, Urbansim, created to model peer-to-peer networks in cities [19]. Further discussion about Urbansim is continued in section 2.3.1.

2.3 Simulators and agent frameworks

When beginning the project, it was hoped that it would be possible to work with the local city council (Aberdeen City Council) and the BigBelly company developing a similar solution to run on their solar compactor product. Unfortunately, such a collaboration didn't come to fruition in the timeline expected. Therefore, in order for the project to progress, it was decided that a simulator would be required. Using a simulator has the added benefit of abstracting away from underlying hardware and any associated issues.

Some exploration was conducted as to existing simulators and also frameworks that might be useful for creating one.

2.3.1 Urbansim

The first candidate considered for the project is a simulator, Urbansim, developed as a final year project [19] last year. The simulator makes use of open standards and frameworks, including a very scalable agent simulation library, MASON[22]. Since MASON is written in Java, Urbansim builds on it and creates a simulator that is relatively easy to extend. Possible extensions to Urbansim are discussed in section .

The most attractive feature of Urbansim however is its inclusion of SUMO[23] integration. SUMO is a traffic simulator and is used to model the movements of agent devices in a physical road network within Urbansim. As such, it creates the possibilities of modelling location-sensitive adverts in a physical peer-to-peer network.

The simulator uses XML[24] configuration files to not only control aspects of the traffic simulation, but also configuration of the agents. Agents can be supplied with XML data of an arbitrary structure, making customisation very easy.

Coupled with the advantages given by the architecture and the technologies, since the developer of Urbansim is a graduate of the university, asking questions about developing with the simulator was a reliable and helpful communication channel.

2.3.2 Agent frameworks and alternatives

The possibility of creating a specialised simulator for use in the project was considered and various frameworks and libraries were explored. During the creation of Urbansim, Skinner also evaluated some existing libraries.

JADE[25] is a framework used for the development and testing of software agents. Although it is a popular tool, it has numerous drawbacks, the first of which being that the developer is forced to follow JADE's architecture of using behaviours, which is not always desirable. Secondly, as Skinner reports [19]:

> *'From previous experience with this library it was clumsy to develop for...'*

[22] `https://cs.gmu.edu/~eclab/projects/mason/`

[23] `http://www.dlr.de/ts/en/desktopdefault.aspx/tabid-9883/16931_read-41000/`

[24] `http://www.w3.org/XML/`

[25] `http://jade.tilab.com`

It is easy to see what Skinner is referring to, as development with JADE can be unnecessarily time-consuming and frustrating. Skinner also remarks on the performance of JADE:

> *'Whilst using it in the past the performance, was poor in comparison to alternative solutions, I expected much more from such a popular library.'*

Skinner goes on to discuss another framework, Peersim[26] which he considered to be unsuitable for his project [19].

Having read Skinner's discoveries in regards to agent frameworks and libraries, the logical choice would be to use Urbansim for this project.

Summary

In this chapter, a background to the project has been given in section 2.1 covering smart cities, general aspects of peer-to-peer networks as well as their applications. Advertising in the traditional sense has been introduced along with online advertising in particular. In section 2.2, topics more closely related to the project have been covered to provide important background information and to support design and implementation decisions discussed in chapters 3 and 5.

[26] `www.peersim.sourceforge.net`

Chapter 3

Requirements and Architecture

The functional and non-function requirements proposed for this project are defined in the following pages, along with the corresponding architecture.

3.1 Functional requirements

FR1 Modelling and Simulation of individual devices (with disparate capabilities) as they communicate and share information using peer-to-peer mechanisms in a smart city

In order to examine the behaviour of devices within a smart city, it is not just necessary to create a model to represent them, but also the modelled devices must be simulated within the context of a city in order to observe their behaviour. Although the potential for a collaboration involving hardware and street furniture was present during the earlier stages of the project, this was ultimately not possible, as mentioned in section 2.3. It must be possible to alter behaviour of a device depending on its capabilities.

To enable communication between the devices, a peer-to-peer protocol will be required so that the devices can exchange messages and information.

FR2 Model behaviour of an advertisement framework running on a peer-to-peer network

Provided a peer-to-peer network is formed between the simulated devices as per FR1, a model of an advertising framework must be created to use the network. The framework must make use of appropriate mechanisms to allow adverts to be displayed. There should be the possibility to decide which adverts to display.

FR3 Collect data from interactions between devices

Messages of significance exchanged between devices should be collected for later analysis. Enough useful data should be captured such that the behaviour of models can be inspected and tested.

FR4 Implement and model at least one auction mechanism

As stated in FR2, there is a requirement for an advert selection mechanism. Previously in section 2.2.3, several auction types for selecting adverts were discussed. Therefore, a suitable auction mechanism must be available to provide a mechanism of selecting adverts.

3.2 Non-Functional requirements

NFR1 Use widely-understood standards where appropriate to ensure cross-compatibility

It is possible that the project might be integrated with other software and tools, or be further developed. In order to make future development and maintenance simple, well-known and widely-adopted standards should be used. Using this approach lessens the learning-curve and initial effort required to make changes to the project.

NFR2 Use open source and 'free' software

Any extra components or libraries that are not developed from scratch should be open source and free, in the sense that they are openly modifiable. Using open source software allows support to be sought from the community surrounding the software and also keeps costs at a minimum.

NFR3 Develop features in a modular fashion

Ensure that modularity is a core attribute of the project design to enable easier future development, as well as easier debugging. In addition, variables that form integral parts in business logic should be configurable either within the code itself, or from a configuration file. Where possible, specific implementations should be abstracted away from the behaviour of the project, such that different implementations can be used.

3.3 Architecture

This section details the architecture of the peer-to-peer advertising framework described by the requirements in section 3.1.

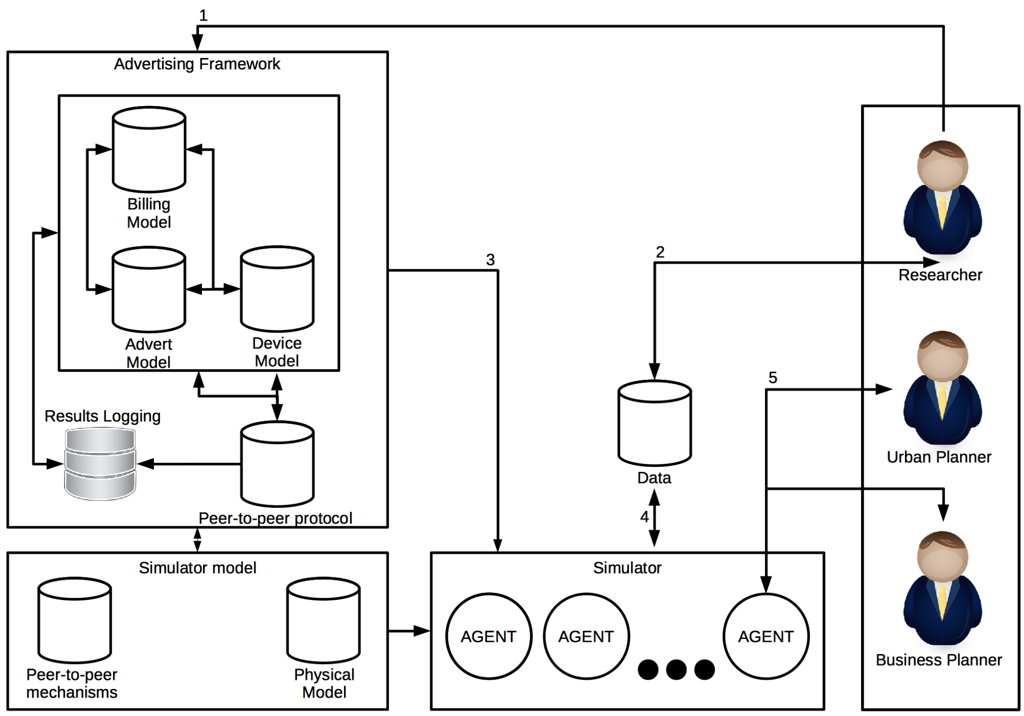

Figure 3.1: Architecture

3.3.1 Simulator

The architecture depicted in Figure 3.1 shows the underlying architecture of the Urbansim simulator itself, as well as the advertising framework being developed. The overall architecture of Urbansim remains unchanged from [19]. **This component satisfies FR1**

3.3.2 Advertising Framework

The advertising framework as shown in Figure 3.1 is the main focus of this project. It consists of various core components and models such as the billing, advert and device models; as well as the peer-to-peer mechanisms and results logging components. The overall system behaviour is defined by these models and their configuration. **This component satisfies FR2**

3.3.3 Device Model

This component models physical devices within the peer-to-peer network, including their features and behaviours that vary depending on the available functionalities. The device model is a key component of the framework which executes most of the business logic for the entire system, including the peer-to-peer protocol.

This component satisfies FR1 and FR2

3.3.4 Advert Model

The advert model represents an advert within the framework, along with any necessary attributes such as a bid, budget or payload. This model is quite simplistic in nature since it is not required to implement many custom functionalities.

This component satisfies FR2

3.3.5 Billing Model

The billing model defines the behaviour of the economic ecosystem within the framework. This component dictates how credit flows through the network.

This component satisfies FR4

3.3.6 Peer-to-peer protocol

The peer-to-peer protocol is executed by the device model, enabling devices to communicate with each other and exchange data in a structured 'conversation'. It also relies on other models to define what specific functionalities are required within the protocol. Therefore, the protocol is strongly linked to the billing and device models.

This component satisfies FR1 and FR2

3.3.7 Results logging

The results logging component provides the capability of recording the states of simulation, as well as logging instances of other models, such as the advert model. This component is critical in enabling the collection of results for processing.

This component satisfies FR3

3.3.8 Inter-component connections

Various components within the architecture need to communicate with each other in order to create the overall system.

The most important connection is the connection between the advertising framework and both the simulator and simulator model. These connections are formed so that the device model developed can be run in software agents by the simulator. Connecting with the simulator model also gives the device model the possibility of being mobile and simulated in an urban environment.

The link between the peer-to-peer protocol, and the models within the advertising framework are also critical. The protocol must be aware of all models which are present in the peer-to-peer network. As mentioned in 3.3.3, the device model executes the peer-to-peer protocol and therefore the two components are well integrated.

The billing, advert and device models are also inter-connected. The billing model is used by both the advert and device models to control the economic aspects of the framework. The advert model is used primarily by the device model since adverts and devices are involved in the peer-to-peer protocol.

The results logging component is connected to all components of the advertising framework to provide the necessary functionalities to log useful data.

Users can alter the parameters of the framework from within the models of the simulator, allowing easy configuration of the simulation.

Chapter 4

Design

This chapter discusses various elements concerned with the design of the system. This begins with the entire ecosystem being modelled, as well as the components therein and how they are represented to lend important context to the peer-to-peer protocol being developed. The design of the peer-to-peer messages and protocol, the main component of the system, is later described within the context of the devices. Justification and explanation of various design choices are given as well. The design of the system developed in conjunction with its implementation and many of the outlined features and components were created iteratively, gradually expanding on what behaviour was required or seemed sensible.

4.1 Ecosystem

This project considers the design of an ecosystem in which advertisers would wish to target the public within a smart city. The advertising mechanism would rely on a peer-to-peer protocol to distribute adverts and other information around a network of similar devices. The following section defines various global behaviour parameters upon which other behaviours are based. Predominantly, adverts, devices and the peer-to-peer mechanism are the fundamental components that form the ecosystem.

4.1.1 Devices

The devices are the basic component of this network, since they follow the peer-to-peer protocol and enable communication between themselves. Regardless of the type of device being used, the same protocol is followed to enable the devices to form a network. In some cases, it is necessary to alter the behaviour slightly depending on the features of a device.

Therefore it was decided that a distinction should be made between devices that can and cannot display adverts. This is done to enable different behaviour between the device types in regards to how adverts are handled, specifically surrounding selection and billing mechanisms. Classifying devices in this manner was considered superior to categorising devices by whether they were mobile or not, since a fixed device can still show adverts. The representation of adverts in this ecosystem is discussed in section 4.3.

Devices that can display adverts have a fixed number of ‘display slots’. This is the number of adverts that the device can display either concurrently, or in constant rotation after selection. For the ease of testing and modelling, all devices are assumed to have the same number of display slots. To take advantage of different types of devices, a more accurate model could be developed to enable devices to show varying numbers of adverts. For instance, a bus fitted with such a device might only have one or two display slots but can also only display one advert at any given time. However mobile phones and smartphones might have various areas where adverts can be displayed simultaneously, therefore enabling more display slots than some other devices.

Display devices operate on a ‘pay-per-impression’ billing basis. In turn, this means that an advertiser would be billed when their advert is shown, in line with a traditional advertising approach, rather than whether or not the user engaged with the advert. This is contrary to sponsored search advertising, which is based on pay-per-click and similar mechanisms. The reason for this is that some devices that are capable of displaying adverts may not have any interface for user interaction. For example devices attached to public transport vehicles such as buses and taxis, or displays at bus stops.

The possibility of charging adverts a ‘transport fee’ for passing through a device was considered and deemed unnecessary. Having such a charge would of course be possible, however it could be seen as off-putting to prospective advertisers. This is especially true when one considers the fact that an advert might never be displayed at all and may just propagate through the network. Further details surrounding charging and advert selection mechanisms, advert budgets and billing mechanisms are discussed in section 4.2.

