The Urban Cloud:

The feasibility of using a Cloud Computing infrastructure for Urban Traffic Control Systems

by

Colin Lyons, B.A(mod)

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

September 2009

Declaration

Permission to Lend and/or Copy

Acknowledgments

Firstly, I wish to thank my supervisor Vinny Cahill for his advice and guidance. Secondly, I wish to thank Meghanne Flynn for her encouragement and support throughout the past year. I wish to thank Niall Bolger of Dublin City Corporation Traffic Division for answering all my questions. I wish to thank Niall O' Hara, a summer intern in DSG, who was always there to help with any queries I had. Finally on a less serious note, I wish to thank the Pepsi Corporation for keeping me awake during the late nights and early mornings spent at my computer.

Colin Lyons

University of Dublin, Trinity College
September 2009

The Urban Cloud:

The feasibility of using a Cloud Computing infrastructure for Urban Traffic Control Systems

Colin Lyons, M.Sc.
University of Dublin, Trinity College, 2009

Supervisor: Prof. Vinny Cahill

Emerging Cloud Computing technologies allow for an inexpensive use of mass quantities of storage, bandwidth and computing resources using the pay-per-use model on which it thrives [6]. The adoption of broadband insfrastructure in various forms (such as ADSL, 3G and Fibre) across cities allow for high bandwidth, low latency, high-reliability connections to the internet. Traditional adaptive Urban Traffic Control Systems were designed back in a time when the only form of data transfer were the high latency, low bandwidth DS0 lines.

Combining these ideas and focussing on the more fine-grained Infrastructure as a Service (IaaS) [5] through the use of the Amazon Web Services (AWS) platform [21], this dissertation intends to make the case that the implementation of Urban Traffic Control Systems 'on the cloud' is a feasible venture.

More specifically, the design and implementation of a reference architecture, dubbed the Urban Cloud Framework along with the placement of SCATS [46] atop this framework, demonstrates this feasibility.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

With the emergence of Cloud Computing technologies providing access to cheap storage, bandwidth and computing resources as a service, there is a new platform in which applications can be built to maintain an online presence. The adoption of broadband infrastructure in various forms (such as ADSL, 3G and Fibre) across cities allow for high bandwidth, low latency, high-reliability connections to the internet. Traditional adaptive Urban Traffic Control Systems were designed back in a time when the only form of data transfer were the high latency, low bandwidth DS0 lines.

Combining these ideas, the question arises as to whether implementation of a control infrastructure 'on the cloud' is possible and feasible. While several interesting hurdles exist, potential adaptations are readily visible. The successful implementation of such a system would lend itself to all the cities in a country (such as Ireland) pooling their traffic control resources together into the cloud allowing for large amounts of data to be aggregated at one source online. This could prove very helpful to Urban Traffic Control Reasearchers as they will have access to large amounts of archived and live traffic data which they can in turn use.

As well, having a more unified model of Urban Traffic Control Systems would allow for cities to be controlled by a varying set of algorithms which would most suit the current traffic conditions. In an attempt to move towards this idea, the implementation of a generic framework for Urban Traffic Control Systems 'on the cloud' is necessary as well as testing the feasibility of one of the most widely used adaptive systems on top of it to make a move towards this idea.

1.2 Road-map

This dissertation consists of six additional chapters beyond this one. The following is a brief outline of each one.

1.2.1 Background

Chapter 2 begins with a brief introduction as to how Urban Traffic Control Systems are broken up into their smallest components, and will find the relationships between them. By having an understanding of them, we can begin to figure out how these components should be modelled and their potential use for the monitoring and control of UTCs. The second section looks into the definition of Cloud Computing, details some of the history on how the phrase was coined and the definition's evolution in both an academic and commercial sense. This is followed by a look at the architecture of Cloud Computing, detailing its split into three tiers: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). These allow us to formulate an idea of which tier is the most useful for design and implementation.

1.2.2 Urban Traffic Control Systems and Cloud Technologies

Chapter 3 focuses mainly on two systems: SCATS and AWS. The first section examines the construction of the Sydney Coordinated Adaptive Traffic System (or SCATS for short), then moves on to the ways in which SCATS gets data from the street, the operation of the SCATS algorithm and a look at its three tier architecture and the challenges it poses to implementation. After which this section looks at the SCATS algorithms themselves, namely Original Volume (OV), Degree of Saturation (DS) and Reconstituted Volume (VK), detailing how they are smoothed. This section ends with the comparision of another popular adaptive traffic system, SCOOT.

The second section looks into how Amazon Web Services (AWS) work. It examines each of its components, such as: the processing array known as Elastic Computing Cloud (EC2), how storage is done via the Simple Storage Service (S3), how applications can be designed to be functionally scalable using the distributed SimpleDB, how Amazon offers an Event Queue with their Simple Queue Service (SQS) and lastly a look at how seriously Amazon take the security of their cloud. This section will end with a comparison of AWS with other technologies such as Google's App Engine and Microsoft's Azure platform.

1.2.3 Design

Chapter 4 is split into three sub-sections. The first is an introduction, which attempts to establish feasible evaluation criteria for this context. Following this, it briefly states why the VISSIM multi-model simulator and Amazon Web Services (AWS) were chosen over others.

The second section details a list of requirements whereby an implementation of the Urban Cloud Framework architecture can be made by gathering Functional and Technical requirements for each of the three components. These components are the Vissim Component, the Junction Controller (JC) and the Main Controller (MC).

The third section deals with planning out exactly what the architecture will look like for the UCF, its functional and technical architecture, include a look the tools that will be used to implement.

1.2.4 Implementation

Chapter 5 is split into three sub-sections. The first deals with the Urban Cloud Framework itself. Each of its elements such as UTCElements and UTCController will give the process by which they were written.

The second section deals with the implementation of the Vissim Component, each of its elements and how they work together. This shows that the control of traffic using only the COM interface is possible, over the more traditional method of having to write a C++ controller DLL which is then embedded later on.

The third section is the implementation of SCATS on top of the Urban Cloud Framework (UCF). It details challenges of doing so, how exactly it is situated on the cloud, what services it uses and how easy it can be used to implement any controller on the UCF.

1.2.5 Evaluation

Chapter 6 looks at some of the tests that were done to answer the question of whether UTCs are feasible to run from a Cloud Computing infrastructure based on the criteria in the Design section and commenting on each of the results.

1.2.6 Conclusion

Chapter 7 is split into three sections. The first deals with what this dissertation has achieved, the second section looks into who could possibly benefit from this research and the last section looks into what future work could be done to continue this project.

Chapter 2

Background

2.1 Urban Traffic Monitoring and Control

As there can be quite a steep learning curve in the UTC field, it would be best to take the time to clearly define the terminology used in traffic systems. In Figure 2.1 you will see a concise diagram of how a traffic cycle is split up into its components (namely approaches, phases and clearance intervals) for a standard right-angle junction. The definitions of these terms [1] are as follows:

- *Cycle Length:* This is the time needed to complete one full running of all the active phases.
- *Approach:* Is a single road, where traffic comes from it into an intersection.
- *Phase:* This is the movement of a set of approaches during a cycle. There is normally at least two (three if there is a pedestrian crossing) in a cycle.
- *Interval:* A discrete portion of the signal cycle during which the signal indications (pedestrian or vehicle) remain unchanged.
- *Split:* This is the percentage of time of the full cycle length that is dedicated to a particular phase. All splits added together will make up the cycle length.
- *Clearance Interval:* This is the time alloted between phases to allow the junction to clear before starting the next phase.
- *Offset:* This is the relationship between the start or finish of the green phases in successive sets of signals within a coordinated system.

As you can see, Figure 2.1 is an example of a junction comprised of two phases which will allow the flow of traffic from two opposite approaches. The clearance interval is defined to give the first approach time to clear the junction before allowing the flow of traffic from the second approach. The splitting of the cycle into phases allows for the Time-Division Multiplexed flow of traffic from all approaches into the junction while minimising the possiblilty of collision. Obviously, great attention must be given to the generation of phases, in order to avoid causing traffic accidents.

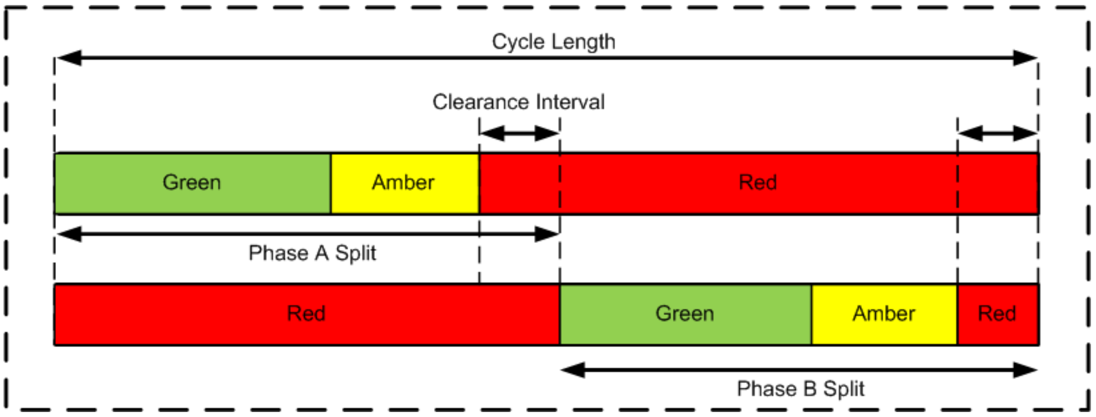


Figure 2.1: How a cycle is made up

2.1.1 Signal Phasing

Taking an analogy from the wireless communications world, to view the vehicles as data, the approaches to a junction being data transmission points and the junction itself being a medium, then phasing can been seen as the medium access control to the system. Phasing allows for the movement (transmission) of vehicles (data) across the junction (medium) while removing the possbility of crashes (interference). This can become very complex as the number of approaches, and lanes contained therein, increase.

The ‘A Phase’ on O’Connell Street (shown in Figure 2.2) is an example of how more complex junctions inherently produce a more complex phase set. This phase allows for the movement of vehicle traffic from both the O’Connell Street Bridge and Westmorland Street approaches. As well, it allows for the flow of pedestrian traffic along Burgh Quay and from the centre of the O’Connell Bridge to Westmorland and D’Olier Street. The left filter from Westmorland Street around to Aston Quay is active in the A phase but not in any other phase, as time has to be given to the pedestrian crossing Aston Quay. The arrows shown in blue are inactive paths which will be active in another phase. There are five different phases in this junction (A - E), which allow for all other paths to get their turn (such as the adjacent approach on Burgh Quay to run while allowing all other pedestrian traffic to be active).