For ease of modelling, it is assumed that all devices will follow the same charging mechanism.

4.1.2 Currency and credits

A system of currency is required for an economic environment to be created in the described ecosystem. It is proposed that a system of ‘credits’ will be used to emulate an arbitrary currency within the network. With this rudimentary credit, adverts can spend credits and devices can accrue them from adverts that are successful in being selected for display.

In order to make this credit useful, it must be supplied from a source and also be in demand from collectors. Various different mechanisms were explored in terms of practical functionalities as well as implications on the behaviour of the ecosystem. These are further discussed in section 4.2.

As a precursor to discussing the different possible mechanisms, it was decided that the owner of the device would ultimately receive the credits accumulated on that device from adverts. This is a somewhat simplified approach and, in practise it is conceivable that various parties would demand a certain percentage of credits accrued by devices.

4.2 Advert budgets, selection and billing

Given a system of credits, there should be a mechanism to enable the spending and billing of these credits as adverts move through the network.

4.2.1 Advert budgets

One explored early on stipulated that a certain amount of credit could be paid by the advertiser upon introducing an advert into the network. This could then form a budget for the advert to spend as it travels through the network, in turn allowing devices to collect the resultant credit. For advertisers in general and especially those trying to reduce costs, this is an attractive feature as it intends to guarantee that a campaign will cost a fixed amount despite being varied in terms of how it is conducted. In the event that an advert did not spend all of its credit, the credit could be refunded or the advert could be re-introduced to the network until it was depleted.

A noteworthy co-requisite of advert budgets is that there should only be one copy of the advert in the network at any one time. The reason for this requirement is that if the advert were able to be copied, the multiple copies of the advert could collectively spend more than the budget allocated to the initial advert. A workaround for this was to consider splitting the budget every time an advert is sent from one device to another. However this would also require the initial budget to be sufficiently large such that the advert can propagate effectively throughout the network and still have budget to spend in a meaningful number of selection rounds. Therefore, in the interest of advert longevity, this option was excluded from the model.

4.2.2 Advert selection mechanisms

In order to make decisions about which adverts should be shared or should be displayed, one or more selection mechanisms are required.

To take advantage of the peer-to-peer mechanism that is being developed, a selection mechanism can be created. This mechanism would assign adverts to neighbouring devices prior to sending, in accordance with various rules and conditions to ensure that devices receive adverts that are most appropriate to them.

Overall, there are many possibilities on how to determine where adverts should be sent or which adverts should be displayed. There are basic fields which are considered such as the release date, expiry date and geofence. These fields are further discussed in section 4.3. Given more time, it would be worthwhile to explore the possibility of creating a taxonomy for adverts and devices such that, given an ontology, devices can reason about which adverts are suitable for which destination. Unfortunately although such a sophisticated mechanism would be useful, it is not essentual and would distract from the core functionalities being implemented. In the wider sense, such mechanisms also indirectly influence which adverts will be displayed on a device.

Along with the mechanism to share adverts, a method of selecting adverts for display is also required. A very basic selection method would be to pick adverts at random and charge a uniform fee for their display. However, this creates a problem of a lack of competition between adverts and in turn, this compromises the potential for an economy within the ecosystem.

Therefore, it was decided that an auction should be used to determine which adverts should be displayed on a device. Adverts will been created with a pre-defined bid, which is considered during the auction, discussed further in section 4.3.1. Exactly which auction type to use also requires careful consideration.

4.2.3 Auction mechanism

As reported in section 2.2.3, research was conducted into auction mechanisms used for selecting adverts to appear alongside sponsored search results, as well as auctions used in agent computing.

The Generalised Second Price (GSP) auction, as mentioned is the most commonly used auction mechanism for online advertising. Since in this implementation adverts cannot dynamically bid or interact with the auction in any way, the choice of mechanism seems somewhat redundant. However, it is still important to consider the impact on advertisers using the system since the type of auction used is one of the main influences on which adverts are displayed.

Other auction types such as the English, Dutch, one-shot sealed-bid and Vickrey auctions were also considered. As mentioned, adverts do not form an active part of the auction, therefore open-cry auctions such as the English and Dutch are less suitable than if the devices ran software agents to interact with and place bids in the auction. If the required functionality were developed to enable adverts to bid dynamically, it would be sensible to more closely explore the effects of the various auction types, including open-cry auctions.

This leaves sealed-bid auctions such as GSP and Vickrey. Sealed-bid auctions are ideal for the current design since adverts only need to submit the bid they are created with, and the auction will handle the rest of the process. The reading conducted in section 2.2.3, suggests that the Vickrey-Clarke-Grooves auction is unsuitable, as it is difficult to calculate the impact a bid has on other bidders, hence the creation and adoption of GSP.

Thanks to its prominence within online advertising, a basic implementation of the GSP auction has been used in this project. When an auction is run, adverts are ranked in descending order according to their bid to fill the display slots on the device, as seen on line 6. Each advert is charged the amount of the next highest bid that is less than the bid of the advert itself. In the case that there are fewer adverts than display slots, the last advert is charged a reserve price and the auction finishes with some display slots unallocated.

Consider a device, D with three display slots. Device D holds adverts A and B with bids 4 and 2 respectively. Advert A would be charged 2, which is the bid of B, while B would be charged the reserve price since there are three display slots but only two available adverts.

Algorithm 1: Adverts GSP auction

```
   Input: adverts[], numberOfSlots, reservePrice
   Output: advertsToDisplay[]
1  begin
2      if | adverts[] | = 0 then
3          return
4      advertsToDisplay[] ⟵ ∅
5      index ⟵ 0
6      adverts[] ⟵ sortDescending(adverts[])
7      for index < numberOfSlots AND index < | adverts[] | do
8          currentAdvert ⟵ adverts[index]
9          nextAdvert ⟵ adverts[index+1]
10         amountToCharge ⟵ 0
11         if nextAdvert = null then                  /* No ads remaining */
12             amountToCharge ⟵ reservePrice
13         else
14             amountToCharge ⟵ getBid(nextAdvert)
15             index ⟵ index + 1
                   /* Deduct the amountToCharge from the advert's budget */
16         deductFromBudget(currentAdvert, amountToCharge)
17         advertsToDisplay[] ⟵ currentAdvert ∪ advertsToDisplay[]
18     return advertsToDisplay[]
```

Various different aspects of the auction mechanism were considered with respect to the behaviour of repeated auctions. A potential behaviour is that adverts could be held and prevented from participating in auctions for a given period if they were successful in the last auction. This could create greater amounts of diversity among the devices as to which adverts get shown, as well as preventing other adverts from being constantly out-bid. An alternative behaviour which could be used to ensure variety would be to send the advert to another device, then refuse the advert if it is received again.

4.2.4 Billing records

An alternative mechanism to advert bugdets was considered where each billable transaction, typically displaying an advert, would cause a record to be created. This record would uniquely identify the transaction that took place, including recording which advert and device it happened on, how much credit was spent as well as the physical location and time of the transaction. In order to accurately bill the advertiser, these records would have to be aggregated in a central, designated location so their uniqueness can be guaranteed and then advertisers could be billed appropriately. The credits obtained from the advertisers would then be paid to the device owners. This solution is considerably more complex than the aforementioned budgets since it requires a centralised component to receive the records regardless of how the records are transmitted. Another aspect is that this mechanism relies on billing in arrears, whereas providing adverts with a budget is billing in advance, with a pre-pay model.

Despite the fact that adverts are given budgets of credit to spend, billing records are still created and logged in order to document how much credit an advert spent.

4.2.5 Billing record propagation

In addition, two propagation mechanisms for these billing records were considered. One possibility is to assume that devices that can display adverts have the potential of being connected to the internet. Devices unable to display adverts could forward any billing records to capable devices using the peer-to-peer protocol so the records can be uploaded to a centralised server. Although this method would arguably be more reliable than using the peer-to-peer network, it does then mean that devices are reliant on the internet.

Alternatively, a device within the network could be designated as such a server and other devices would seek to push records as close as physically possible to it. The process would then continue across a chain of devices as records approach the physical location of the server. This approach is roughly based on the Document Routing protocol in that records are forwarded based on a distance metric. Using a more peer-to-peer oriented approach makes the system more resilient overall but includes the common risks associated with such networks. Most fundamental is the possibility for records to be lost if a device leaves the network without propagating them, and never returns.

Once the records are in the same location, they can be properly processed.

4.3 Advert representation

Adverts are modelled as single objects within the simulator. They contain various variables and pieces of information to characterise the advert itself, but also to enable logging.

4.3.1 Budgets and bids

In order for adverts to be able to spend the modelled currency, they must themselves have a representation of a credit. As discussed in section 4.2.1, an advert is given a budget to spend within the network. This is an attribute of the adverts themselves that is spent as and when they succeed in auctions.

Since auctions were decided to be the most viable selection mechanism to decide which adverts should be displayed on a device, each advert needs a way of bidding in the auction. In the interests of maintaining a simpler model, it was decided that adverts should be created with a single bid defined for them. This bid is the maximum amount of credit the advertiser is willing to spend per impression of the advert. There was the potential to have adverts bid dynamically and intelligently based on the properties and attributes of the current device, however this was deemed too complex given the scope of the project.

The two fields of budget and bid in combination enable the advert to have a concept of limited currency and expenditure as well as creating competition among advertisers.

4.3.2 Geofencing

A geofence defines a geographic area within a virtual border that is usually designed to relate virtual objects or events to a physical geographic area and restrict their behaviour depending on whether they are inside or outside of the defined border. In the case of this project, the geofence defines in which geographical area an advert may be displayed. To enable geofencing, adverts require two important and mandatory pieces of information; which are its point of origin in the network and a radius. In addition, another variable is included as an optional placeholder, which is the advert's current location. This is typically the current location of the device the advert is stored on. The point of origin will usually be assigned to the current location of device that imported the advert into the system from the beginning of the simulation, but it can also be supplied. The radius dictates how far in a straight line from the point of origin the advert is allowed to travel. This enables 'as the crow flies' distance checking to ensure that the advert is displayed within the specified area.

It was discussed with the project supervisor that adverts could also be forbidden from being shared across devices outside of their supplied geofence, however in the interest of device satisfaction and advert propagation, it was decided that adverts should be allowed to spread freely, regardless of any geographic restrictions.

The optional variable, the advert's current location, exists for logging purposes to allow the advert's path through the network to be determined.

4.3.3 Timing and expiry

To enable more fine-grained control for advertisers, two timing parameters are defined: a release date and an expiry date. These parameters control not only when an advert is allowed to begin propagating through the network, but also when it can take part in auctions. A beneficial side-effect of including these parameters is the ability to gently release the adverts into a simulated environment, rather than having every advert that will ever exist in the simulation available from the beginning.

Finally, a fourth time-related field is recorded, and that is the time the advert was received by a device. This is logged along with the current device ID as an alternative to logging the path of the advert. More details are discussed in section 4.4.1.

Message type	Expected payload
REQUESTDETAILS	null (no payload is expected)
DEVICEDETAILS	DeviceDetails Object
ADVERT	Advert Object
ADVERTACK	Advert Object from previous ADVERT message
BILLINGRECORD	BillingRecord Object
BILLINGACK	BillingRecord Object from previous BILLINGRECORD message

Table 4.1: Message types and payloads

4.3.4 Payload

The advert model includes a payload, which is the actual content to be displayed. In the scope of this project, the payload is just a placeholder and the file format does not currently matter. As it stands with the current project implementation, the payload is not normally populated. This is to ensure that the memory consumed by the simulation does not grow uncontrollably.

4.4 Peer-to-Peer mechanism

The peer-to-peer mechanism is the main component of the system being developed. It is used to share adverts and other supporting objects between devices. Since at least some of the devices running the protocol will be mobile, the protocol should be designed with speed and efficiency in mind. The possibility of two devices moving out of range of one another should be taken into consideration.

In order to also be useful, the mechanism should enable devices to find out information about their neighbours. For this part of the mechanism, a simple handshake will suffice to allow devices to establish a conversation.

There is scope for making the protocol and mechanism more sophisticated, especially in regard to making decisions about which neighbouring devices should receive which messages. The fundamental components to enable such logic are included in the design of the mechanism.

4.4.1 Message structure and payload

The messages used in the peer-to-peer mechanism are relatively simple. They consist primarily of two fields: a message type and a message payload.

The message type allows the protocol to determine how the message should be processed once it is received. The message types and applicable payloads are detailed in Table 4.1. The payload of a message is a string, which is usually populated with a serialised model class such as adverts, billing records and device details.

Table 4.1 lists each message type and its associated payload.

As marked in Table 4.1, the REQUESTDETAILS message is not accompanied by a payload. The acknowledgement messages, ADVERTACK and BILLINGACK, both require a payload of the object that they are acknowledging, otherwise it cannot be guaranteed which object is being acknowledged.

It was discussed whether any or all of the messages developed as part of the peer-to-peer mechanism should include a time-to-live (TTL). In order to ensure adverts are well propagated, a TTL was considered unsuitable, especially since the adverts have an expiration date.