2.1.2 Modelling Traffic

There are two common ways in which traffic can be modelled in simulations. These are in the domain of the microscopic, in which individual agents and their behaviours are modelled explicitly, and the macropscopic in which the modelling is done at a higher level that does not involve single agents, but rather the flows which they would exhibit.

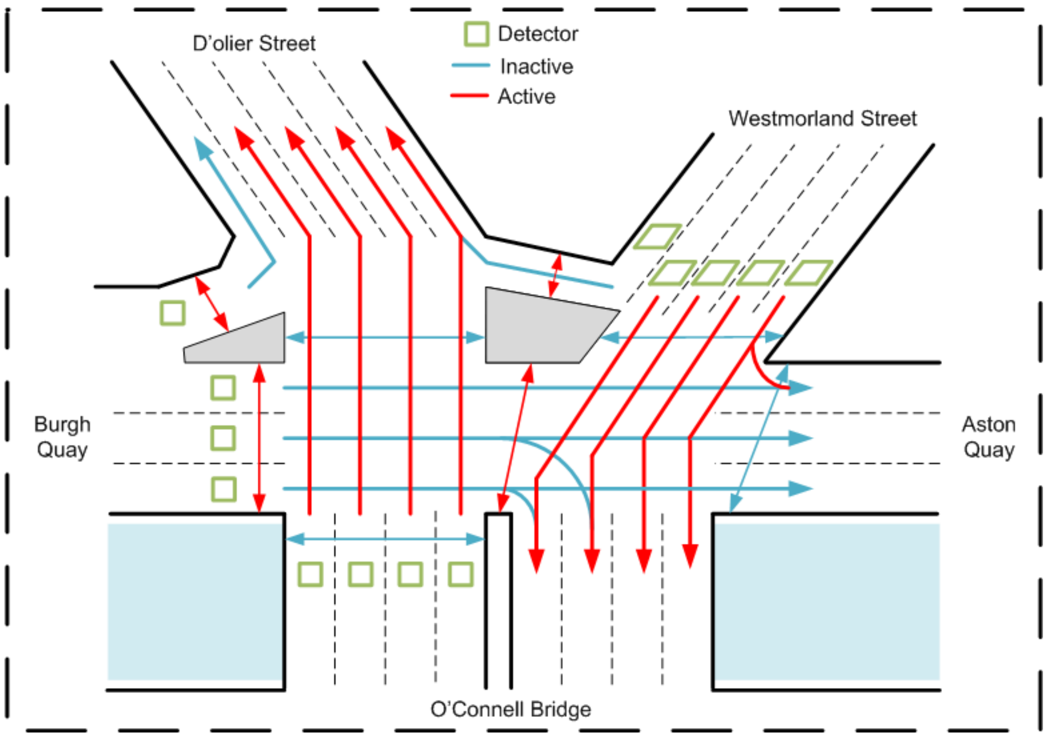


Figure 2.2: An example of the 'A Phase' at the South O'Connell St Bridge Intersection

Macroscopic Simulation

One of the first such uses of the macropscopic model came from M. J. Lighthill et al [4]. In this paper they likened the behaviour of traffic to that of gas kinetics. The model they used was in the domain of fluid dynamics in which the goal was to reach equilibrium, which occurs through the interaction process. Using this concept with some alterations, it resulted in a traffic model which relied on the fluctuation of velocity to achieve this same equilibrium. Although the equations for this can be altered to accommodate different driver behaviours, it lacks the flexibility that microscopic simulation provides.

Microscopic Simulation

Contrasting the macroscopic model, the microscopic model simulates on a per-vehicle basis, which allows for a number of driver behaviors. This is because the vehicle interacts with the world under its own rules which are defined before the simulation. There are two ways in which the grouping of these agents exhibit certain interactions.

Cellular Automata The use of Cellular Automata is one technique for the design and simulation of simple driving rules for vehicles. CAs use discrete, partially-connected cells, which can be in a set of states; for example, a road cell can contain a vehicle and so on.

In the Nagel et al model [2], at each time step vehicles in the system will increase their speed until a maximum velocity is reached. In the case that there is a slower vehicle ahead, the current vehicle will slow itself to avoid a collision. The experiments used to evaulate this show that the behaviour of the vehicles in the CA model was a realistic one, with the result of start-stop waves showing emergent behaviour.

Cognative Multi-Agent Sytsems Another approach, the Cognative Multi-Agent System, is a more advanced approach to traffic simulation than its Cellular Automata counterpart. In the CMAS the agents have the ability to interact with each other and the infrastructure which they are a part of. Each agent has a goal which it is trying to reach, exerting the minimum amount of effort in its part. A vehicle receives its information from the environment via a set of sensors associated with the vehicle. The agent is then capable of making decisions using the sensors combined with set rules to select an appropriate action. The Dia model [3] used the behaviour of real drivers in their implementation of a CMAS to model drivers' responses to having real-time travel information as a factor in their decisions. Based on a survey of a congested traffic corridor, the selection of route and time of departure were recorded by the study and the results used to formulate a set of agents. The use of these agents could then be used to measure the effect of different information systems simulated for that particular corridor.

2.2 Cloud Computing

Cloud Computing has no exact definition. As a starting point we can first gain perspective from the academics who are working on the current state of the art and then move on to people in the business world who have their own ideas about this new platform.

In a June 2009 overview of Cloud Computing written for the ACM queue [6], they define Cloud Computing as follows:

> *"Cloud computing is about moving services, computation and/or datafor-cost and business advantageoff-site to an internal or external, location-transparent, centralized facility or contractor. By making data available in the cloud, it can be more easily and ubiquitously accessed, often at much lower cost, increasing its value by enabling opportunities for enhanced collaboration, integration, and analysis on a shared common platform."*

In the quote above, the cloud is a service which looks centralised to the observer outside the system, but internally the system is a complex set of services which which allow universal access to a companies service. It also plays on the utility computing aspect, in which a user will pay less for an overall service to use only the resources they need or only at certain time. They divide the Cloud Computing infrastructure into three elements: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). For a more in depth discussion on these, see Section 2.2.2. There are obviously a number of enabling factors which have contributed to the success of Cloud Computing. These are pre-cursor technologies such as:

- Inexpensive storage and CPU power.
- Broadband widely available to the public.
- Commercial Grade Virtualisation.
- Technologies such as Software-Oriented Architectures, HTML, Ajax and CSS.

These factors play a significant role in allowing large companies (such as Google, Amazon, IBM and Microsoft) to put together the cloud for use by small to mid-range companies.

In Rajkumar Buyya's view [8], the recent advances in Information and Communications Technology are leading to the vision that computing will one day become the 5th Utility (next to Water, Electricity, Gas and Telephone). In his view, it is a mistake to view Cloud Computing platforms as being the combination of grid computing and clustering. He sees Cloud Computing as the next generation of data centres which work together with technologies like Service Oriented Architectures and Virtualisation to provide a service that is much more than their predecessors. As well, they have provisions for business-oriented approaches to computing such as Service Level Agreements (SLA's) and the concept that everything is managed for you as part and parcel of the service.

It can be very easy to throw around buzzwords when trying to define a concept which is in its infancy. In gaining industry perspective, *The Cloud Computing Journal* ran an article in January 2009 documenting the definition of Cloud Computing from twenty-one industry experts [7]. There are some very interesting definitions presented, but three in particular that seem particularly apt for consideration in this paper.

Blizzard's Jeff Kaplan describes Cloud Computing as, "A broad array of web-based services aimed at allowing users to obtain a wide range of functional capabilities on a 'pay-as-you-go' basis that previously required tremendous hardware/software investments and professional skills to acquire. Cloud computing is the realization of the earlier ideals of utility computing without the technical complexities or complicated deployment worries." In this Kaplan captures the enterprise aspect of Cloud Computing. The cloud works the same as a public utility such as electricity, allowing the customer to use varying amounts in a fixed term, which the customer will then pay at the end of the term. This is the monitoring and accounting side of the cloud. He also talks about the usability of such as service, everything is pre-managed and put in place to make the process of building web applications easier.

Another interesting definition is from the CEO of Stanford Technology Group and Plumtree Software, Kirill Sheynkman. In his view, the cloud focuses on,"Making the hardware layer consumable as on-demand compute and storage capacity. This is an important first step, but for companies to harness the power of the cloud, complete application infrastructure needs to be easily configured, deployed, dynamically-scaled and managed in these virtualized hardware environments." His argument is more to do with the grid being split into individual resources that can be consumed seperately, this is akin to the infrastruture as a service aspect of the cloud. He also brings the scalability aspect into play on processing and storage, as they are always seen as 'on demand'.

Last, but certainly not least, is some perspective from the CTO & Founder of the cloud management platform RightScale, Thorsten von Eicken. He very simply states, “Most computer savvy folks actually have a pretty good idea of what the term ‘cloud computing’ means: outsourced, pay-as-you-go, on-demand, somewhere in the Internet, etc.” He covers the bases of the other definitions with the addition of location transparency. This brings us back to the original cloud analogy for the internet; it doesn’t matter where your application is on the cloud, it just matters that it is there.

There are some trade-offs for companies considering using a Cloud Computing infrastructure. The ACM queue [6] describes the capital expenditure versus operational costs dilemma which most companies must think about when building a web application. Traditionally, a company would buy a number of servers to meet their expected peak demands which would accrue substantial capital expenditure (expecially for start-up companies). Cloud Computing can address this problem by tailoring a company’s costs directly to the amount of resources they are using, as well as cutting the costs of a system administration team needed to keep their own servers running. Essentially, by using a cloud infrastructure the initial expenditure costs are then moved to become operational costs further into the future. A start-up taking advantage of this can quite possibly put more money into the product/service which they intend to sell and be able to track their expenditure.

2.2.1 History

In order to build an accurate understanding of Cloud Computing’s nature, we must understand the concept’s background. Many opinions differ on the origin of Cloud Computing.

Cloud Computing is closely aligned with the term Utility Computing which, “Is the packaging of computing resources, such as computation and storage, as a metered service similar to a traditional public utility” [9]. In 1961, Turing Award winning computer scientist John McCarthy suggested that computing should be viewed as a public utility, much like the electric grid or water [10]. In McCarthy’s vision, the user consumes resources as needed; a monitoring and billing system would be attached. This view coincides with the current understanding of Cloud Computing.