Provisions for adverts and devices

In order for the adverts to move throughout the peer-to-peer network across devices, various fields are necessary within the representation of both adverts and devices to control their flow and enable their transmission.

The most important field included by both is a unique identifier for the objects themselves. Having such a field ensures that multiple copies of the same advert do not exist on the same device. For devices, this identifier field means neighbours and familiar peers could be recognised and the behaviour further modified to account for this possibility. The identifier fields are also critical for logging purposes, this is covered in more detail in section 5.6.1.

As part of an earlier design, an advert included a ‘path’ variable, which contained a list of device IDs corresponding to devices which the advert had passed through. This mechanism proved problematic in later testing as adverts are not heavily restricted in terms of where in the network they can travel. As a result, the path variable would grow quickly in size, causing the object serialisation to become slow and was therefore removed. In order to maintain the same calibre of object logging, two new fields were added: ‘device ID’ and ‘time received’. These fields used in combination allow the path of an advert to still be determined.

4.4.2 Peer-to-Peer protocol

The following description relates two devices, *A* and *B* with the defined protocol, showing how the protocol is followed. Although only two devices are being used as an example, multiple devices can follow the protocol simultaneously. This protocol is also given in AUML[1] in Figure 4.1.

1. Device *A* initiates a connection with all neighbours and sends a REQUESTDETAILS message.
2. Device *B* responds with a DEVICEDETAILS message, including a DeviceDetails object in the payload.
3. Device *A* can then send an ADVERT message, the payload contains a single advert object.
4. Device *B* receives the advert and responds with an ADVERTACK message, the payload is the same as the ADVERT message.
5. Device *A* can also send a BILLINGRECORD message, the payload contains a BillingRecord object.
6. Device *B* then replies with a BILLINGACK message, the payload contains the same BillingRecord object.

Steps 1 and 2 must be the first steps that are executed, sending the associated messages. These form the first stage of the protocol (details exchange), and without this interaction the rest of the protocol becomes invalid. Once this interaction has occurred, the device may execute a following stage comprising step 3 followed by step 4 (advert exchange) or step 5 followed by step 6 (billingrecord exchange).

[1] `http://www.auml.org/`

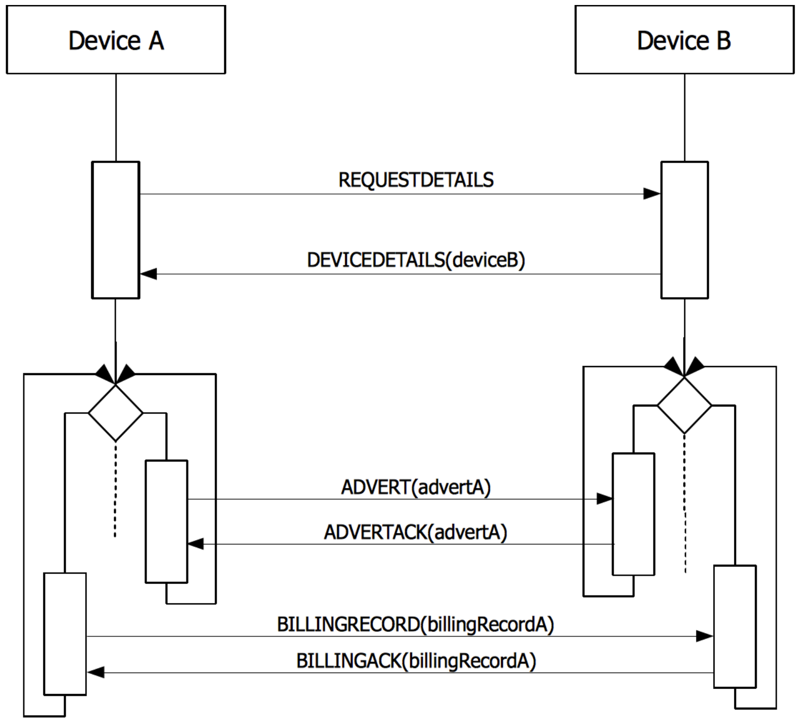

Figure 4.1: AUML protocol representation

As can be seen from the AUML diagram (Figure 4.1), the two possible stages after the first stage are not mutually exclusive, meaning that as long as the interactions within each stage are executed in sequence, the ordering between stages does not matter. It should also be noted that any stages which follow the first stage are optional.

Each message that initiates a conversation (REQUESTDETAILS, ADVERT and BILLINGRECORD) requires a reply either supplying the requested information or as an acknowledgement of an action.

The acknowledgement messages defined in the protocol are important for driving processes in the business logic. Having those messages allows devices to maintain only one copy of an advert or a billing record within the network. If an acknowledgement is not received, it is assumed that the message was not successfully sent and it will be recreated and sent to a different peer.

This method of error handling has been chosen as testing showed that if the protocol assumed the device had received the advert, the population of adverts decreased rapidly.

4.4.3 Storage limitations

In order to prevent over-growth, devices have a limit dictating how many adverts and billing records they can hold at one time. The reason for this is to not just ensure devices are not overburdened with objects, but that there is diversity among devices in terms of the objects they hold. This is a consideration for the fixed amount of storage and memory devices have, as well as the overall potential impact on the network. Having a fixed number of storage slots also requires some intelligence in sending devices as to how many objects should be sent to any given device.

Limiting the number of objects a device can store also alleviates some memory burdens from the simulator.

Summary

In this chapter, an ecosystem has been defined around devices and adverts. The key features of devices have been discussed along with their significance regarding a device's behaviour. A system of currency was proposed along with suggested billing schemes for adverts. Various advert selection mechanisms were explored before deciding on the GSP auction. Accounting for successful auctions and logging records were covered in addition.

The modelled representation of an advert was discussed along with required attribues. Finally, the peer-to-peer protocol was defined, along with the messages used and a consideration for device storage and resource limits.

Chapter 5

Implementation

This chapter will describe the processes, tools and approaches used during the development of the main software for the project. Other aspects such as development methodology and infrastructure such as versioning and backups are discussed along with tools. Next the protocol itself will be explored along with other fundamental components.

5.1 Development process

The design and development processes used for the project follow an exploratory programming approach. Since there is little existing literature on what has been created, there were many unknowns and unexplored areas. Other than exploratory programming, there were no particular methodologies or philosophies that were followed.

As new features were developed, a proof-of-concept prototype was put together using initial and rudimentary testing. A basic simulation was run to check for expected behaviour and any initial bugs, both syntactic and logical, printing information out to the console where required. Once the new code had been tested, it was refactored as necessary.

It was rare that significant amounts of refactoring were required, typically only when an entire feature or behaviour was incorrect. This is because the code was written with modularity in mind, ensuring operations were appropriately abstracted from each other. There are many benefits of this, foremost is the process for changing behaviours and implementations of individual functionalities is far simpler and should leave other features unaffected. Programming in this fashion also made fixing bugs relatively easy, since there was usually a maximum of one or two places where a change was needed.

Occasionally, modifications were required to be made to Urbansim itself, with the help of its developer. These are discussed in detail in section .

While development was occurring, functionalities were repeatedly tested for expected behaviour by running a test simulation. This is discussed further in section 6.1.1.

5.1.1 Backups

Regular and thorough backups were kept using third party services to ensure redundancy in case of a disaster and data loss. This was arranged in two ways: source control in combination with source repositories, and storage providers. The source control setup is covered in greater detail in section 5.2.2.

Dropbox[1] and Google Drive[2] were used for online storage to hold redundant copies of the entire project, both code and documents. After each new feature was tested or patched and retested, the changes were synced with the cloud services. This meant that the project was copied usually several times per day.

An interesting challenge was to ensure that the Git data was kept separate from the rest of the project and not backed up to the online storage services. To do this, an rsync[3] script was made to ensure the `.git` folder was not included when backing up the project to online storage. If the Git[4] data folder was included, synchronisation took a long time on account of all of the files that Git stores in its `.git` directory. Given the Git data was already being copied elsewhere, it was sensible to exclude it from the overall project.

To help with the backup process, two makefiles were written to make the process simpler. Although this may seem excessive, it meant that shorter commands were needed to backup the project. It gives the added benefit that the commands were recorded in case they needed altering.

5.2 Development tools

The following tools were used to aid development.

5.2.1 Language, IDE and build tools

Since the chosen simulator, Urbansim, uses the MASON Java library, the rest of the project will be using Java, thus avoiding cross-language integration issues. Urbansim was also developed with the Eclipse IDE[5], and has been packaged as an Eclipse project, so it was sensible to continue development in Eclipse.

Eclipse has the ability to run and generate Apache Ant[6] build files, one of which was supplied with the simulator. This made setting up the project relatively easy, however the project used to build to a folder full of classes rather than a JAR file [7]. Given the testing for this project will require automation, the method of running a simulation had to be simple. Therefore the Ant build file was modified to produce a JAR file of the simulator rather than a directory containing binaries. This also made specifying the classpath easier, which is essential to the simulator, since it is contained in the JAR's manifest file. Further modifications were made to the build file to handle dependencies better. The script now follows the Eclipse project layout convention and will automatically detect any required JAR dependencies and copy them to the relevant directory.

The reason for enabling this, is that the code being developed to run in the simulator is technically a library used by the simulator. Therefore, the resultant binaries from this project must also be on the classpath of the simulator and it made sense to dynamically include them. Since this project also uses external libraries, all of this project's library dependencies must also be copied over and placed on the classpath. The only downside to this was that each time a new library was added, both projects needed to be rebuilt.

[1] `http://www.dropbox.com`
[2] `http://drive.google.com`
[3] `https://rsync.samba.org`
[4] `http://git-scm.com/`
[5] `http://http://eclipse.org/`
[6] `http://ant.apache.org/`
[7] `https://docs.oracle.com/javase/tutorial/deployment/jar/basicsindex.html`

For the purposes of source control and separation of concerns, this project was developed as a separate IDE project, instead of adding code to the simulation project. This is feasible because the simulator looks for the required agent classes on the classpath using reflection in Java. So, the IDE project for this project's code produced a JAR file which was then copied into the dependency directory of the simulator.

Unfortunately it was not possible to package a JAR and have it contain the required libraries within the JAR file. Having researched methods how to achieve this, it seemed too complex and time-consuming to pursue, and the Ant build files were improved instead.

5.2.2 Source control

Along with the various approaches mentioned in 5.1.1, Git[8] was used for source control along with GitHub[9] and BitBucket[10], two repository hosting services. Git was used as opposed to other tools mostly out of familiarity and reputation. Similarly to the backups for general files, the two repositories were used together for redundancy.

As previously mentioned, every time a considerable change happened to the code, either new functionality or bug fixes, a commit was made. The commit was then immediately pushed to both GitHub and BitBucket as well as syncing the project folder with Dropbox and Google Drive. This ensured that recent backups were kept.

Occasionally source control was used to work out where bugs had been introduced, as well as rescue lost pieces of code, especially in the closing days of development.

Only one branch was used in source control, since all functionality was being developed from scratch, it was not worth the overhead of maintaining various branches.

5.3 Urbansim modifications

Although it was intended that Urbansim would be used without any further development being required, some modifications were made.

The first change introduced was to expose an existing method to get a device's current position. This method had to be declared in the device interface provided by Urbansim to agent classes in order to be used. A device's position was required to enable geofencing for adverts.

The second change made was to enable Urbansim to run without a GUI (Graphical User Interface). The GUI would not be needed during the automated testing, therefore the simulation had to be controlled entirely from the command-line.

As mentioned in section 5.2.1, the Ant build files for Urbansim were also modified to create a smoother building process. Although this seems minor, it was in fact a non-trivial task, requiring a constant reference to the Ant documentation. These changes to the build process were also instrumental in setting up Urbansim so that it could run in the automated evaluation framework.

Finally, the XML logging feature of the simulator was turned off along with any unnecessary console printing. This particular feature was not being used and was creating large amounts of data per simulation.

[8] `http://git-scm.com/`
[9] `http://www.github.com`
[10] `http://www.bitbucket.com`

5.4 Devices

Most of the logic for the network, including the peer-to-peer protocol is contained within the `Device` class. The possibility of modularising and separating the protocol from other device behaviour was considered, however the supposed trade-off between the added complexity and the estimated development time was not sufficiently significant. Therefore, in order to maintain simplicity, the peer-to-peer mechanisms have remained in the `Device` class.

5.4.1 Main loop

The main loop of the `Device` class has been created in such a way that the overall behaviour of the device can be easily determined by reading the method names. Algorithm 2 shows the procedure of the main loop, which is executed by the simulator. Most of the logic is extracted out to the responsible method in order to maintain simplicity.

Lines 2 to 4 ensure that any adverts supplied through configuration files are assigned a point of origin. This is a mandatory parameter, as mentioned in section 4.3.2. Since the file containing the adverts is only processed once, it is possible to use a boolean to keep track of whether adverts have an assigned point of origin. Lines 5 to 6 ensure that any expired adverts are no longer on the device, as well as clearing the list of neighbours in preparation for receiving a new set. Lines 7 to 13 run the developed peer-to-peer protocol. Each step is followed by a call to `listenForReplies`, since in order for the loop to progress, all messages must have been processed prior to the device initiating any new conversations. A potential improvement would be to allow these behaviours to run concurrently, however this could create problems if required information is not present. Once the protocol has finished running, the device disconnects from all its neighbours and then makes a call to the `sleep` function. This method call is a requirement of the simulator and also allows devices to conserve battery power.

Algorithm 2: main

```
   Input: advertsChecked, USE_SMART_MECHANISM
1  while deviceIsRunning do
2      if not advertsChecked && currentPosition ≠ null then
3          assignLocationToAdverts
4          advertsChecked ⟵ true
5      cleanOutAdverts
6      cleanOutNeighbourDetails
7      connectNeighbours
8      requestNeighbourDetails
9      listenForReplies
10     forwardAdverts
11     listenForReplies
12     forwardBillingRecords
13     listenForReplies
14     disconnectAll
15     sleep(100)
```

5.4.2 Simulator features

There are various features available in Urbansim which could have been used in the implementation of devices. These functionalities mostly relate to the state and environment of the device itself, attributes such as available battery charge or amount of storage space remaining.

These first two features were not used because the focus of the project was more concerned with creating the ecosystem and protocol required, and such features would have added more complexity to debugging processes. In addition, the device storage feature was not used primarily because all of the advert and billing record objects were kept in memory. Since advert objects were not carrying payloads of significant size, permanent storage was not required. If a realistic payload of an advert were to be implemented, this feature of the simulator would provide the required functionalities without requiring much time or effort to implement.

Another key feature of the simulator is realistic message sending times, an arguably more important feature than the storage space or device battery. This would allow the system to behave more closely to a practical, real-world implementation and simulate the time necessary to send messages between devices. Arguably, the system developed in this project is simplistic since advert objects have no payload, and messages do not take significant time to send.

5.5 Peer-to-Peer protocol

Although in section 4.4.2, the design of the entire peer-to-peer protocol is described, it was gradually and incrementally developed, instead of being created all at once. This incremental style was used as a result of devices needing certain functionalities. As a device's needs increased, the protocol was further developed to support them. Developing the protocol in this way meant that it was possible to consider the best method of implementation given how the protocol would be used by the devices. This resulted in a protocol that matches with the functionalities of a device quite closely, and does not include any extra or unnecessary aspects.

5.6 Object serialisation

In many peer-to-peer protocols and applications, serialisation and message parsing are very important to the performance of the system and the network. Some protocols, such as Gnutella use byte-sensitive message formats. In the interests of focussing on the message protocol itself, it was decided to use an existing textual data storage format, instead of developing a customised serialisation method. There are various existing open and common standards such as XML[11], JSON[12] and YAML[13], all of which have various implementations and libraries available for use with Java. Regardless of the specific format used however, it was considered important to ensure this critical component was implemented and integrated in a modular format. Java's interfaces were used to ensure that adding new parsers or changing the parsing method were easy. This will allow the parsing implementation to be changed according to any future requirements.

After considering how the serialisation would be used, it is important that the chosen format was simple and easy to process. XML processing in Java seemed to be very involved from a development perspective, so other formats were more closely investigated. Many XML libraries

[11] `http://www.w3.org/XML/`
[12] `http://www.json.org/`
[13] `http://yaml.org/`

also require the developer to examine the DOM tree[14], limiting their usefulness. Another disadvantage of XML is that it is quite a verbose data format, requiring a lot of surrounding structure to represent a given set of data.

JSON on the other hand, offers a much more compact, and neater way of capturing data. This is particularly important when considering that peer-to-peer systems are reliant on exchanging messages between devices and as such, bandwidth is precious and should not be wasted. Choosing a compact and space-efficient method of data representation is then key to the performance of a peer-to-peer network. It was decided that JSON was the most suitable existing format given its ease of use and compactness in comparison to XML. A bespoke and proprietary serialisation format was considered to further increase the efficiency of peer messaging, however the time that would have been required to create the serialising and parsing functionalities was too great. Using JSON has other benefits when processing the logged objects once the simulation has finished. Ruby[15] was chosen as the language to implement an automated evaluation framework because of its speed of development. The use of Ruby and JSON go hand-in-hand as Ruby provides native JSON parsing methods that turn JSON strings into key-value pair dictionaries. This again reduced the amount of time involved in the overall serialisation process.

Once this decision was made, various JSON parsing libraries were investigated, to allow the system to serialise objects into JSON. Oracle provides an API for processing JSON in Java[16] that has been implemented by various developers, however it is clunky and still requires the developer to manipulate the structure of the format and determine how the data should be parsed. Such an involved process was not suitable for this project, and other libraries were investigated instead. A suitable library, gson by Google[17], was found that allows the developer to call the serialisation and parsing methods supplying only an instance or similar example of a class, and the library will reflect on the object and automatically serialise it. Since the parsing is handled dynamically in this form, the parsing implementation required during this project was very minimal, and in most cases, the parsing methods are only one line long.

An added benefit to the gson library is that although an object in JSON is typically spread over multiple lines, the output from gson contains no new line characters, meaning that an entire object is contained solely on one line. This makes handling the JSON in external applications much easier.

5.6.1 Logging

Since this project is relying on a simulator, logging is a very important functionality required for collecting data before processing it into results. In order to capture experiment results and simulation details, a means of logging data was required.

Although Urbansim does some logging of its own, it is only able to log XML, and its logging is very low level, logging the individual messages exchanged between devices. Urbansim creates a maximum of one XML file per step of the simulation. Unfortunately this adds an overhead to parsing since there is structure repeated across the document and more file objects to process. It is possible for new components developed with Urbansim to make use of this pre-existing logging,

[14] `http://www.w3.org/DOM/`
[15] `http://www.ruby-lang.org`
[16] `http://www.oracle.com/technetwork/articles/java/json-1973242.html`
[17] `https://github.com/google/gson`

however a neater and more relevant solution was more desirable.

Initially it was conceived that a device would create its own files to log adverts and billing records to its own files. In early tests, this proved to be very resource intensive and caused a significant bottleneck in the simulation. Having attempted logging to files, tests were run attempting to log results to a database, which was even slower. Ultimately, it was decided that Apache Log4j 2[18] would be a suitable candidate, allowing the logging mechanism to meet the demands of the simulation. Elementary testing showed that Log4j was more than capable of meeting the logging requirements and removed the bottleneck from the process.

The nature of distributed systems means that uniqueness is sometimes difficult to guarantee, therefore, exact duplicates might appear. The causes of this are not always certain, however it is conceivable that the same event could occur more than once, causing multiple records. Therefore, a mechanism of filtering out duplicate records was required. To achieve this, it was decided that records from the logging process would be loaded into a database, where the duplicates could be filtered out on various criteria.

5.6.2 Logging format

Although Log4j is much more powerful and capable than its purpose in this project, it is nonetheless well suited. Objects are serialised and supplied to Log4j to be logged and this object serialisation process is explained in section 5.6.

Since Log4j allows classes to retrieve a logger based on name as well as other methods, all classes append to the same log in this project. This is more flexible than the built-in Urbansim logging, since it creates just one file per data type to be recorded - adverts and billing records. Processing results is then an easier process, since there are only one or two files to read as opposed to several. In addition, the configuration file for Log4j is very simple.

A logging requirement, as noted is that each record should be somehow uniquely identifiable. Requiring such an attribute gives a double benefit, in that devices in the simulation can distinguish objects from one another as well as enabling the results process to uniquely identify objects. To achieve this, each object that needs to be identified in such manners is assigned a unique identifier. Doing so guarantees that it is possible to collate records according to objects and extract meaningful data from the records.

Other information is also included in logging, which might not be needed in a practical implementation of this project. These fields, such as the time received or current location have been selected in such a way that the resulting overhead is as small as possible, to avoid negatively impacting the rest of the simulation. An example of where fields had to be reconsidered is in section 4.4.1.

Summary

In this chapter, the development process was outlined, along with the tools and backup procedures used alongside development. Modifications to Urbansim were discussed along with simulator features that were not used.

The implementation of devices, including the main loop of the device agents was described, immediately followed by notes about the implementation of the peer-to-peer protocol.

[18] `http://logging.apache.org/log4j/`

An important subject, object serialisation is explored in detail along with its relation to logging. Finally, details surrounding the logging format are discussed.

Chapter 6

Testing and Evaluation

This chapter will describe the testing processes, architecture and framework used in the evaluation of the project. System and component testing approaches are discussed in detail along with a list of relevant functionalities.

The architecture of the evaluation framework is then described in detail before discussing details of the experiment simulations and the proposed hypotheses. After which, the results are analysed, discussed and explained.

6.1 Software and system testing

The following sections will discuss how the system was tested both as a whole, from a high-level overview, and as individual methods.

6.1.1 System testing

Following the nature of cyclical, incremental development, as well as exploratory programming, the system was tested gradually as functionalities were developed. This was required since functionalities often depend on one another, so it is imperative that they behave as expected before they are used. The testing approach used is mostly grey-box, meaning a combination of white-box and black-box testing [10]. This means that functionalities are inspected at both the code and system levels to ensure the individual methods are working as expected and, as a result, the system also behaves as expected.

In terms of testing mechanisms, diagnostic messages printed to the console were the main and preferred method of debugging. Since the simulator starts many threads during its operation, it is not really feasible to attach a debugger to the JVM[1]. This is the same reason for not using a unit testing library such as JUnit[2], along with the fact that learning how to use the JUnit framework would have required a considerable investment of time and effort. As a result, printing to the console seemed to be the next simplest method of debugging and testing, allowing variables and attributes to be printed in context, giving a full picture of how the code was behaving.

The console print-outs were obtained by running simulations locally, as mentioned in section 5.1. These simulations were only run for long enough to observe obvious issues and often running for a maximum simulated time of around 3 minutes. For most bugs and issues, this was sufficient time to expose the symptoms for diagnosis.

[1] `http://docs.oracle.com/javase/8/docs/technotes/guides/vm/`

[2] `http://junit.org`

Functionalities per development cycles		
First cycle	Second cycle	Third cycle
Primitive message exchange	Adverts move between devices	Simulation parameters from XML
Rudimentary advert model	Billing records created at auction	Second price auction with reserve
Basic auction sorted by bid	Internet billing record propagation	Logging with Log4J
Early JSON serialisation	Basic device classification	Improved advert logging fields
Shared objects with cloning*	Devices log to files*	Ruby controls simulation automatically
Devices collect adverts	Database logging*	Test run generation in SQL
Geofencing for adverts	Created templates for XML files	Concurrent result processing
Advert expiry dates	Randomised advert generation	Adverts matched to prospective devices
Hard-coded adverts in devices	Adverts imported from XML	Adverts wait until release date
Advert budgets	Processing JSON log with Ruby	Results summarised automatically

Table 6.1: The functionalities developed according to development cycle

Table 6.1 shows the overall features and functionalities that were developed, and which cycle they were developed in. During each cycle, various different ways of achieving a feature were explored before a particular one was chosen. These are listed and marked with an asterisk since they were developed but have since been removed.

The first cycle includes the development of basic and fundamental components such as devices and adverts. During this cycle, various parameters and other data were hard-coded so that testing and development could continue. Most of these parameters were then stored in XML files in later cycles. By the end of the first cycle, adverts were being shared between devices and had a budget to spend.

The second cycle improves the protocol to ensure adverts move from one device to another, rather than being copied and creating duplicates. Billing records are introduced and linked to auctions to keep a record of when an advert has been successful in an auction. This second cycle also contained various different logging mechanisms that were mentioned in section 5.6.1. Creation of the automated evaluation framework as described in section 6.2.1 began towards the end of the cycle. Once the second was complete, randomised adverts were created by the evaluation framework and read in to the simulation from XML files. The resultant log files containing the JSON objects from the simulation were also able to be processed by the evaluation framework.

The third cycle continues the development of XML configuration file processing, where the simulation parameters are read in from an XML file. Various features enter their final development state and do not receive any additional development. For instance, the logging mechanism is changed to use Log4j, the fields captured by adverts to track their location are altered and no further changes are made to that process.

6.1.2 Component testing

As mentioned previously in section 6.1.1, all features and functionalities were tested as they were developed to ensure tight integration between common components. Table B.4.2 shows the functionalities tested as part of a device.