The first known usage of the term ‘Cloud Computing’ is by NetCentric, a company whose goal was to help in the creation/adoption of open source projects [11]. Or in other words to, “Help manage the world’s efforts for co-ordinated response to humanitarian and community projects”. NetCentric attempted to trademark Cloud Computing in May of 1997 under the patent serial number 75291765 [12]. They later abandoned the application in April 1999.

The term again emerges in an April 2001 *New York Times* article by John Markoff documenting software designer David Winer’s negative thoughts toward Hailstorm [13]. Hailstorm, Microsoft’s new platform at the time, was based on their .Net framework. In the article Markoff observes, “For Microsoft, the idea behind .Net is software programs that do not reside on any one computer but instead exist in the ‘cloud’ of computers that make up the Internet.”

August 2006 saw the first high-profile figure to use Cloud Computing in any context, when Google’s Eric Schmidt referred the Software as a Serviced (SaaS) as Cloud Computing at a search engines strategy conference. Schmidt noted, “It starts with the premise that the data services and architecture should be on servers. We call it cloud computing they should be in a ’cloud’ somewhere.” [14]

In a resultant cloud of media criticism, Dell also attempted to submit for the trademark to Cloud Computing in early 2007. Dell responded by saying that they were not making exclusive claim to the term and instead wanted to, “protect the combined term” [12].

As of publishing this paper (August 2009), the current owner of the trademark Cloud Computing without Compromise is utility computing magnate, 3Tera [15]. 3Tera specialises in the bulk of areas which Cloud Computing addresses, such as: clustering, load balancing, virtualisation and networking.

2.2.2 Architecture

By definition Cloud Computing is not a single service, but rather a host of different services which are layered on top of each other and work together [16] [5], this is known as Everything as A Service (EaaS). EaaS is defined as the ability to a use various set of fine-grained services available across a network. It is split into three tiers, known as ‘Infrastructure as a Service (IaaS)’, ‘Platform as a Service (PaaS)’ and ‘Software as a Service (SaaS)’.

Infrastructure as a Service (IaaS)

The first layer, Infrastructure as a Service is exactly as its nomenclature suggests, the delivery of various resources in computer systems as a service. Rather than a customer buying a single server in a hosting company (which will have small amounts of resources like storage, memory, processing power and bandwidth), with Cloud Computing the customers can buy exactly what they need.

For example, if you need to run a distributed ray-tracing program, you will need a lot of processing power and memory but practically zero storage. In this case it would not make sense to buy a grid of machines to do the task as it would be a waste of money. Instead we could use IaaS to abstract us away from the hardware and split each resource into units that can be sold in units.

The billing to the customer is done the same what you would pay for a public utility like electricity, you use as many units as you need in a given period and pay for it at the end of a contract term.

Some of the key characteristics of IaaS include:

- Dynamic scaling of services as required.
- Some form of SLA (Service-Level Agreement.)
- Variable cost in billing with a fixed price-per-unit of resource.
- Multiple Users on the same physical hardware.
- Enterprise Grade to allow mid-size companies to reap the benefits of aggregated resources.

This is obviously not enough to provide a platform capable of servicing the needs of the business world. There needs to be an abstraction above this to provide services for developing on the cloud. An example of a Cloud Computing infrastructure that takes this approach is Amazon Web Services. In this case the user has to explicitly deals with how the topology of their infrastructure is made up, when it should scale up and back and define a set of limits to how many computational units should be used to get the job done. This is a fantastic platform for anything needing vast, but variable amounts of computation in order to acheive their desired goal (such as simulations).

Platform as a Service (PaaS)

The second layer of the three, Platform as a Service necessitates the development and deployment of applications onto the cloud without needing to consider how they fit onto the hardware on the lower layers [17]. This takes the buying and management of the computing resources out of the equation for the customer and thus makes the cloud transparent to applications. On the developers' side, it provides a service to design, develop, test, deploy and host their web application entirely on the cloud, without the need for downloading or installing any special dev tools.

Key characteristics of PaaS include:

- An environment for development, testing, integration, deployment and hosting.
- Integration with databases and webservices.
- Support for collaborative development.
- All IaaS requirements.

In adopting a cloud platform, there are both positive and negative factors to consider.

The collaborative aspect of implementing a cloud platform attracts many developers because it allows decrentralized, multi-national users to develop an application in a central location. As well, implementers see an overall savings in time and money by having the service provider handle programming aspects. This includes such difficulties as security and providing an easy-to-use programming model, which abstracts the developer from the hardware.

However, the lack of interoperability between vendors lends itself to the inherent problem of lock-in. If we wish to develop an application with the Amazon EC2 cloud, we would have to take the effort to port the application to the Flexiscale cloud. This can be a large deterrent for companies to avail of services from less established companies like GoGrid. The shutting down of GoGrid would add significant and immediate costs to a web application as it will need to be ported to another cloud infrastructure.

In Jon Brodkin's article, "Cloud Interoperability Remains Wispy, but Progress Being Made", he documents the current state of interoperability between some of the major players of the cloud computing world [18]. Describing the current efforts as "lip service", he goes on to describe the current progress and cooperation between major vendors. The two open standards which are currently being proposed are the DMTF's Open Cloud Standards Incubator [19] and The Open

Cloud Manifesto [20]. They have already gained wide support by companies such as AMD, Cisco, Citrix, EMC, HP, IBM, Intel, Microsoft, Novell, Red Hat, Savvis, Sun Microsystems and VMware.

As well, there will always been a limit to the rate of growth of a web application in terms of both scalability and complexity of features. Complex features will most likely be difficult to integrate with the platform and as a result much reasearch has been put in to making applications functionally scalable.

Two examples of Cloud Computing infrastructures that make use of the PaaS model are the Google App Engine and Microsoft's Azure platform. They allow the user to develop on their platform without needing knowledge of the infrastructure below. The delivery of hardware, computing and storage resources are handled by the IaaS below.

Software as a Service (SaaS)

The highest of the three tiers is the Software as a Service layer. These are services which allow the user to access software over a network through the use of web-services. An early approach to the same problems were through the use of Application Service Providers (ASPs) [16]. These ASPs would provide a method for subscribing to the service and would then bill for the use of the services over a contract period.

Two examples of the use of SaaS are products such as the Facebook Application Framework and Google Maps. In the case of Google Maps, they provide an API in which the user can create a new instance of a map and have the ability to embed custom data in it by using Google's API. Of course, not all SaaS applications are on a pay-as-you-go paradigm as Google Maps suggests, they rely on a different source of revenue. The Facebook API works on the same principle, where the API itself is downloaded to the user's webserver. The application is then built on that webserver using the API that will make the proper calls to the application and ensure the application conforms to the specifications set out by the Facebook SLA.

Chapter 3

Urban Traffic Control Systems and Cloud Technologies

3.1 SCATS

The Sydney Co-ordinated Adaptive Traffic System (SCATS) is a widely used system developed by the Department of Main Road, New South Wales, Australia in the early 1970s. Although an aging system, SCATS is still active and growing in many parts of the world. Widely used in Australia, the system was originally implemented in Sydney, with Melbourne following shortly in 1982. Other cities of the world using SCATS include: Dublin (Ireland), Tehran (Iran), Hong Kong (China), Gdynia (Poland) and currently being adapted by Atlanta (USA). The SCATS system was originally designed to combat the need for an adaptive UTCS to co-ordinate the majority of Sydneys traffic signals.

Dublin currently posesses 672 SCATS enabled intersections controlled by five regional computers separated into City Centre, North City, South City, West City and Other Areas. Within these regions operate about 160 traffic cameras used to give operators a bird's eye view of the current state of any junction in the system. As of May 2009, Dublin City Council decided to upgrade the system again to 750 intersections by awarding the contract to Aldridge Traffic Controllers. Along with the upgrade, they will also provide maintenance on all currently deployed controllers [44].

3.1.1 Sensors

There are two main methods which SCATS uses to measure traffic data in the real world: the use of induction loops under the road and traffic cameras, which are both used to count vehicles.

Induction Loops

The induction loop is an electromagnetic device which works on the same principle as a magnet being moved across a coiled wire, which will result in a current being induced by the changing magnetic field. As a vehicle drives over the induction loop, the changing magnetic field induces

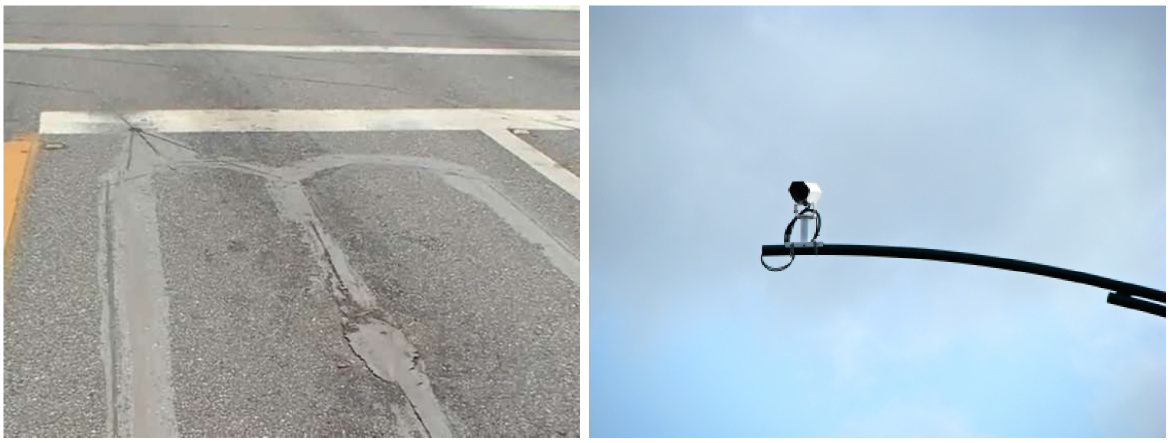


Figure 3.1: An induction loop (left) and traffic camera (right)

a current which is sent back to the junction controller to be processed. This is quite a diverse piece of equipment as it can be used to count vehicles, as well as the gaps between vehicles and can also give data regarding the size of the vehicle (differentiating a motorcycle or bus) over it.

Traffic Cameras

In Dublin, where the surface of the road is unstable or unsuitable fo the use of induction loops, traffic cameras such as the Traficon [47] are used. These cameras are typically placed at a high point above the junction, then making use of computer vision techniques to split the image into boxes which represent the lane locations. With this method, each vehicle which passes through the vision box is counted as well as the time delay between them. The camera has also proven superior than the induction loop in vehicle detection.

3.1.2 SCATS Operation

The SCATS system uses three parameters to control traffic in its domain: cycle time, phase split and offset [43] [46] [1]. The control provided by SCATS is split into strategic and tactical control.

Strategic control relies on management from Regional Managers by using data collected from the sensors on the set of junctions controlled by it. The Regional Manager then sets the optimum cycle time, phase split and offsets based on the data provided.

Tactical control allows for the management of traffic by Juntion Controllers and allows them to make changes to cycles on a per intersection basis. This allows for phases to be skipped or shortened based on the current traffic demands. Again, this is based on data from sensors at the junction, some of which are strategic. Although the Junction Controllers assert their own form of independent authority, configurations as to how much they can change the normal operation of traffic are overseen by the Regional Manager.

3.1.3 The SCATS Architecture

The SCATS architecture is a three tier architecture split into the Junction Controller (JC), the Regional Manager (RM) and the Control Management System (CMS) [43] [46].

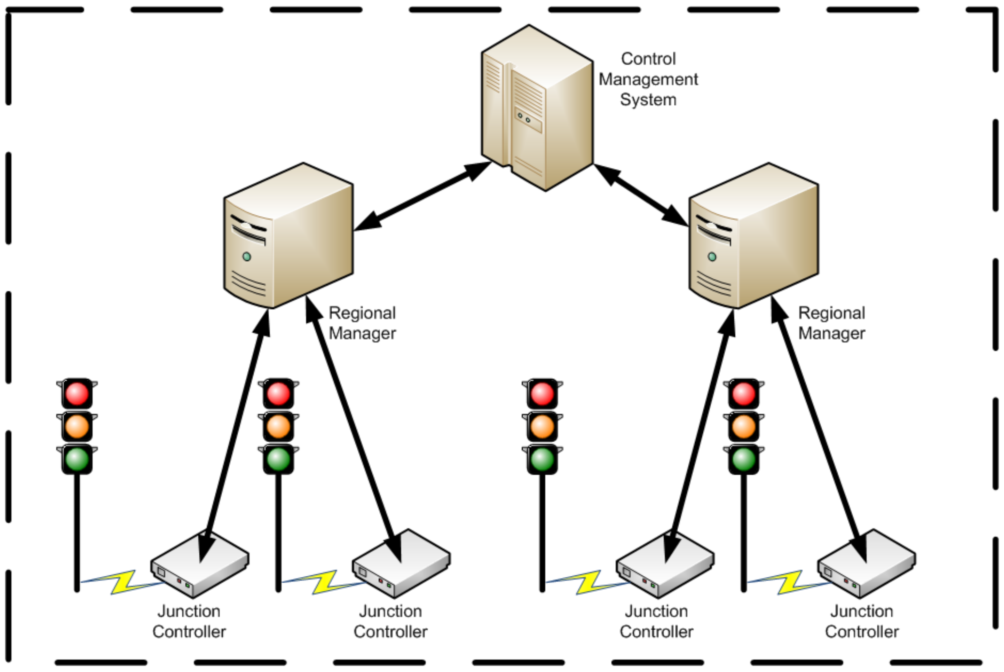


Figure 3.2: The SCATS Architecture

Junction Controller

The first of the three tier structure is the Junction Controller, a small computer which sits at every SCATS enabled junction. The JC is responsible for:

- Collecting, processing and posting data from the sensor to the RM.
- Receiving and implementing updated timing data from the RM on the junction.
- Logging faults such as blown lights or broken pedestrian buttons and posting these when applicable back to the RM.

These controllers can also be grouped into systems and sub-systems. Systems do not interact with each other as they are probably not geographically related. Sub-systems on the other hand, can link together to form systems. They are a system of multiple junctions (in the range of about 1 to 10) which are grouped together based on them being in close proximity to each other. Most of the time, each sub-system will run independently of the next one, but as traffic starts to increase the sub-systems may start to “marry” with sub-systems which are in proximity to each other to form larger sub-systems.

Each sub-system has minimum, maximum and optimum cycles in its own scope. As well, there are four “background green split plans” stored for each available sub-system (for example, a morning rush hour cycle, evening rush hour, pre dawn cycle, etc). Each of these variables are selected depending on what the current flow of traffic is at the time. Located within a sub-system is the entity known as a strategic detector. The strategic detector analyzes which junctions in a sub-system will likely experience large traffic flows at certain times, thus marking them as a critical intersection in the sub-system.

Sub-systems also contain five "background internal offset plans" which determine the offsets between junctions and five "external offset plans" for linking between adjacent sub-systems. All junctions contained in a sub-system are subject to the same common cycle length. As well, for each sub-system to "marry" another, there are up to four plans for when they should do so. The two sub-systems then become their own system and the common cycle length is set to which ever sub-system had the longest cycle before linkage. As for the "divorce" of a sub-system from a system, this is done when the conditions for linkage are no longer met, resulting in the sub-system reverting back to an independent mode.

Regional Manager

The second tier contains the Regional Manager (RM), which can provide control for up to 250 junctions in the current implementation [48]. This is the point where all the leased lines from the JCs meet and where the actual SCATS algorithms are contained. These machines are required to be state of the art in order to handle the hundreds of connections and constant streams of data from the JCs. RM to RM communication can be done via the Control Management system. Connections between the RM and JCs can be implemented using both wired and wireless comms.

For example, in Dublin, depending on the location and importance of the junction they use a DS0 leased line, 3G/GPRS connection using a private Access Point Name (APN) or even use Dublin City Council's own fibre connection. The DS0 connection is the most common of the three for junctions, providing a 64kb/sec connection between the JCU and the RM. The mobile communication connections are for places where it would be otherwise too expensive to run a leased line to a junction, but where 3G/GPRS is freely available.

Control Management System

The top of the three tiers is the Control Management System. This system looks after systems like databasing results, providing a graphical interface to the operators, where faults are logged and system accounting. It also provides a place to collect data from the system for further inspection. This service is typically very secure and allows external access to provide maintenance contractors with information about faults.

3.1.4 Sensing SCATS Data

In a given junction, data sensed from the detectors is collected by the JC and sent on the regional computer at the end of every cycle. There are two different types of data needed from the detectors for the SCATS algorithms to work:

1. *Vehicle Gaps:* The number of gaps occuring between vehicles (that is, the number of times nothing is detected) and the total non-occupancy time during a particular lanes green time. Non-occupancy is defined as "the amount of time during a lane's green time that the detector has no vehicles travelling over it", which is measured in Seconds(s).
2. *Unused Phase Time:* This is the phase time for a lane plus any unused phase time. Phase time is unused, for example when there are no vehicles passing through a junction during a phase when a pedestrian pushes the button on the crossing. In this case, the JC will end the phase prematurely.

3.1.5 The SCATS Algorithms

There are three algorithms defined by Lowrie [46] [45] which SCATS needs for each lane in the system to function. They are split up into Original Volume (OV), Degree of Saturation (DS) and Reconstituted Volume (VK).

Original Volume (OV)

The Original Volume (OV) is the number of vehicles n that have passed over the detector in one traffic cycle. This calculation is done by adding the number of spaces occuring during the green time of a phase and adding *1* for the car currently over the detector. OV will be needed to calculate the other formulae. The equation for calculating OV:

$$\boxed{OV = n + 1}$$

Degree of Saturation (DS)

Lowrie defines the degree of saturation (DS) as the "ratio of effectively used green time to the total available green time" [46]. The DS (measured in percent) takes the green time for the phase (measured in seconds) $g\prime$ and divides it by the phase time given g plus any remaining phase time r. $g\prime$ is calculated as the the green time g minus the result of subtracting the Total Non-Occupancy time T from the space-time associated with each vehicle t for each vehicle n.

$$\boxed{\frac{DS = NF[g - (T - t.n)]}{g + r}} \equiv \boxed{\frac{DS = NF(g\prime)}{g + r}}$$

The space-time at Maximum Flow (MF) is t. MF is defined as the the greatest number of vehicles which have been recorded passing over the detector during the phases green time (measures in vehicles per hour). This can be calculated using by also using KP, which is defined as the average occupancy recorded per lane during an MF (measured in percent). This produces the following equation:

$$\boxed{t = \frac{3600}{MF} - \frac{KP}{100}}$$

DS Results With the DS calculated for each lane, we can come to two conclusions about its result:

- A DS value less than 1 will show that current traffic flow for the current cycle isn't saturated.
- A DS value over 1 will show that the current traffic flow for the current cycle is over-saturated. The cause of a DS over 1 is lack of traffic flow over the detector on the stop line due to congestion on the outgoing links from the junction.

The SCATS 6 Function Description [48] states that the ideal Degree of Satuation is around **0.9** and the system will therefore vary cycle length in the range of 20 to 240 seconds to achieve this.

Reconstituted Volume (VK)

The Reconstituted Volume is a measurement of how many cars should have passed over the detector at the stop line for the current DS, which is measured in vehicles. This is defined by the formula:

$$\boxed{VK = \frac{DS.g.MF}{3600}}$$

The ratio of Original Volume (OV) to VK is a useful metric for finding the ratio of excpected throughput for the junction versus the actual throughput for a cycle. Normal operation of the system will suggest that this ratio carries a result of 1. A ratio greater than 1 will likely show an indication that congestion is occuring.

3.1.6 Smoothing SCATS Data

With SCATS trying to update itself on a cycle-by-cycle basis, it can be prone to significant variations in the three metrics, OV, DS and VK. To combat this, SCATS employs a decreasing weighted averaging mechanism which will take the values from the last three cycles (including the current one, which has just finished). This can be seen in the following formulae:

$$\boxed{AOV = 0.45(OV') + 0.33(OV'') + 0.22(OV''')}$$

$$\boxed{ADS = 0.45(DS') + 0.33(DS'') + 0.22(DS''')}$$

$$\boxed{AVK = 0.45(VK') + 0.33(VK'') + 0.22(VK''')}$$

3.1.7 Phase Split, Offset and Group Linking

The Phase Split is determined in SCATS by [50] attempting to equalise the DS on critical approaches. The VK for each split plan will be calculated and the plan with the lowest maximum DS will be selected. This is for use with systems that are in the full Masterlink mode.

For systems that are in Isolation Mode, the split is determined by a pre-made series of splits that are activated depending on the time of day.

In each sub-system the offset between junctions will be selected through looking up procedure which is based on relative volumes for each intersection. Sub-systems share the same cycle length and use the offset as a way of maximising traffic throughput over greater areas. Without the offset, traffic would only get to the next junction mid-way into the next cycle. This results in a greater number of stoppages and more congestion. Small variations in offset are allowed on a cycle-by-cycle basis.