Method	Status	Expected result
constructor	Passed	Sets up necessary variables such as the corresponding device in the simulator
readSimulationSettingsFromXML	Patched	Reads simulation parameters from the supplied XML file and sets default values if invalid
readAdvertsFromXML	Patched	Reads adverts for the simulation from the supplied XML file
main	Passed	The main loop executed by the simulator
connectNeighbours	Passed	Scans for devices in range and connects to them
requestNeighbourDetails	Passed	Sends a REQUESTDETAILS message to all neighbours
forwardAdverts	Passed	Sends a collection of adverts to each neighbour
matchAdvertsAndDevices	Patched	Matches a subset of adverts to a neighbour
allocateAdsToDevice	Patched	Selects a subset of adverts for a given neighbour
forwardBillingRecords	Passed	Invokes the appropriate mechanism to transfer billing records
uploadBillingRecords	Passed	Mimics uploading records via the internet
pushBillingRecordsOutwards	Patched	Attempts to send billing records to internet capable devices
pushBillingRecordsInwards	Patched	Attempts to send billing records to devices closest to the billing server
listenForReplies	Passed	Listens to the message queue and calls the appropriate methods to handle messages
handleDeviceDetails	Passed	Add the received device details to the neighbour cache
handleAdvert	Patched	Add the received advert to the collection if possible
handleAdvertAck	Passed	Remove the supplied advert from the collection
handleBillingRecord	Passed	Add the supplied billing record to the collection if possible

Method	Status	Expected result
handleBillingAck	Passed	Remove the supplied billing record from the collection
assignLocationToAdverts	Passed	Assigns the current location to all adverts in the collection
cleanOutAdverts	Patched	Removes adverts that are expired or have no budget left
selectValidAdverts	Passed	Encapsulates the rules for adverts that are eligible for display and propagation
checkNextAuction	Passed	Calls for an auction if it is the right time
holdAuction	Patched	Runs the auction based on the current adverts in the collection
displayAdvert	Passed	Encapsulates a device displaying an advert, creates billing record where appropriate
cleanOutNeighbourDetails	Passed	Empties the cache of neighbouring peers
disconnectAll	Passed	Disconnects all current neighbour connections
getDeviceDetails	Passed	Returns a DeviceDetails object for the current device
sendDeviceDetails	Passed	Sends a DeviceDetails object to the supplied peer
sendAdverts	Passed	Sends the supplied subset of adverts to the supplied peer
sendAdvertAck	Passed	Sends an AdvertAck and the supplied advert to the supplied peer
sendBillingRecord	Passed	Sends the supplied billing record to the supplied peer
sendBillingAck	Passed	Sends a BillingAck and the supplied billing record to the supplied peer
sendMessage	Passed	Abstracts from sending a message to a peer, adds a delay if required
logAdvert	Passed	Serialises the supplied advert and logs it with Log4j
logBillingRecord	Passed	Serialises the supplied billing record and logs it with Log4j

Method	Status	Expected result
canHaveInternet	Passed	Decides whether the device is capable of an internet connection
hasInternet	Passed	Decides whether the device currently has an internet connection

6.1.3 Known bugs

There is one known bug, which is that non-display devices can sometimes hold auctions, which is not the behaviour designed. Fixing this bug will be relatively trivial thanks to the modular approach to coding, and seems to have been missed during testing.

6.2 Evaluation

In this section, the results from many test runs will be presented, in combination with the experiments and their expected outcomes. In order to gather these results, an evaluation framework was proposed and developed.

6.2.1 Evaluation framework

To present good quality and meaningful results, several simulations would be needed to provide data for each experiment. As these simulations can be time consuming, it was not sensible to run them all manually. Therefore, an automated framework was developed to run test simulations automatically. The overall architecture of the framework is shown in Figure 6.1.

Hypervisor and storage

A hypervisor[3] was used to create virtual machines which would run the simulation. XenServer(Xen)[4] was used because it's free, open source and has no memory or other resource limitations beyond the hardware, unlike VMware[5]. XenServer also appears to have better performance for linux-based virtual machines, given previous experience. Nexenta NexentaStor Community Edition[6] was used to provide storage to XenServer over Fibre Channel[7]. NexentaStor was used because of its Solaris 11[8] roots as well as its ZFS[9] capabilities. One of the main benefits of NexentaStor is that it can make use of the server's RAM as a cache for the storage it presents to other servers. Having such an architecture means that the storage system generally performs faster. To take advantage of this caching, the NexentaStor server was fitted with 24 GB of RAM.

The physical server used for XenServer was a somewhat more substantial machine. In order to run as many concurrent simulations as possible, the XenServer machine has a total of 16 cores and 72 GB of RAM. In total, along with the required infrastructure machines, a total of seven

[3]http://en.wikipedia.org/wiki/Hypervisor
[4]http://xenserver.org/
[5]http://www.vmware.com/uk/products/vsphere-hypervisor/
[6]http://nexenta.com/products/nexentastor
[7]http://fibrechannel.org/
[8]http://www.oracle.com/us/products/servers-storage/solaris/solaris11/overview/index.html
[9]http://docs.oracle.com/cd/E19253-01/819-5461/zfsover-2/

worker machines were able to fit onto the server. These virtual machines are discussed in more detail in section 6.2.1.

Virtual Machines

Several worker virtual machines (VMs) were created to run simulations automatically as mentioned in section 6.2.1. Each worker VM originally had two cores, but was later increased to four cores since, although the simulations place a heavier demand on RAM than on CPU, the extra CPU helped with processing the simulation results in a timely manner. Each VM also had 8 GB of RAM to support the needs of the simulator. During the experiments, it was observed in passing that the RAM usage of each machine increased during the results processing phase. Ubuntu 14.04 Server[10] was installed on each worker VM, as well as some of the infrastructure VMs. Ubuntu was used mostly out of familiarity and ease of setup compared with some other distributions. In his user manual, Skinner also suggests Linux[11] as a platform for the simulator, since that is the Operating System (OS) that the simulator was originally developed on [19]. Using Linux also provides easy options for configuration management, in turn simplifying the process of managing the VMs.

The additional infrastructure VMs were created to support the VM setup process as well as the VMs in general. This infrastructure included a Windows Server VM for administration of XenServer, a NFS[12] server to serve ISO[13] images so that they could be used to install Ubuntu on the VMs. Finally, another VM was created to host configuration management software, as mentioned in section 6.2.1. The worker VM architecture is shown in Figure 6.2.

Configuration management and orchestration

Puppet[14] and Foreman[15] were set up to control the overall configuration of the simulation workers. This allows one configuration to be defined, and all machines will be configured as desired. Using such a system was deemed very desirable to avoid repeating the same process for each of the seven worker VMs. An added benefit of using such tools is that if a worker VM needed reinstallation for whatever reason, the process was very quick and easy.

Foreman and Puppet work well together, creating a flexible configuration management system. Unfortunately there is usually a learning curve when creating configurations to control settings or aspects of an OS that one has not experienced before. During the setup of the worker VMs, issues were encountered when installing Ruby, however these were eventually fixed. Ultimately, despite the learning curve, using these tools saved time versus administering each VM individually and ensured consistency among them.

Ansible[16], a Python-based orchestration and configuration management tool used as an orchestration tool, since Puppet was already in use for configuration management. Ansible was deployed after Puppet, so it was possible to use Puppet to put the required structure in place for

[10] `http://www.ubuntu.com/server`
[11] `http://en.wikipedia.org/wiki/Linux`
[12] `http://en.wikipedia.org/wiki/Network_File_System`
[13] `http://en.wikipedia.org/wiki/ISO_image`
[14] `https://puppetlabs.com/`
[15] `http://theforeman.org/`
[16] `http://www.ansible.com/home`

Ansible. Ansible has mostly the same capabilities as Puppet but was used to run the same command across all or some machines quickly. A common occurrence was to run a command to check how many records were being produced during the simulation, or whether there were any errors that were not being captured correctly.

Results and Control Database

As mentioned in section 5.6.1, a database was used to collect logged records and filter out duplicates. The same database was also used to provide simulation configuration and co-ordinate the workload across the worker VMs. PostgreSQL[17] running on Ubuntu was chosen because of its good performance and predictable behaviour. Aside from being open source, it is highly customisable and tunable; benefits which were taken advantage of during the database setup.

The database server was not running on XenServer, since it was filled with as many workers as possible. As a result there were no available resources left in terms of both RAM and storage space. Therefore, a spare desktop PC was repurposed to serve the database for the evaluation framework. Originally the database server was running as a VM in VirtualBox[18] on Windows. Unfortunately although Windows was installed on a solid-state drive[19], the database performed slowly, and contributed a detrimental bottleneck to the evaluation framework. To improve the performance, the virtual machine was migrated and reinstalled directly on the desktop hardware, using storage from the NexentaStor server mentioned in subsection 6.2.1. This meant that the performance of the machine improved greatly given it had access to all of the resources of the desktop machine, as well as the performant storage of the NexentaStor server. Despite these efforts, the database server still remained a bottleneck, however its effects were significantly reduced after this migration.

Appropriate indexing was used to ensure that queries across large data sets were still performant, and the memory settings of the PostgreSQL database were carefully tuned to prevent the database from using swap space[20].

6.2.2 Evaluation Process

To run the evaluation process, a bash shell script was written to invoke a separate Ruby script when a worker VM started up. This was done since the script also rebooted the VM if the simulation was successful, and therefore needed higher privileges than the Ruby script. The Ruby script did not need such privileges and was run normally.

Ruby

Ruby was chosen as the language to develop most of the automated evaluation mechanism. This is because Ruby is quick to develop with and makes the overall development process very easy. The Gem[21] package control mechanism is flexible and provides access to useful libraries without much developer intervention. Since Ruby is also an interpreted language, the code could be placed on the target machines and run easily without requiring compilation.

Ruby as a language also includes many useful features by default without the use of separate

[17] http://www.postgresql.org/
[18] https://www.virtualbox.org/
[19] http://en.wikipedia.org/wiki/Solid-state_drive
[20] https://www.centos.org/docs/5/html/5.2/Deployment_Guide/s1-swap-what-is.html
[21] http://guides.rubygems.org/what-is-a-gem/

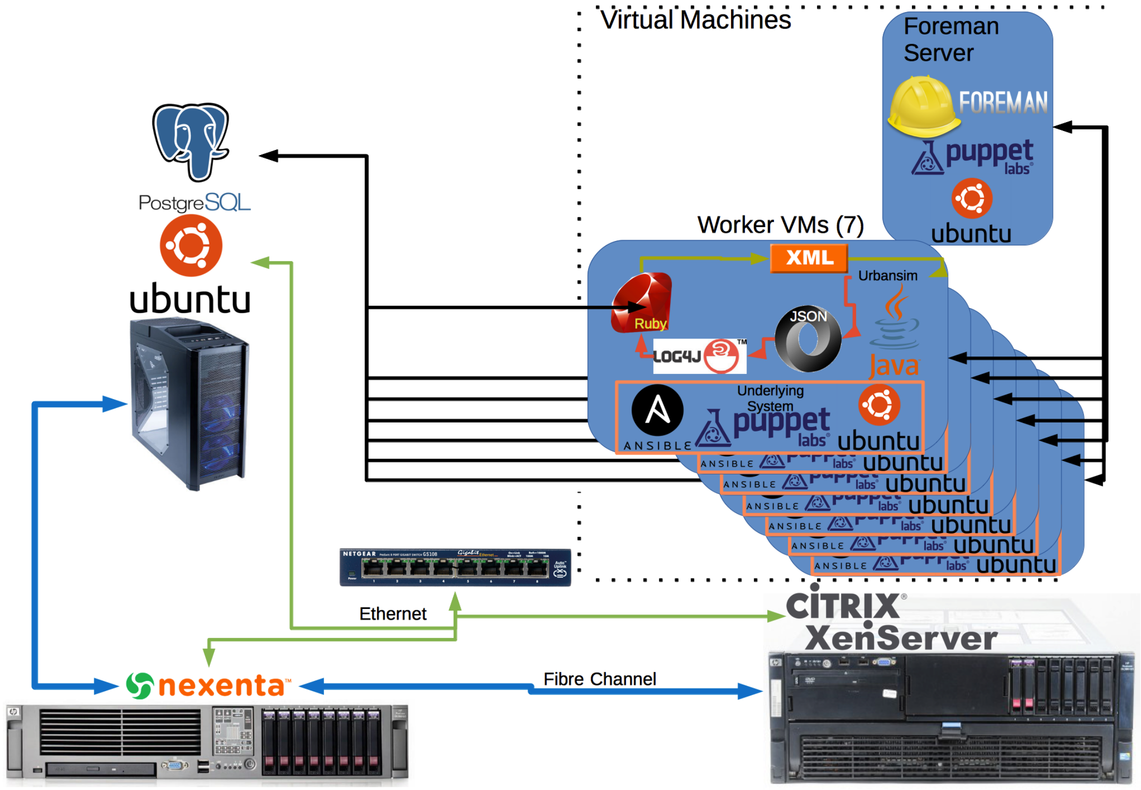

Figure 6.1: Evaluation Framework Overview

libraries. Since the results were stored in JSON, as mentioned in section 5.6, Ruby was a logical choice since parsing JSON is a native and simple feature.

Ruby's ERB templates were another reason for its use. The simulator's configuration files use XML and so ERB templates were made to allow those files to be dynamically created according to need. This again was very quick to do and provides a great amount of flexibility.

In addition to configuration values, adverts were dynamically generated by the Ruby script with some parameters randomised, and the adverts were then injected into the devices through their relevant XML files.

More details on the steps involved in the automatic process are detailed in Appendix C.

6.2.3 Experiment simulations

Each simulation ran for a simulated time of two hours to provide a fair chance of adverts being spread around the network. This required that the configuration files for SUMO were regenerated, since they did not have enough data for two hours' of simulation. Once the guide from Skinner [19] had been consulted, this was relatively trivial and the new files were pushed to the VMs with Puppet.