As the cycle lengths and offsets are calculated from the critical junction, the linking and unlinking of sub-systems will be done on the basis of whether the current junction has a desirable cycle length which is close to an adjacent sub-system's CL. The decision to link will be based on whether two sub-systems' desired cycle lengths are within a certain threshold, they will link. The decision to unlink will then be based on when their desired cycle lengths exceed this threshold.

3.2 Other UTC Systems

Many UTC systems exist which this dissertation does not focus on, that use methods in the domain of fuzzy logic, reinforcement learning and genetic algorithms to enable traffic control. One such system, which is a popular alternative to SCATS, is the Split Cycle Offset Optimisation Technique (SCOOT).

3.2.1 SCOOT

The Split Cycle Offset Optimisation Technique (SCOOT) is a system developed in the UK in the 1970's [52] and is used in many places in the world such as: London (UK), Bejing (China), Toronto (Canada) and Sao Paulo (Brazil) among others. The SCOOT architecture is organised into a fully centralised model, in which data is passes from the traffic lights directly to a data centre where processing occurs (differentiating it from the three-tier architecture of SCATS). SCOOT also employs [51] a second set of induction loops located anywhere from 50 to 300 metres before the stop-line (unlike SCATS which only utilises the single set on the stop-line).

The second set of detectors provide a count of vehicles coming up to the stop line so the queue length can be determined. This allows for a more up-to-date snapshot of traffic, which enables SCOOT time to communicate between junctions and the control centre. The difference between SCATS and SCOOT in this instance is that SCOOT will have information about traffic flow before cars cross the junction, whereas SCATS only receives information regarding traffic flow after vehicles pass the stop-line. The cycle length of SCOOT can only be varied after 150 seconds, where SCATS can be varied on a cycle-by-cycle basis. SCOOT also only does estimates of DS, whereas SCATS determines accurate measurements of DS at the stop-line.

3.3 Amazon Web Services (AWS)

In early 2006, Amazon started AWS to provide companies with an easy-to-use, reliable, cost-effective, secure and flexible platform providing a cloud computing environment through a set of web services [21]. The key to the platform's success (60,000+ individual customers) lies in providing an immediate service, containing no contracts, remains 'pay as you go' and is completely platform agnostic with a set of well established services. These are: the Elastic Computing Cloud (EC2), CloudWatch, Auto Scaler, Elastic Load Balancer, Simple Storage Service (S3), SimpleDB, Simple Queue Service (SQS), and the Elastic MapReduce service. For a list of prices (as they are probably updated regularly) visit `http://aws.amazon.com/`.

3.3.1 Elastic Computing Cloud (EC2)

The EC2 is a large array of physical hardware displaced around Europe and the USA which Amazon uses to provide IaaS [22][23][25]. EC2 provides a great deal of the backbone features of cloud computing, some of which include:

- *Scalability* - Amazon provides an interface for defining rules for when your application should scale up and back.
- *Location Transparency* - The customer/end-user doesn't need to know where the web application is located on the cloud; this, in fact, will change over time. Amazon provides "Elastic IP Addresses" which tie an IP address to your account, but not to any particular instance. This can have a host of benefits when dealing with multiple instances.
- *Load Balancing* - The Elastic Load Balancer is a free service that will automatically distribute traffic across all available instances seamlessly.
- *Region Specific Instances* - The best course of action is for a company to deploy their application in close proximity for efficiencies sake. Amazon has data-centres in two regions on the globe, Europe and the USA.

From a technical perspective, each of the servers are Linux and Windows based and on top sits an array of virtualised servers which are based on the open source Xen Virtual Machine Monitor [26], which allows the multiple tennants on a single physical machine.

When the user launches another "instance" through the AWS Management Console, an "Amazon Machine Image (AMI)" is booted up somewhere in the cloud and the customer receives root access to it. This is an instance which belongs completely to the user and as such will have full control over every part of it. There are a number of different variants of AMI available to the public, some of the more popular ones are based on Ubuntu, Fedora, Debian and even Microsoft Windows Server 2003.

For the developer it is as simple as downloading an AMI, installing the applications the customer requires for their application (for example Java, Python, MPI) and uploading it back to S3. To make things a simple as possible for the user, Amazon provides a web interface for configuring, launching, terminating and monitoring all of the customers AMIs.

EC2 provides two different types of instances: standard which is built to suit the needs of most business customers and high-CPU instances which are intended for applications of a HPC nature. For a complete list of instance types visit `http://aws.amazon.com/ec2/#instance`.

CloudWatch Amazon CloudWatch is an important web-service provided as a system for monitoring and managing your currently running instances [34]. It is designed to view statistics from both the Amazon Elastic Load Balancer and EC2 itself. CloudWatch operates by collecting raw data from these services and processing the information into a human readible metric format.

Raw data collected from these instances is grouped into what is known as a *measure* with an associated value, unit and a timestamp. Depending on the measure's context, it may also have a *dimension* or *namespace* providing extra information about the current measure (such as the instance ID) if applicable.

For example, from an EC2 instance important measures such as CPU Utilization which is used to get a percentage of CPU time used, DiskWriteBytes which is how many bytes have been written to the instances disk (not S3), DiskReadOps and for Networking information NetworkIn and NetworkOut. Measures are split into one minute chunks, so if ten measures come in during a one minute slot, they will be aggregated into a single chunk.

Once the raw data is processed and aggregated together by a particular unit, it then becomes a *metric*. In the conversion from measure to metric it maintains its dimension, namespace and unit. For each metric, there are different views such as Average, Sum, Minimum and Maximium, which can be queried for any metric / set of metrics.

Using the data provided by Amazon CloudWatch, a developer can adjust the current set of instances to suit the current needs of the application. CloudWatch provides both a Query and SOAP/WSDL API to send statistical operations and recieve results.

Auto Scaling Amazon Auto Scaler is a web service providing the ability to launch, monitor and terminate EC2 instances using triggers defined by the user [35]. It provides both a SOAP/WSDL and Query API to set parameters by which an AutoScalingGroup will be scaled. An AutoScalingGroup is to scale up and down a set of EC2 instances or to stay within a certain range. These groups can only be for instances within a single Availability Zone.

To start an AutoScalingGroup you must first define a launch configuration. Then, define triggers which use metrics from CloudWatch as well as threshold defined by the user which will be used to scale up and down. When a launch configuration is updated it will only appear on new instances launched.

Elastic Load Balancer The Amazon Elastic Load Balancer is an easy-to-use web service (with both a Query and SOAP/WSDL API) for improving the scalability and reliability of your web application [36]. This service allows for the distribution of traffic across the instances you have running. As capacity hits a peak, or the usage of your web application goes down, the ELB will dynamically register or unregister instances from the Load Balancer.

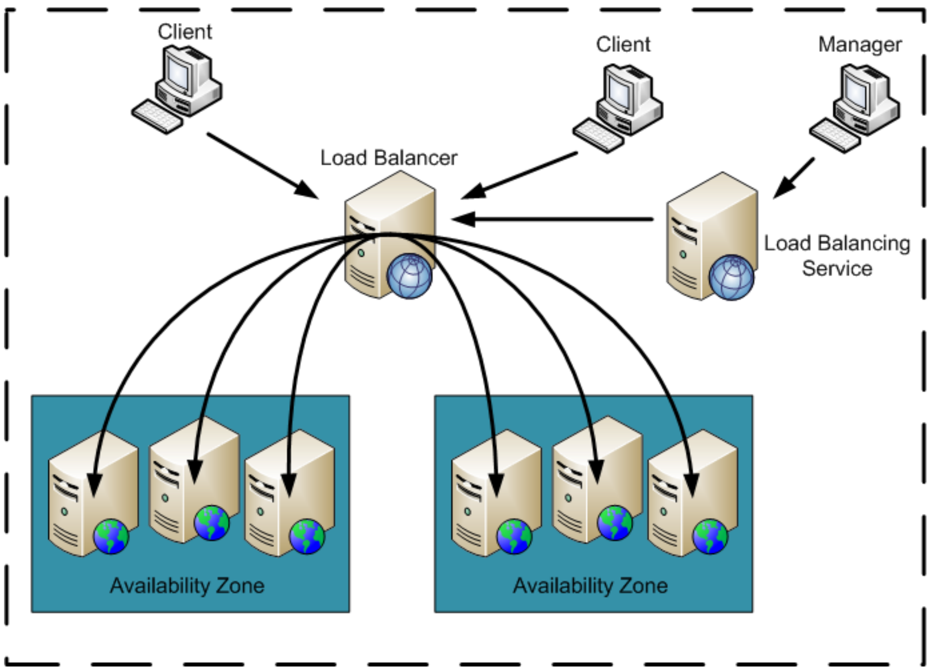


Figure 3.3: The Elastic Load Balancer Architecture

The ELB uses DNS CNAMEs (such as `http://www.example.com`) and a set of ports. When the LoadBalancer is created for your service you will need to register instances with it. As well, the LoadBalancer also provides a "health check" on each instance registered with it, so in the case of problems (such as a crash or hardware fault), the LoadBalancer will stop distributing traffic to that instance and spread it among the remaining ones. This is a great feature when high availability is an absolute requirement for your application.

The LoadBalancer will only distribute traffic across Availability Zones equally. For example, if you have five instances in one zone and two instances in another, each zone will get the same amount of traffic. As a result, Amazon recommends that you keep the number of machines across availability zones as equal as you can.

Figure 3.3 shows an example of the ELB architecture. Clients across the internet will connect to the service via its hostname, which the LoadBalancer will then map to an active instance to serve them. The manager can then make updates to the Load Balancing Service which will propagate updates to the LoadBalancer set-up for the service.

3.3.2 Simple Storage Service (S3)

The S3 is an IaaS put in place to separate storage as a resource for web applications [22][27]. From a PaaS perspective, it provides a web service interface (in the form of SOAP/WSDL/XML or REST) which is used to retrieve and store data of any size at any time.

Each data item is split into "buckets" which are anywhere from one byte up to five Gigabytes. Each item is assigned with a unique key used to reference it at any point. Although it has the standard HTTP interface used in almost every web application, they also provide an alternative BitTorrent interface for content that is more distribution oriented.

As with all the services in the Amazon suite it provides a scalable, fast, inexpensive, reliable, location transparent, fault-tolerant and region-specific design [28]. Requirements unique of the S3 include:

- *Concurrency Control* - Designed so concurrency control is handled and little thought is needed by the customer.
- *Decentralised* - Removes problems with single points of failure and bottlenecks due to scalability.
- *Controlled Parallelism* - Abstracts the customer away from problems related to parallelism.

From the developers' perspective, data in the cloud is accessed almost exactly the same way it would be locally. The current cost for transferring to S3 is 0.18c EUR (for European data centres) or 0.15c (for US data centres). Exempt from this fee are transfers between EC2 and S3 and within EC2.

A *bucket* is simply a container for multiple objects, which are stored in Amazon S3. For example, if we have an *object* named "photos/test.jpg" stored in the testUser bucket, then we can access this object by URL at `http://testUser.s3.amazonaws.com/photos/test.jpg`. Objects are fundamental to S3; they consist of data and metadata. The buckets themselves are not only used for separating data into different sections, they also serve as a unit by which statistics can be gathered on a bucket for usage purposes. A *key* is used to uniquely identify an object in a bucket. Accessing any unique piece of data is as simple as referencing the bucket and the key. In the example above the key for the test.jpg would be "photos/test.jpg". This allows us to logically partition our data much like a Unix file system.

3.3.3 Simple DB

The Simple DB is intended to provide some of the core database functions an enterprise application will need with the added benefits of their cloud infrastructure [29]. Simple DB harnesses the power of both the EC2 and S3 to provide functionality for querying, storing and processing data sets.

Oftentimes, it is very expensive for a small company to lay out their own distributed database, with the main costs being in the design and hiring a dedicated database administrator for maintenance.

SimpleDB uses the same service-oriented architecture approach as all the other services in their suite [30]. Using SimpleDB is a simple as:

- *Building* There are a set of functions at your disposal. You can create, delete and list the current domains. These functions are called CreateDomain, DeleteDomain and List.
- *Retrieving* - The API allows functions for SQL select statements, a function for retrieving records by ID, a Put function for modifying items in a domain and the ability to delete data items. These functions are Select, Get, Put and Delete respectively.

Although a very flexible service there are some hard limits on the use of SimpleDB. Each domain has a maximum size of 10GB. As well, there are constraints on data retrieved by a query. A query has to be achievable within five seconds to be valid and can only return up to 2500 data items at once and can be up to one MB in size.

3.3.4 Simple Queue Service (SQS)

The Amazon Simple Queue Service (SQS) is a distributed queuing system which can quickly make messages to be consumed by another service [37]. The queue is intended to decouple components of a larger system which are separated by data that needs to be processed. Any application in the system can store a message in the queue. The queue provides a reliable, fail-safe unit of data which is sent between services using Query or SOAP/WSDL.

SQS provides the following features:

- *Redundancy built in:* SQS guarantees the delivery of every message at least once, as well as high access availability in sending and receiving messages.
- *Multiple Writers and Readers:* Any component in your system sends and receives multiple messages at once to the queue. SQS ensures a locking mechanism for messages that have already been received by other components to avoid processing the same message multiple times.
- *Configurable Queues:* Each queue is configurable and does not have to be the same as any other.
- *Variable Message Size:* As there is no fixed size on messages (they can be up to eight kb) this allows for a very flexible queue. For messages larger than eight kb, storing a pointer to an S3 or SimpleDB object allows for the processing of very large messages. Alternatively, they can be split over multiple messages.
- *Unlimited Queue Length:* Amazon does not have a hard limit on the number and size of your queues.
- *Access Control:* Amazon allows control over what component can send messages when, and who can receive them.

SQS is overall a very flexible system. However, it still remains a distributed system and thus is constrained by the problems facing them. There are several issues that must be addressed before moving on. SQS makes an effort to preserve the order of messages as they are received to a queue, but the system cannot guarantee correct ordering. Instead, it suggests that an ordering should be placed in the message itself if it is a requirement. As SQS uses a form of replication to provide high redundancy and availability, if a server storing a copy of the message becomes unavailable for a time, then there is a chance that a copy of that same message will be received again by that node. Thus, applications must be designed with idempotency in mind. Also, there is an issue when attempting to consume messages from queues that are not very large (say, under 1000 messages). As the queue only samples a subset of the servers in your queue, there is a chance that no message will be received at all even though there are messages in the queue as a whole. As a result, having a system continuously looking for messages will mean that the queue will eventually sample all available snapshots.

SQS depends on the usage of 3 elements the Queue URL, Message ID and Reciept Handle. The Queue URL is used to address (by URL) the queue you want to use. For example, if we have a queue called 'TestQueue' and our account number is '1111111111' then we can access this queue at `http://queue.amazonaws.com/1111111111/testqueue`. The Message ID is a unique identifier for each message needed in order to delete the message from the queue in older versions. As this is a distirbuted system, there are no guarantees that a received message will be processed. As a result it is necessary to specifically delete a message for it to leave the queue. In the current version, the Reciept Handle is received when you receive a message from the queue; this can then be used to delete the message or change its visibility timeout. The visibility timeout is used to block the message which still remains in the queue to give it enough time to be processed by the component which recieved it. If it doesn't get the Receipt Handle before the timeout, it allows the message to be received again by a different component, and so on.

3.3.5 Elastic Map Reduce

Amazon's Elastic MapReduce allows customers to process large data sets easily on the cloud [31]. It allows the user to use as much processing power as they need without having to worry with any issues to do with scaling. The MapReduce is done using Apache's Hadoop [33] framework which allows the processing of massive data sets across clusters in Java.

It is made up of the following features:

- *Hadoop Processing* - Necessitates processing of Hadoop work flows by starting, configuring and shutting down clusters of EC2 instances as it needs.
- *Multi-Step Job Flow* - A job flow can have more than a single step.
- *Job Flow Monitoring* - You can get real-time information on the status of a flow.

From a more technical point of view [32], the process of using Hadoop for a MapReduce is done in quite an intelligent way. As you can see in Figure 3.4, there are five steps for executing a MapReduce on EC2.

1. Upload the data set to be processed along with the mapping and reducing executables and direct the Elastic MapReduce client to start the job flow.
2. Elastic MapReduce then starts a cluster of EC2 machines which are loaded up with Apache Hadoop.
3. Hadoop executes the uploaded job flow with the data downloaded from S3 onto the EC2 slave instances.
4. The cluster of EC2 instances processes the data and uploads the results to S3.
5. The user is notified when the job is completed and results can be taken from S3. The process is complete.

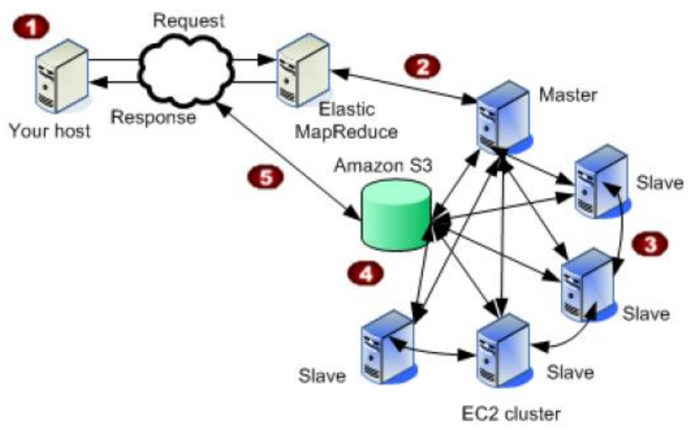


Figure 3.4: The Elastic MapReduce Process [32]

Apache Hadoop works using the MapReduce programming model which divides large amounts of data into smaller more manageable chunks. The master Hadoop node then distributes one of these chunks as well as the job flow executable amoung all the slaves. Each of these slaves run MapReduce on their subset of data and the results are combined from all slaves into a result. The results are put into a bucket of the users choice.

Elastic MapReduce provides an easy-to-use web front end which can be used to create, monitor and delete a set of job flows as well as choose how many instances should be used to process it.

3.3.6 Security

Amazon is a company who is no stranger to having large data centres. In their Security Whitepaper [38], the same experience has been applied to their AWS services. Starting with physical security, their data centres are housed in a low profile centre with military grade beaming. They also have strict access controls put in place to both the inside and internally in their data centres which is controlled by a state of the art security system with an experienced security team. There are at least two tiers (internally) of security for any employee to get to the floors of the server rooms. Visitors must be signed in and asked to show identity at every security station.

Data stored in S3, SimpleDB and EBS is in multiple and logical physical locations. In addition they are protected by keys as well as SSL connections. EC2 security is deemed very important, as each phyiscal machine can be populated by many virtualised instances. Multiple tennants on a single machine pose security problems. To combat this, the host OS may only be accessed by an administrator with their individual SSH key. There are different security levels for each system and every action is logged, audited and routinely inspected.

The Guest OS is completely controlled by the user, in the sense that they are root of their own domain. AWS administrators do not have access to customer machines and customers have the choice of a password based or key based authentication (key being preferred by Amazon) for each of their instances. Each of the instances on a single system are separated via the Xen

Hypervisor, with Amazon being active in the Xen community. Also, EC2 provides a full firewall set-up in “deny all mode” so the user is required to specifically open any ports they want to use. The firewall is not controlled by the host itself, but needs a X.509 cert and key to authorise a change. The hypervisor does not have access to the physical RAM, Disk or Network directly. Each has to pass through a layer of security to ensure no data bleeds into another instance. The instance then uses a virtualised disk which is deleted on a block by block basis when the instance is no longer used to ensure data confidentiality.

All API calls are signed by the users X.509 cert or their Secret Key. API calls can also be encrypyted with TLS (over HTTPS) to maintain confidentiality and consistency on a point to point basis (this is recommended by Amazon).

The internal AWS network provides counters to most traditional security problems such as DDOS, Man in the Middle (MITM) attacks, IP Spoofing and Port Scanning.

3.4 Other Cloud Technologies

Besides the three main contenders out there, there are of course many other companies out there that provide similar services. Some of note are the Google AppEngine, Microsoft Azure, IBM’s Blue cloud, Freescale, Gogrid and 3Tera.

3.4.1 Google AppEngine

The Google AppEngine is a Platform as a Service (PaaS) that allows users to build web applications to run on Google’s infrastructure in a Java or Python environment [39]. It has many of the underlying services that AWS provides, such as IaaS in the form of DataStore [40], which provides the same functions of its rival, the AWS SimpleDB and their JDO which allows for S3 like storage. Although there are a lot of hard quotas, the service is free and you can even pay to use anything above the hard quotas. In Jack Schofield’s article [41], he describes the Google AppEngine as a definite rival to AWS for the business market. It also provides other functions like Memcache, which provides a distributed shared memory cache for applications and URLFetch, which allows fetching resources and communication with other hosts using HTTP securely via HTTPS. They provides a mail service in the form of the Mail API, which allows for the sending mail to users.