Overall, two variables are being controlled and two attributes are being observed. The advert budget and device density are varied while the device revenue and advert propagation are being measured. The budget ranged from ten to forty credits, increasing in steps of ten. These simulations were also performed with different numbers of adverts in the network to produce a more broad spectrum of results. Parameters such as an advert's release date, expiry date, bid, radius

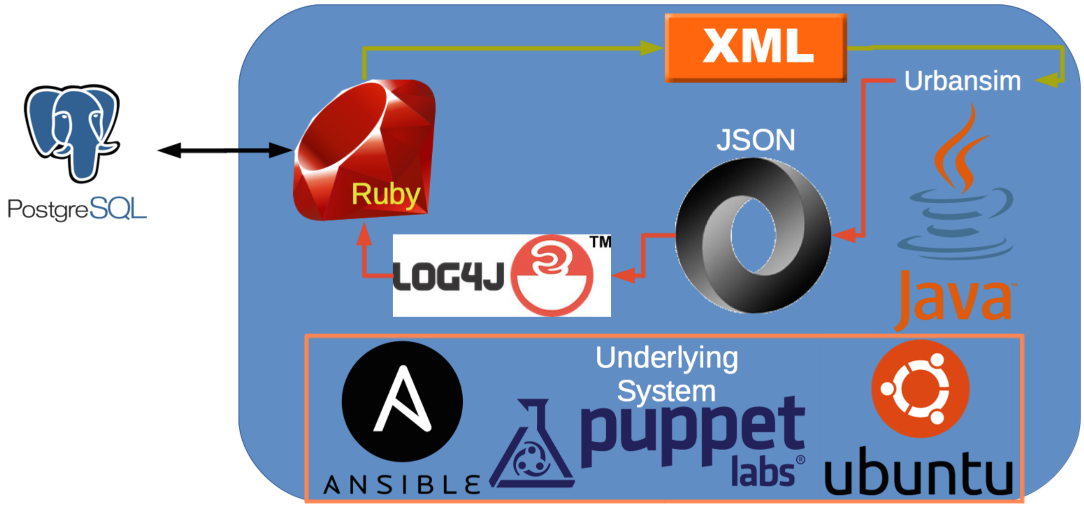

Figure 6.2: Worker VM architecture

were randomised for each advert and each simulation. Parameters controlling the behaviour of the network, such as the auction period, reserve price and maximum adverts per device were set to sensible values and kept constant for all simulations. For example, the auction period was set to five minutes of simulated time, meaning a maximum total of 24 auctions would occur per device.

For each combination of parameters, twenty-five simulations were run to collect enough data for analysis. These results were then averaged using the mean to give a representative value for each variation of the parameters.

6.2.4 Experiment hypotheses

After the variables and steps for the experiments were decided, a number of hypotheses were proposed to be tested by the experiments. These are as follows:

1. As the density of devices increases, the more credit is accrued by devices
   (a) The total amount of credit spent by an advert is proportional to device density
   (b) The rate an advert's budget is spent is proportional to device density
2. The distance travelled by an advert is proportional to the advert budget
   (a) The number of devices that receive an advert is proportional to the advert budget and device density

The results from the simulations are presented over the following pages. Results corresponding to hypotheses 1, 1a and 1b will be presented first.

Figure 6.3 shows the increase in revenue accrued by devices as the overall advert budget increases. As the budget starts out at 10 credits, both densities show devices accruing a moderate amount of credit. Devices accrued a mean of 17.26 credits in the denser network and 16.14 credits in the sparser network, giving a difference of 1.12. After the budget increases to 20 credits, the difference between the densities remains similar, 1.27 for a budget of 20, versus 1.12 for a budget of 10.

Once the budget reaches 30 or 40 credits, an interesting observation can be made. With a sparser network, increasing the budget from 30 to 40 has much less of an effect than in the denser

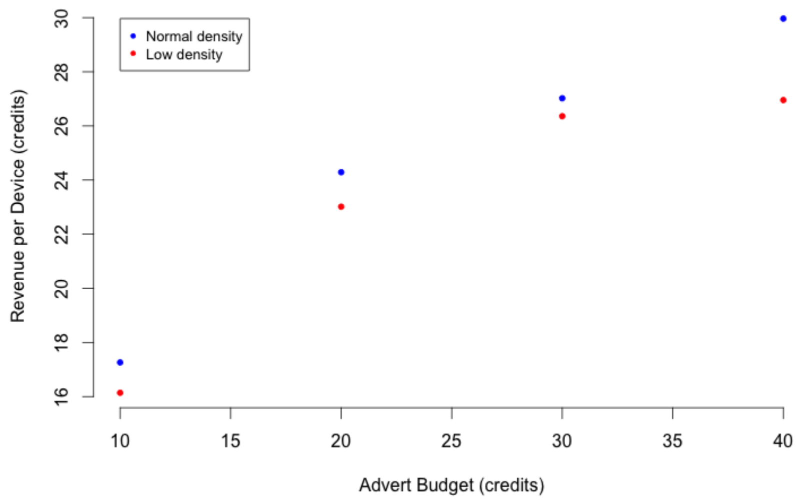

Figure 6.3: Device Revenue vs Advert Budget

network implying that a limit has been reached. This is most likely because in the sparser network, there are fewer devices to hold auctions, and therefore the advert budget is less likely to be depleted at the same rate as it would in a denser network. This is in line with what can be observed from using an advert budget of 40 in a denser network. The revenue accrued by devices increases as expected, with devices accruing a mean of 29.97 credits in the denser network.

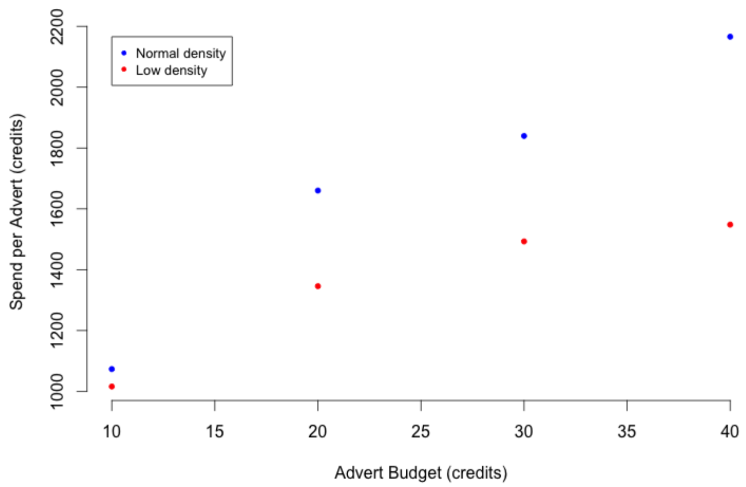

Figure 6.4: Advert Spend vs Advert Budget

Figure 6.4 shows the increase in advert spending as the overall advert budget increases. The spend of each advert provides some interesting information about the behaviour of the network. The first and most important observation to make is that the Y-axis scale begins at 1000, despite the fact that the lowest value used for the advert budget was 10 credits. This observation clearly suggests that the budgeting mechanism for adverts is not working correctly as intended.

There is a plausible explanation for this behaviour. The protocol defined in section 4.4.2 defines an ADVERTACK message, which is used to acknowledge when an ADVERT message was successfully received by the recipient. The business logic triggered by these messages should ensure that as an advert is sent from one device to another, it is removed from the sending device, to ensure that only one copy of the advert exists in the network.

It is possible, during this message exchange, that the two devices would move out of each other's range after the ADVERT was received. If this is the case, the ADVERTACK message would never reach the intended recipient, and the advert would then exist on both devices, creating a duplicate and effectively doubling the budget available to the advert. This issue is further explored in section 7.2.

The differences in advert spend as the advert budget is varied match with the results displayed in Figure 6.3, as one would expect. Once again, in a sparse network, higher budgets seem less influential than when the network is more densely populated.

Overall, the differences between the network densities are slight when observing device accrued credit rather than spend per advert. This would suggest that the credit spent by the adverts is being distributed relatively evenly among devices. Therefore, it would indicate that the more influential variable on credit accrued by devices is the advert budget, although there is a moderate increase in accrued credits between the two network densities. This supports hypothesis 1.

Considering hypotheses 1a and 1b along with Figure 6.4, the differences in advert spend between the network densities for each value of the advert budget increase with each step of the advert budget. As hypothesised, not only does the total amount of credit spent increase as density increases, but also the rate at which it is spent also increases.

Next, results corresponding to hypotheses 2 and 2a will be presented. Measuring the propagation of adverts within the network may seem somewhat redundant since they are all contained within a geofence, however the random variations created when simulations are started should ensure sufficient variety.

Beginning with Figure 6.5, the mean distance travelled per advert remains mostly unchanged at around 221 regardless of the value of advert budget, with the exception of when the budget is 30 where it drops to approximately 218. Considering that the mean distance travelled in the sparse network with an advert budget of 40 is very similar to when the budget is 10 or 20, the result for a budget of 30 could be considered anomalous and may require repeated testing.

The comparatively low values for distance travelled when the advert budget is 10 are to be expected. With a low budget, it is likely that the budget will be depleted and the advert will be removed sooner than if its budget had been greater. When the budget is 20, 30 or 40, the denser network appears to allow adverts to travel a greater average distance. Since adverts are not restricted in terms of which devices they can be sent to, it is not surprising that the budget does not seem to have a strong effect on how far the adverts travel, with the exception of low budgets such as the first result in Figure 6.5.

Figures 6.6 and 6.7 respectively show how many unique devices and non-unique devices adverts were sent to. This gives an overall indication of an adverts propagation as well as how

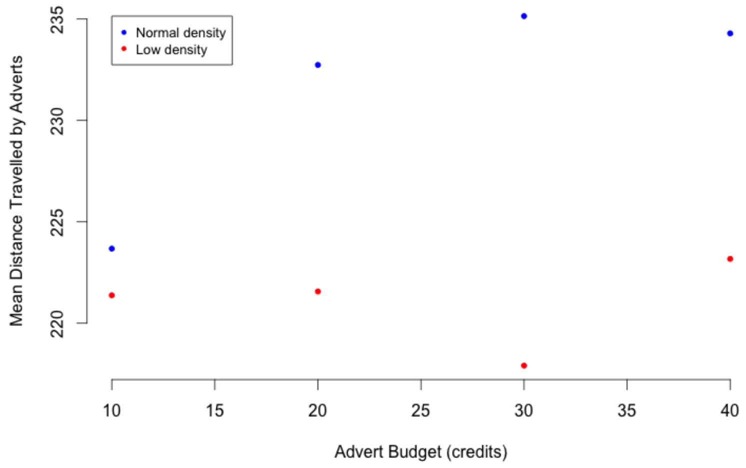

Figure 6.5: Mean distance travelled per advert

many times it was shared. The proportions between the individual points of Figures 6.6 and 6.7 appear to be similar across both graphs.

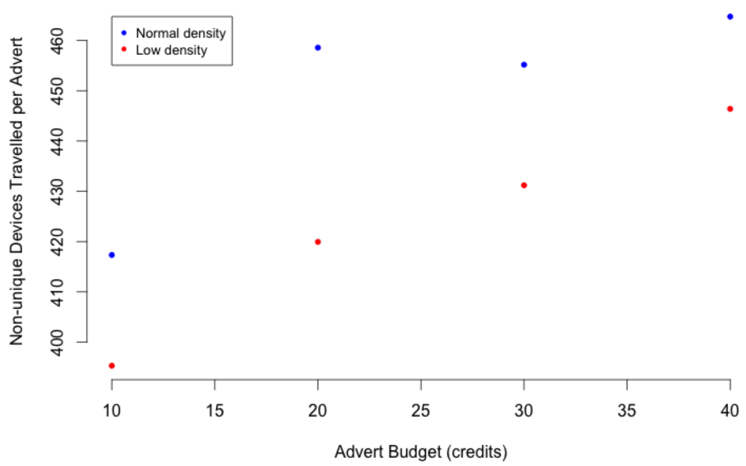

Figure 6.6: Total devices travelled per advert

It seems logical that the differences would be of a similar relative magnitude, as the number of unique devices will be a numeric factor of the number of total devices that an advert will travel through. Just as with Figure 6.5, the value for when the advert budget is 10 is much lower than when the budget is 20, 30 or 40. It is presumed that the same reason would apply, that is that the low budget was depleted quickly, causing the advert to be removed from the network.

It is interesting to observe that with each increase in advert budget, adverts travel through more devices in the sparser network. A proposed reason for this is that in a sparser network, there are fewer opportunities to exchange adverts with other devices. As a result, it is likely that there will be less variety of adverts available for auction, and therefore less competition. If an advert is

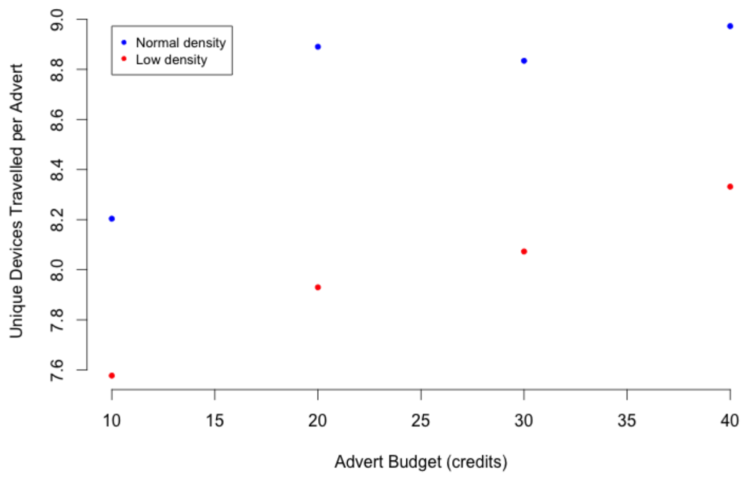

Figure 6.7: Unique devices travelled per advert

unable to be transferred to another device, and it frequently succeeds in auctions, its budget will deplete at a faster rate. Assuming that this is the case, adverts may be less likely to propagate if their budget is not large enough.