3.4.2 Microsoft Azure

Microsoft Azure is another cloud platform entering the market within the last year which boasts the same flexibilities of AWS with the added bonus of being heavily integrated with their .NET platform [42]. They provide a host of services which directly compete with the AWS platform. Again they provide a Platform as a Service (PaaS) rather than the AWS IaaS meaning that you cannot control individual units of computation, storage. Instead you just write you application for their platform and deploy it on their cloud with the lower layer being automatically handled for you.

Chapter 4

Design

Before implementing anything, it is of utmost importance that time is spent gathering requirements and specifications; with these details we can proceed with the system design. The more planning that is put into the project at this stage, the fewer problems will be encountered during the implementation and evaluation stages.

The first concern is summarising what this dissertation is trying to achieve. This paper wishes to examine the feasibility of building an Urban Traffic Control System capable of running on a cloud infrastructure. Of course, before we can move forward we must outline a working definition of feasibility in respect to UTCSs and Cloud Computing.

When thinking in terms of feasibility in this context we are confronted with the problems associated with distributed systems. In order to ensure the implementation is not only possible, but plausible from a usability perspective, we must consider questions such as: Is the system scalable to meet the requirements of even the biggest theoretical UTC? Is the system reliable enough to be used for such a critical application? Does the latency between the JC and the MC a problem for real-time simulation? Is the system tolerant to faults? Can it recover from these? Is the system secure enough to be used and ensure that all data is consistant and confidential? From an economic perspective, does it make financial sense to implement a system like this? These are the terms by which the urban cloud will be judged as feasible.

It is not plausable for this system to be tested in the real world. There exist a multitude of well-constructed traffic simulation environments which allow the return of high fidelity results. For this dissertation, VISSIM multi-model traffic flow simulator will be used, as it has been a proven environment for over 15 years. It also has room for future work as it simulates cyclists, pedestrians and even rikshas, as well as the usual vehicle and public transport. It is capable of carrying out microscopic simulations, which is the more proven of the different types as individual behavior is captured. As well, it has a very extensive system for carrying out evaluation of simulations.

As for choosing a cloud computing infrastructure, the Amazon Web Services (AWS) framework will be used as it gives you more control of individual units of computation and storage and allows the user to define their own rules for scalability, by using EC2's Auto Scaling Groups.

It also has very nice integration with the Visual Studio in C#. The integration provides an easy-to-use development environment. Microsoft Azure and Google App Engine are possible alternatives, but they are PaaS infrastructures rather than IaaS, which would provide less of the fine tuning that is required for this system.

From this, the gathering of requirements and design of the Urban Cloud Framework (UCF) using both VISSIM and AWS can begin.

4.1 UCF Requirements

This sections provides a brief list of all the requirements needed to build the UCF architecture, which are divided into sections which correspond to the sub-components of the system. The first set of requirements apply to the VISSIM component, the second to the Junction Controller (JC) and the third to the Main Controller (MC) component.

As a global requirement, the system must be made in a modular fashion allowing the replacement of parts if required (such as the simulator, or the control algorithm).

4.1.1 VISSIM Requirements

The requirements listed for the VISSIM component are divided into two categories: functional requirements and non-functional requirements. The VISSIM Component is required to build an interface between VISSIM and the UCF, allowing a translation of monitoring and control data between VISSIM and UCF objects.

VISSIM Functional Requirements

1. The VISSIM module must call the simulator's COM Interface at every time step and translate relevant sensing data (such as Detector Loop Data) to UCF objects. These objects (known as UTCElements) must then be usable by the other components in the system.
2. The VISSIM module must be able to implement the changes (such as Changes in Phase Time) in UCF objects on the simulator.

VISSIM Non-Functional Requirements

1. The VISSIM module must be on a machine fast enough to run simulations in at least real-time.

4.1.2 JC Requirements

The requirements listed for the Junction Controller (JC) components are divided into two categories: functional requirements and non-functional requirements.

JC Functional Requirements

1. The JC must be open ended enough to support the adding of custom control logic and custom events between itself and the MC.
2. The JC must be able to communicate with its associated junction(s) in the simulation through only the use of UCF Objects. It must not have any knowledge of the simulator.
3. The JC must be able to monitor how the junction is operating.
4. The JC must implement changes to UCF objects itself or changes by the MC.
5. The JC must also provide a fall-back coordination plan in case data communications between itself and the Main Controller (MC) fails.

JC Non-Functional Requirements

1. Communications between JC and MC must be secure.
2. Communications between JC and MC must be able to work over the internet (this means through firewalls).

4.1.3 MC Requirements

The requirements listed for the Main Controller (MC) component are divided into three categories: functional requirements and non-functional requirements.

MC Functional Requirements

1. The MC must be open ended enough to support the adding of custom control logic and custom events between itself and the JC.
2. The MC must be able to handle work load ranging from a single Junction environment to practical limits without problems.
3. The MC must provide a platform to collect data for analysis.

MC Non-Functional Requirements

1. Communications between MC and JC must be secure.
2. Communications between MC and JC must be able to work over the internet (this means through firewalls).
3. The MC data must be presentable in a clear and extensible format.
4. The MC data must be fault-tolerant and reliable.
5. Data stored in MC must be verified to be consistant.

4.2 UCF Architecture

This section first describes the functional and technical architecture of the Urban Cloud Framework as a whole before providing a more detailed description of the architecture for each of the components: the VISSIM component, the Junction Controllers (JCs) and Main Controller (MC) component.

4.2.1 UCF System-wide Functional Architecture

The functional architecture of the entire UCF is described in this section, beginning with a high-level overview of the system, a description of why a Service-Oriented Architecture approach is the most suitable for the system, and finally an analysis of how the chosen architecture helps us to fulfill the requirements listed above.

Overall Architecture The functional architecture of the Urban Cloud framework can be divided into three separate components, which themselves can be split into three layers. As we can see in Figure 4.1 the three main component areas are the VISSIM component, the Junction Controller (JC) and the Main Controller (MC). The three layers shown in this diagram represent at the base, the operating system layer, the network layer and the logical system layer.

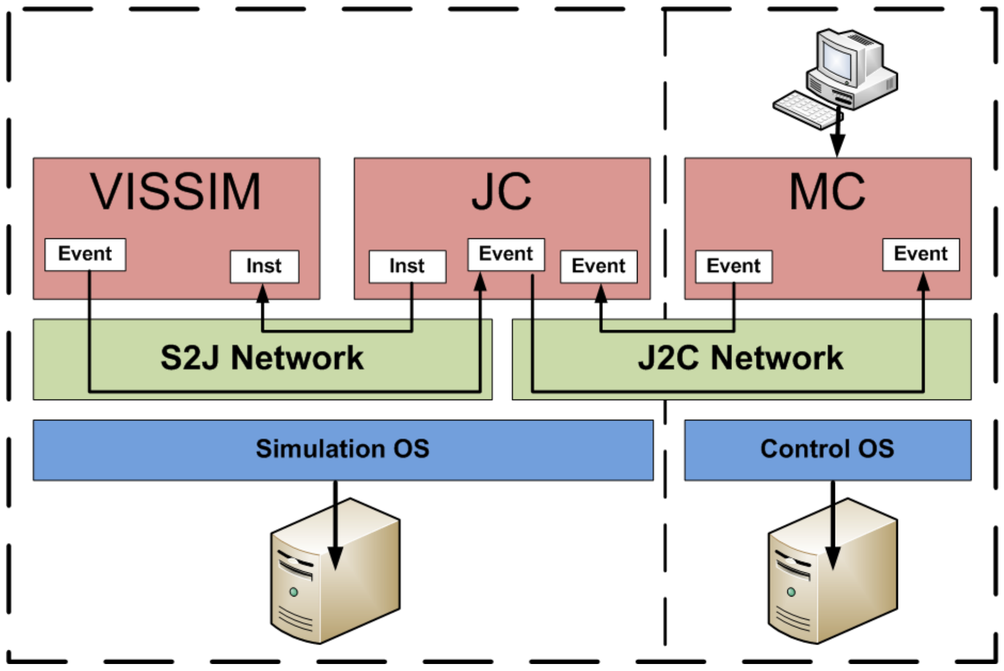


Figure 4.1: UCF System-wide Functional Architecture

The importance of viewing the architecture at this level is that it shows what communications between the components will look like. In a real world example, data from the VISSIM module would represent all the ground detectors and traffic lights in a major city. For example, in the SCATS architecture there is a controller box for each junction which has the ability to grab data from these sensors (for instance the induction loop or a camera used to count vehicles) as well as control the physical traffic lights on the road. The link between these and the detectors and traffic lights are an array of dedicated wires.

As the use of VISSIM for the traffic simulation is a requirement, and with that an inherent separation between VISSIM and the set of JC's, we must have a more complex network (than just simply wires) in place to accommodate this. As a result, we've introduced the simulator-to-junction controller (S2J) network. VISSIM will generate UCF data from the simulators COM interface, the VISSIM component will then pass this traffic data to the appropriate JC over the S2J network. On the flipside, the JC, which has a set the active phases, splits and offsets for a Junction. In reality, the JC is implemented on top of the VISSIM component, but from a functional perspective they are very different.

The second network introduced is the connection between the JCs and the MC module running on Amazon Web Services. The Junction to Controller (J2C) network is a connection between the JCs and the MC component. The JCs send relevant traffic data in the form of SimEvents, that was taken from the VISSIM component and forward it onto the MC for it to use. From the MC's perspective, this network is used to transport updates (for example changing the phase timings for a particular junction, updating the set of active phases) from the MC component to the appropriate JC. For the most part the MC will be a reactive (event based) system in the sense that it will only take action when receiving information from a JC.

Service Oriented Architecture The design of this architecture will utilize a Service-Oriented Architecture model for the link between the JCs and the MC which will help to meet the global requirement of modularity. The loosely coupled nature of SOAs, allows us to swap out a new control algorithm (perhaps SCOOT) if needed in the future. This is possible by having a clear partition between services which are bridged with web services. The same goes for partitioning of services within the MC module such as storage, databasing and processing; these will all be accessed using web services. The separation between the VISSIM component and JCs will be that of different C# DLLs, where interaction between the simulator components will be done by importing different namespaces into the project.

4.2.2 System-wide Technical Architecture

The system-wide technical architecture shows the same general division the function architecture does. It is split up into the VISSIM component, the Junction Controllers (JCs) which are both running on a Microsoft Windows XP machine and the Main Controller (MC) running on Amazon Web Services.