Overall, having observed the results from Figures 6.5, 6.6 and 6.7, hypotheses 2 and 2a should be considered. Given that no significant change in distance travelled was observed as the advert budget increased, particularly for the denser network, hypothesis 2 does not hold completely true. The same patterns were observed in the number of devices an advert travelled through and therefore, since there was little response to the changes in the advert budget, 2a does not hold completely true either.

Summary

In this chapter, the methods of system and component testing were outlined and described in context. The evaluation framework is then explained in detail alongside its architecture. The automatic evaluation process is then described.

The experiments used in the evaluation are then outlined, along with five proposed hypotheses. The results are then analysed, with three of the five hypotheses being supported by the observed behaviour.

Chapter 7

Conclusion, Discussions and Future Work

This chapter will conclude the work for this project, followed by a discussion about the development experience and any possible bug fixes. The chapter ends by proposing possible future work for the project.

7.1 Conclusion

Overall, the project has satisfied the requirements set out in sections 3.1. The project has also shown support for three of the five hypotheses defined in section 6.2.4 and given an indication that such a peer-to-peer network would be possible between devices in a smart city.

All of the non-functional requirements as laid out in section 3.2 have also been met as a result of the technologies used and the process that was followed.

There are two bugs which are still present in the project. One of which, mentioned in section 6.1.3, causes non-display devices to run auctions when they should not. This bug originates from an oversight during development.

The second bug, mentioned in section 6.4 enables adverts to spend much more credit than their allocated budget. Potential solutions to this are discussed in section 7.2.

Had this project been attempted from the start again, efforts would have been made to ensure there was sufficient time for more varied testing and evaluation.

7.2 Discussion

The development process overall went well, although during development, various issues and bugs were created through rushing or lack of due care and attention. These could have been easily avoided if more care was taken while coding, saving time later in the project. This project has also provided useful learning opportunities, allowing for the exploration into performance of distributed systems, systems programming and automation, as well as peer-to-peer protocols and mechanisms. Fortunately it was very rare that any of the disaster recovery methods described in section 5.1.1 were put to use in recovering data.

7.2.1 Proposed bug solutions

The bug causing adverts to spend several times their given budget is quite serious, and a fix should be found. If a viable fix is not possible, the budget functionality should be removed and replaced with a suitable alternative.

One proposed solution to this problem lies in how the adverts are received by devices. In the current version, adverts are checked for uniqueness using their identifier. When a device receives an advert, if it already holds an advert with the same identifier, the received advert is simply ignored. However there is an opportunity to inspect the received advert and check its budget value with the budget value of the received advert. If the budget of the received advert is less than that of the advert that already exists on the device, the latter could be updated to match the new budget.

Although this would not completely solve the problem, it would lessen the severity and more closely limit how much an advert can spend.

Another solution would be to ensure that adverts cannot spend over a certain amount per device, keeping track of these values on the device itself.

Finally, it would be possible to allow adverts to propagate to multiple devices, providing that the budget is divided before the adverts are sent. However, this has its own set of drawbacks, foremost of which is the cause of the original overspending problem. That problem arises as devices move out of range as messages are being sent, causing the messages to fail. If an advert's budget is split and it fails to send successfully or no acknowledgement is received, there is the potential to further duplicate the budget, or to split it more times than necessary.

If an advert budget is allowed to be split in this way, it must be sufficiently large when the advert enters the network.

There are many potential features that could be explored within this system, and many parameter combinations which could be evaluated. Several variables and weights used during the automatic generation of adverts could have been made configurable to give the system more flexibility.

Although the majority of the project code is well structured and abstracted, the methods to parse the configuration values from the XML files need improving. Currently they are quite verbose and not immediately self-explanatory.

7.3 Future Work

Since this project is relatively new, there is substantial scope for future work, additional development and further evaluation.

7.3.1 Advert taxonomy and representation

This system would be a good candidate for knowledge inference and reasoning to enable semantic matching of adverts to devices during propagation and auction. This would enable adverts to more intelligently propagate through the network, and 'collect' in areas where they are most applicable in terms of content and target audience. If used in conjunction with geofencing, this categorisation of adverts would restrict certain types of adverts to certain geographic locations, improving the targeting capabilities of the system.

7.3.2 Advert propagation and repetition

There are a few aspects of advert propagation and some mechanisms to control it that could be explored.

The concept of a 'sliding window' was discussed which would prevent adverts from either propagating, taking part in an auction or both, depending on the advert's behaviour within a certain time period. Depending on the functionality developed, such a feature could prevent the same adverts repeatedly succeeding in auctions.

Moreover, it would be interesting to test the effects of advert propagation restrictions on metrics like device revenue. For instance, investigating how the revenue might change if a device is only allowed to propagate within its geofence.

Another interesting possibility would be to assign each advert a TTL as well as an expiry date, and observe how this affected the propagation as well as the total spend of the advert in relation to its budget.

Further possibilities are discussed in Appendix D.

This project has a lot of potential to be used for research into the areas discussed in section 7.3, and also potential to be developed into a more comprehensive solution.

Bibliography

[1] Aggarwal, G., Goel, A., and Motwani, R. (2006). Truthful auctions for pricing search keywords. In *Proceedings of the 7th ACM Conference on Electronic Commerce*, EC '06, pages 1–7, New York, NY, USA. ACM.

[2] Aggarwal, G., Muthukrishnan, S., Pál, D., and Pál, M. (2008). General auction mechanism for search advertising. *CoRR*, abs/0807.1297.

[3] Aggarwal, G., Muthukrishnan, S., Pál, D., and Pál, M. (2009). General auction mechanism for search advertising. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 241–250, New York, NY, USA. ACM.

[4] Ashlagi, I., Braverman, M., Hassidim, A., Lavi, R., and Tennenholtz, M. (2010). Position auctions with budgets: Existence and uniqueness. *The BE Journal of Theoretical Economics*, 10(1).

[5] Baset, S. and Schulzrinne, H. (2004). An analysis of the skype peer-to-peer internet telephony protocol. *CoRR*, abs/cs/0412017.

[6] Buchan, B. (2012). Peer-to-peer networks using mobile devices and bluetooth. Undergraduate honours thesis, Department of Computing Science, University of Aberdeen.

[7] Caragliu, A., Del Bo, C., and Nijkamp, P. (2011). Smart cities in europe. *Journal of Urban Technology*, 18(2):65–82.

[8] Chourabi, H., Nam, T., Walker, S., Gil-Garcia, J., Mellouli, S., Nahon, K., Pardo, T., and Scholl, H. J. (2012). Understanding smart cities: An integrative framework. In *System Science (HICSS), 2012 45th Hawaii International Conference on*, pages 2289–2297.

[9] Dillet, R. (2014). Spotify removes peer-to-peer technology from its desktop client.

[10] Doungsa-ard, C., Dahal, K., Hossain, A., and Suwannasart, T. (2007). Test data generation from uml state machine diagrams using gas. In *Software Engineering Advances, 2007. ICSEA 2007. International Conference on*, pages 47–47.

[11] Edelman, B., Ostrovsky, M., and Schwarz, M. (2005). Internet advertising and the generalized second price auction: Selling billions of dollars worth of keywords. Working Paper 11765, National Bureau of Economic Research.

[12] Geddes, B. (2014). *Advanced Google AdWords*. Wiley.

[13] Gregori, E. (2002). *Web engineering and peer-to-peer computing Networking 2002 workshops, Pisa, Italy, May 19-24, 2002 : revised papers*. Springer, Berlin New York.

[14] Krishna, V. (2010). *Auction theory*. Academic Press/Elsevier, Burlington, MA.

[15] Meyer, D. (2003). *The economics of risk*. W.E. Upjohn Institute for Employment Research, Kalamazoo, Mich.

[16] Nam, T. and Pardo, T. A. (2011). Smart city as urban innovation: Focusing on management,

policy, and context. In *Proceedings of the 5th International Conference on Theory and Practice of Electronic Governance*, ICEGOV '11, pages 185–194, New York, NY, USA. ACM.

[17] Qin, T., Chen, W., and Liu, T.-Y. (2014). Sponsored search auctions: Recent advances and future directions. *ACM Transactions on Intelligent Systems and Technology (TIST), to appear*.

[18] Rose, A. (2008). Introducing bbc iplayer desktop.

[19] Skinner, A. (2014). Peer-to-peer for information sharing in urban scenarios. Undergraduate honours thesis, Department of Computing Science, University of Aberdeen.

[20] Steinmetz, R. (2005). *Peer-to-peer systems and applications*. Springer, Berlin New York.

[21] Taylor, I. (2009). *From P2P and grids to services on the web evolving distributed communities*. Springer, London.

[22] Vickrey, W. (1961). Counterspeculation, auctions, and competitive sealed tenders. *The Journal of Finance*, 16(1):8–37.

Appendix A

User Manual

This guide will demonstrate how to run a simulation.

A.1 Requirements

Since this project runs with Urbansim, an understanding of how to use the Urbansim project is required. This can be obtained from [19]. This system is designed for use on Linux and Mac OS X. Please ensure the following software is installed before continuing:

- Oracle Java 7
- SUMO

To install java, please consult the Oracle website[1]. SUMO can be installed through homebrew[2] on Mac OS X or through binaries[3] on Linux.

A.2 Run an Existing Simulation Configuration

This project uses Urbansim and the operating instructions are very similar as in [19]. Below is a simplified version of the manual.

1. Open a terminal and navigate to the project folder. The folder should contain the folders 'jar' and 'lib'.
2. To run the simulator, run this command from the project directory:
   `bash ./run -file test/adverts/case.xml`
   This assumes that Java is correctly installed in the conventional location.
3. The program will then launch. You will be presented with 2 main windows as shown in Figure A.1.
4. Click on the 'play' button to launch the simulation. The simulation can then be paused or stopped. Once the simulation is paused, the play button can be used to advance the simulation by a single step.
5. To end the simulation, click the stop button.

[1] `http://docs.oracle.com/javase/7/docs/webnotes/install/`
[2] `http://brew.sh/`
[3] `http://sumo.dlr.de/wiki/Installing/Binary`

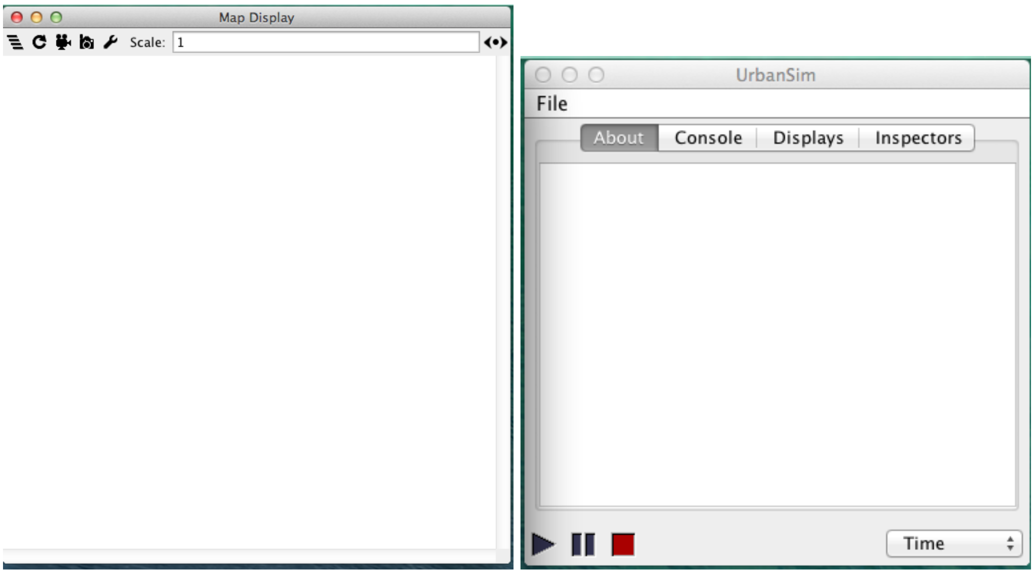

Figure A.1: Urbansim starting windows

A.3 Creating scenarios

Creating new scenarios is out of the scope of this manual. Please consult the user manual in [19].

Appendix B

Maintenance Manual

B.1 Prerequisites

As per the Urbansim documentation, this project can only be run on 64-bit operating systems [19]. Since this project runs with Urbansim, an understanding of the Urbansim project is required. This can be obtained from [19]. Testing has shown that with large networks and large numbers of adverts, the simulation can easily consume 7 GB of RAM. Therefore it is suggested that a suitable computer is used.

B.1.1 Dependencies

The following are required as dependencies and must be installed separately to the project:

- Oracle Java 7[1]
- SUMO and tools[2]
- Eclipse IDE[3]

The project is split across two Eclipse IDE projects: AdDistrib and Simulator (Urbansim). The AdDistrib project depends on the following libraries, included in the `AdDistrib/lib` folder:

- commons-codec-1.10.jar - Apache Commons Codec
- Continuations.jar - Required by Urbansim
- gson-2.3.1.jar - Google JSON library
- log4j-api-2.2jar - Apache Log4j 2 Interfaces
- log4j-core-2.2.jar - Apache Log4j 2 Implementation
- mason.jar - Required by Urbansim

The Simulator project depends on the following libraries, included in the `Simulator/lib` folder:

- ant.jar - Apache Ant library

[1] `http://docs.oracle.com/javase/7/docs/webnotes/install/`
[2] SUMO can be installed using Homebrew (`http://brew.sh` on OS X or through most Linux package managers
[3] `https://eclipse.org/`

- asm-all-4.0.jar - Dependency of MASON
- Continuations.jar
- log4j-1.2.17.jar
- mason.jar
- traci4j-1.6.jar

B.2 Installing

B.2.1 Extracting Files

1. Open a terminal and navigate to the desired destination directory.
2. Extract the files with the following command: `tar -xf StuartKregor_Alexander.tar.gz`
3. The two project folders (AdDistrib and Simulator) will be in the `StuartKregor_Alexander` directory
4. The files are now ready for running or development

B.2.2 Test Run

You can test the simulator by navigating to the `Simulator` directory and running the following command: `bash ./run.sh -file test/adverts/case.xml`
If you see errors in the terminal beginning with 'ClassNotFoundException', the project may need to be built.

B.2.3 Importing Eclipse Projects

1. Open the Eclipse IDE.
2. Import the project into the IDE:
   `File > Import > ''Existing Projects into Workspace''`
3. Navigate to the directory where the files were extracted
4. Ensure the 'Search for nested projects' box is UNTICKED
5. Optionally, select 'Copy projects into workspace', this is personal preference
6. Click 'Finish' to import the projects

The referenced libraries for each project should already be set up. If they are not, simply add all of the JAR files in the `lib` directory to the build path.

B.3 Compiling and Running the projects

Both projects are built with Apache Ant, so you must ensure that Eclipse itself does not try to build the project.
To do this, click on the 'Project' menu, and UNTICK 'Build Automatically'.

The AdDistrib project depends on some of the same libraries as Urbansim, see section B.1.1. These dependencies are included in both projects in order to keep the build path simple. Technically speaking, the AdDistrib project is a dependency of Urbansim, and the `AdDistrib.jar` file must be available on the classpath. The Ant build for AdDistrib should do this automatically.
To compile the AdDistrib project and prepare Urbansim to run:

1. Locate the `build.xml` file for the AdDistrib project.
2. Open the file and find the `urbansim.project.dir` setting. Set this to the location of the Simulator project folder.
3. Right-click on the project and highlight 'Run As'
4. From the popup menu, select the LOWEST Ant build option, it should read 'Ant Build...' Selecting this item allows you to select the build targets.
5. The only targets required should be 'build-project' and 'copy-jar-to-sim'.
   The Ant file will automatically compile all of the Java source for AdDistrib and copy its own dependencies, as well as the JAR file into the `Simulator/jar/lib` directory.
   This will also configure Urbansim to launch with a GUI.
6. Before launching, the `log4j2.xml` will need to be updated with the correct logging directory.
7. Open the `log4j2.xml` file and correct any line that begins with '<File name='. The 'filename' property should point to the 'deviceOutput' directory within the test directory being used.
8. The simulation can now be run as instructed in section B.2.2.

If the simulation should be run without a GUI (headless), simply UNTICK the 'copy-jar-to-sim' target and TICK the 'copy-jar-to-sim-headless' target.
There is a script in the Urbansim project root named `runHeadless.sh` containing instructions on the parameters required.

B.4 Files

B.4.1 Configuration Files

The configuration files for both Urbansim and this project can be found under the `test` directory. This project uses the Urbansim configuration files to set its own parameters.

agentData/busStop@non-display@0.xml

```
<?xml version="1.0"?>
<agent>
    <agentSpecifcData />
    <adverts>
        <advert name="lejkgrrifcwwhekhdcvvmfeoynnopwyjzfunsovojqsttantbgi"
            radius="609.0" maxBid="3.380432272149335"
```

```xml
            budget="30.0" releaseTime="200" expiryDate="2757" />
        <advert name="xlvjslibmigzvdptgdxosuhbbsevjexqdunsvdfpglxh"
            radius="219.0" maxBid="3.752078260764854"
            budget="30.0" releaseTime="167" expiryDate="194" />
    </adverts>
</agent>
```

agents/display.xml

```xml
<?xml version="1.0"?>
<display>
  <class name="adDistrib.devices.Device"/>
  <simulationSettings
    useSmartMechanism="true"
    display="true"
    billingRecordPropagation="INTERNET"
    auctionPeriod="300"
    displaySlots="4"
    auctionReservePrice="0.5"
    billingServerLocation="1320.89,938.44"
    simulationId="111"
    maxAds="40"
    maxBillingRecords="100"
    />
</display>
```

agents/non-display.xml

```xml
<?xml version="1.0"?>
<non-display>
  <class name="adDistrib.devices.Device"/>
  <simulationSettings
    useSmartMechanism="true"
    display="false"
    billingRecordPropagation="INTERNET"
    billingServerLocation="1320.89,938.44"
    simulationId="111"
    maxAds="40"
    maxBillingRecords="100"
    />
</non-display>
```

log4j2.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
  <Appenders>
    <File name="Adverts" filename="/Users/easkay/Documents/workspace/Simulator/
    test/adverts/deviceOutput/111/adverts.json">
      <PatternLayout pattern="%m%n" />
    </File>
    <File name="BillingRecords" filename="/Users/easkay/Documents/workspace/
    Simulator/test/adverts/deviceOutput/111/billingRecords.json">
      <PatternLayout pattern="%m%n" />
```

```
      </File>
      <Console name="STDOUT">
        <PatternLayout pattern="%m%n" />
      </Console>
    </Appenders>
    <Loggers>
      <Root level="trace">
        <AppenderRef ref="STDOUT"/>
      </Root>
      <Logger name="adverts" level="trace" additivity="false">
        <AppenderRef ref="Adverts" />
      </Logger>
      <Logger name="billingRecords" level="trace" additivity="false">
        <AppenderRef ref="BillingRecords" />
      </Logger>
    </Loggers>
  </Configuration>
```

B.4.2 Source Files

These files are found under the AdDistrib directory.

Filepath	Description
bin/	Contains compiled .class binaries
src/adDistrib/auctions/BillingRecord.java	Represents a Billing record. A domain object that forms part of the P2P protocol
src/adDistrib/comparators/DeviceDetailsFreeAds.java	A custom comparator to sort DeviceDetails objects by the number of free advert slots
src/adDistrib/comparators/DeviceDetailsFreeRecords.java	A custom comparator to sort DeviceDetails objects by the number of free billing record slots
src/adDistrib/comparators/DeviceDetailsServerDistance.java	A custom comparator to sort DeviceDetails objects by the distance to the billing server
src/adDistrib/devices/Device.java	The main class that contains the business logic and P2P protocol
src/adDistrib/messaging/Advert.java	Represents an Advert. A domain object that forms part of the P2P protocol
src/adDistrib/messaging/DeviceDetails.java	Represents a Device. A domain object that forms part of the P2P protocol

Filepath	Description
src/adDistrib/messaging/P2PMessage.java	Encapsulates other domain objects as a message with a type and payload
src/adDistrib/parsing/JsonParser.java	A class that implements the Parser interface with a JSON backend, using gson
src/adDistrib/parsing/Parser.java	An interface that provides methods to be implemented by a parser
src/adDistrib/parsing/ParsingFactory.java	Decides which parsing implementation to assign to a device
dist/lib	Contains the libraries required by the JAR file
dist/AdDistrib.jar	The JAR file needed by Urbansim to run the simulation with the adverts platform
lib/commons-codec-1.10.jar	The Apache Commons Codec library, used by BillingRecord
lib/Continuations.jar	A dependency of Urbansim, included to keep build path simple
lib/gson-2.3.1.jar	The Google JSON parsing library for Java
lib/log4j-api-2.2.jar	The Apache Log4j 2.2 Interfaces library
lib/log4j-core-2.2.jar	The Apache Log4j 2.2 Implementation Library
lib/mason.jar	A dependency of Urbansim, included to keep build path simple
SQL/fn_setup_settings.sql	A plPgSQL function to set up an automatic simulation in the database
SQL/fn_summarise_results.sql	A plPgSQL function to process the raw records from each simulation

Filepath	Description
SQL/StructureCreation.sql	The SQL to create all of the required database structure to support automatic simulations
SQL/TestRunAllocation.sql	A small script to allocate batches of simulations
test_runner/agent_specific_data.xml.erb	A ruby ERB template of the Urbansim agentData config file
test_runner/display.xml.erb	A ruby ERB template of the Urbansim agent config file
test_runner/Gemfile	A file that lists ruby gem dependencies
test_runner/json_processor.rb	The script for importing raw JSON files into the database and processing them
test_runner/log4j2.xml.erb	A ruby ERB template of the Log4j2 configuration file
test_runner/models.rb	The file that provides the database logic and global configuration variables
test_runner/non-display.xml.erb	A ruby ERB template of the Urbansim agent config file
test_runner/sim_control.rb	The script for controlling the automatic simulation on a worker VM
test_runner/simulation_script	A bash script used to invoke the ruby sim_control script
test_runner/xml_writer.rb	The script for generating the XML config files. Contains constants
build.xml	The Ant build file for the project
makefile	A file used to simplify commits and backups
results_processing.R	An RStudio project file used to create the plots

B.5 Bug reports

There are two known bugs with the system. The first is that adverts are able to spend more than their allocated budget. The second is that non-display devices are running auctions when they should not be. The latter is a simple fix which can be added to `checkNextAuction()`.

Appendix C

Evaluation Process continued

The following process is used to aucomatically run test simulations.

The Ruby script connects to the database and retrieves the next simulation job for the current worker, along with the simulation configuration and parameters. Next, the necessary configuration files are generated, including the Log4j XML file, device-specific files for advert injection along with generic and global configuration files that configure the protocol and environment for the simulation. These values are retrieved directly from the database, so each simulation is configurable from the same location.

Ruby then invokes the simulation under a timeout to ensure that if the simulation fails, it does not run indefinitely. Any output that is sent to stderr by the simulation is collected in a named pipe for review later. If there is an error, the process times out and the script exits with a failure code, preventing the worker VM from restarting. This preserves any output files from all scripts for debugging. If the simulation times out, it is marked in the database as 'failed', along with any messages sent to stderr.

If the simulation finished successfully, another Ruby script is called to process the JSON results. This script was made multi-threaded with the use of the split[1] program. The split program separated each JSON file into four parts, ensuring it only split at the end of a line to avoid invalidating the JSON syntax. Four threads were created to then process these file fragments to take advantage of the multiple cores. This was done because parsing JSON is mostly a CPU-bound operation. The initial results from the JSON files are inserted in batches into a 'buffer' table in the database. Inserting all the results in one block was very slow since there were usually several hundreds of thousands records.

The results in the buffer were then checked for duplicates and inserted into the results table, recording which simulation run they correspond to. Once any duplicates have been removed, the results are summarised into separate tables according to what attributes are being measured. The records from the buffer tables are then deleted, however this caused issues with the database server in terms of performance and storage space. It was discovered that although the records were deleted, the database had only marked them for deletion, meaning that the records still existed on disk. As a result, the database server quickly consumed storage space, as each simulation can easily produce over one million records. A fix for this problem was created which involved pausing the simulation queue to truncate[2] the tables and reclaim the disk space in turn speeding

[1] `http://man7.org/linux/man-pages/man1/split.1.html`

[2] `http://www.postgresql.org/docs/9.4/static/sql-truncate.html`

up the result import process.

The raw results files were then deleted from the worker VM to avoid sprawling and using excessive amounts of storage. Once the results were processed, the Ruby script exits and its code was checked by the shell script. If the entire simulation process was successful, the machine restarted to clean out the RAM and collect the next simulation.

Appendix D

Future Work continued

D.1 Message sending and object serialisation

Although this topic relates more to physical implementations of peer-to-peer systems, it might be interesting to see whether any benefit resulted from using a more efficient, proprietary message format, as opposed to a textual data storage format such as JSON.

Arguably this may also give benefits to the simulated system, since there would be less text to process in the logged records.

D.2 Use of simulator features

As mentioned, there are some simulator features which remain unused. Making use of these features, as suggested in section D.2 would improve the realism of the system and provide a more accurate indication as to the real-world feasibility of such a system.

D.3 Device variety and variable investigation

During the evaluation, only two parameters were varied and two metrics were used to observe the effects of the parameters on the system.

It would be beneficial to explore the impact of other parameters and variables on the system's behaviour to ascertain which parameters are the more significant among those available.

In addition, various different device types with different features could be set up and tested, with particular regard to the number of display slots per device.

D.4 Auctions

Currently, this simulation implements the GSP auction in order to select adverts for display. Given the amount of literature surrounding online advert auctions, it would be beneficial to explore the effects of the auction type on the network and offer the possibility of using different auction types simultaneously.

Furthermore, adverts could be developed to bid interactively and intelligently in auctions on devices, deciding whether the device was worth bidding on, which display slot was available, as some examples.