The VISSIM component will only use the *VISSIM_COMSERVERLib* to communicate with the VISSIM simulator, which provides methods for extracting monitoring data from the simulator and to send control data back to it. The VISSIM COM Server Interface provides a way of getting information about each junction on a per-step basis from the simulator. It provides access to many of the following objects:

- SignalController, which allow to enumerate through data which is applicable for each Junction Controller.

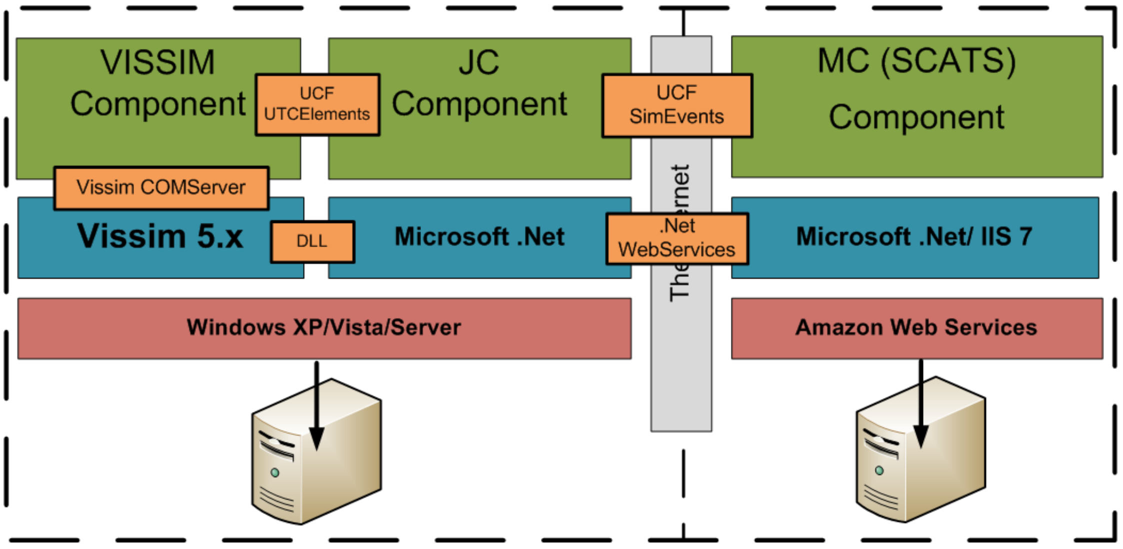


Figure 4.2: System-wide Technical Architecture

- Each SignalController contains a number of SignalGroup objects associated with them which represent phase data in the system. Each of these contain Signal Head Objects which represent a single traffic light from input one Link (or approach) to one output Link.
- Each SignalController contains a number of Detector Objects associated a Link and Lane (or Approach).

For a full list of functions, see the VISSIM COM Manual [53].

Key to the sensing in any controller algorithm is receiving a constant information stream from the Detectors to the JCs. This is done through the Detector Object, which allows us to check attributes like PRESENCE which tells us if a vehicle is currently over a detector, HEADWAY which is the time gap in seconds and OCCUPANCY which tells us how long in seconds since the arrival of the last vehicle. Using this information combined with what approach the detector is on and which junction this data belongs to allows for the clean, segmented extraction of data from junctions to give the JCs.

On the control side of the VISSIM component, traditionally the simulator provides a DLL control interface which allows for the setting the active signal groups, how long the light is green and how long the amber should be. Instead, using the COM interface combined with setting the control DLL to fixed traffic, we can write and abstraction to control traffic this way. The abstraction of this on the VISSIM component will allow for a round-robin change of the current phases which will be directly updated at the end of the cycle. The Phases themselves will not change (as the SignalGroups themselves will never change), instead the amount of split the phase will get will be changed on a per-cycle basis, as well as which phases are currently active and running.

The VISSIM component and the Junction Controller, although functionally different, will be tightly coupled in the same project. The Junction Controller Unit will be programmed in C# and will have a web service interface (using .Net web services) to the MC on the cloud. The .Net WebServices framework allows for the easy serialisation of Traffic Data into XML and will be sent via SOAP to the web service of the MC component, which is defined and made public using Web Services Description Language (WSDL). All communications between the JCUs and the MC will be done using SSL to ensure confidentiality and consistency of the data as it passes between the components.

The MC will run on a number of Microsoft Windows 2003 Amazon EC2 instances equipped with IIS Server 7.0 and the .Net Framework version 3.5. All code will be written in C#, with the use of .Net web services. The Amazon Simple Storage Service (S3) will used in conjuncton with SimpleDB for data storage, such as the current state of the Junctions which will be used by the MC as it will not keep persistant data. There are no gaurantees to whether the Elastic Load Balancer (ELB) give you the same machine the next time a JC sends a SimEvent to the MC, so this poses the same problem that results in a need for external persistant storage.

The MC's presence will be split across a number of Amazon's Availability Zones which ensure that network problems for a zone will be restricted to that particular zone. As stated, the load balancer will be implemented with the Amazon Elastic Load Balancer (ELB) which will tie every instance of the MC to a single public DNS which a domain name can be associated with. This means that externally the system looks like a centralized source, althought internally it is very decentralized.

The ELB will ensure that none of the instances will be overloaded with traffic from the simulator(s). For each of the Availability Zones, all MC instances will be assigned to an Auto Scaling Group. This Auto Scaling Group will be responsible for ensuring that there are enough instances up to consume the amount of traffic send by the simulator(s) and will use metrics from Amazon Cloud Watch to formulate triggers for when the instances should scale up and back.

Chapter 5

Implementation

The implementation of this project is split into three sections. The first section deals with the construction of the Urban Cloud Framework, first describing each of its core components, how they work and how exactly they were implemented. For the UTCController and UTCJunction-Controller, an example skeleton for building a new controller is shown. The second section deals with the abstraction of VISSIM, how its core components work, what objects it works with in the UCF framework and how it was implemented. Finally, the third section deals with the implementation of SCATS on the UCF, how it uses the classes and interfaces defined by the UCF to control traffic on the VISSIM side without having to distinguish whether it is a simulator or the real-world.

5.1 The Urban Cloud Framework

The Urban Cloud Framework (UCF) is defined as a set of objects that can be used with a simulator on one end with is coupled with a Junction Controller (JC) with can speak to a Main Controller (MC) in which both controllers are written by the user.

5.1.1 UTC.UTCElements

The UTCElements are the core objects by which every other object in the UCF is built. These are objects which represent approaches, connectors between approaches, the detectors that exist on incoming approaches, the phases that control which approach is active during a cycle and an object to represent the junction itself. Each of these elements are written as C# objects that are bundled together into a DLL that can be used in other projects.

JApproach

The JApproach is an object which represents any incoming or outgoing approach to a junction defined as a LinkID, LaneID pair of Integers. It is important to have both a LinkID and LaneID as there can be multiple lanes in a single link. From a real-world traffic control perspective, a detector will most likely be a on a per-lane basis, although it is possible to have a single detector for the whole approach. There will be at least three approaches in a junction. An example is the O'Connell Junction shown in Figure 2.2 which contains three incoming approaches and three outgoing approaches. The JApproach is written in such a way that it is [Serializable()], meaning that the object can be serialized into XML without any problems.

JConnector

The JConnector is an object which represents the connection between an incoming approach and an outgoing approach (both defined as JApproach objects) for a junction. The JConnector itself will have a unique ID by which it can be referenced, since a single incoming approach can have multiple outgoing approaches. This is why the connector is important to the framework as it can describe a topology of every incoming approach to outgoing approach, allowing a map for the possible flows of traffic for the junction. The JConnector is written in such a way that it is [Serializable()], meaning that the object can be serialized into XML without any problems.

JDetector

The JDetector is an object that represents a detector in either a simulation or the real world. The type of detector that it is (for example an induction loop or traffic camera) does not matter to this object explicity, but rather its ability to perform some of the main functions of real detectors. Each JDetector has an ID which is unique to the map that is being used. The JDetector is also explicitly associated with the JApproach it is servicing. This is important as it enables the ability to tie the data the JDetector produces to a particular part of the junction.

The JDetector contains primitive functions for telling how many cars have passed over the detector so far, which can then be reset on whatever basis you need, such as at the end of a phase, cycle or on some condition specified by the user and a presence flag which is set if there is currently a car over the detector currently. This class is kept abstract, so the user can extend the object to provide more complex functions for the detectors, such as in SCATS counting the Non-Occupancy time, the Maximum-Flow ever recorded and the Average Occupancy of the lane. The JDetector is written in such a way that it is [Serializable()], meaning that the object can be serialized into XML without any problems.

JPhase

The JPhase is an object that represents a Phase in a given traffic cycle. It contains an ID that is unique to the current junction with which it is associated, as well as a name that will give some context to the user to indicate what phase the current object represents, such as the "A" Phase, which is traditionally defined as the most important phase for a junction. As well, it has values for defining the amount of green time and amber time that the current phase will get as well as the total phase time, which is calculated by adding the green and amber times to the junction-wide clearance interval.

As well, each JPhase has a List of associated SignalGroups, these are groups of signals that are active during a phase. A SignalGroup is a set of signal heads. These signal heads reresent a path from an incoming approach, through a connector to an outgoing approach. It is important to differentiate between each signal head, as there is a functional difference between a lane turning left and the same lane going straight through the junction. The JPhase is written in such a way that it is [Serializable()], meaning that the object can be serialized into XML without any problems.

Junction

The final object in UTCElements is the Junction which represents the junction itself. The Junction contains an ID that is unique to the current junction associated with it in the map. The Junction also possesses a Name in order to give it some context to the user, for example "O'Connell Stree Bridge". Also very important to the Junction is its clearance interval, which sets the time for which all traffic lights should remain red to clear the junction before the next phase starts. The object also provides a Dictionary which has a mapping between DetectorID and its associated JDetector object and Dictionaries for the mappings between PhaseID and their associated JPhase for both the full set of Phases and the sub-set of phases which are the currently Active Phases. The Junction also contains a cache for the currently active PhaseID, Name and PhaseTime for which the currently active phase will run. For this current phase, there also exists a variable which contains the current number in seconds until the phase is finished. This is very important as it is decremented for every second the simulation runs until it reaches zero, when logic from other higher objects will kick in to change it to the next active phase. The Junction is written in such a way that it is [Serializable()], meaning that the object can be serialized into XML without any problems.

5.1.2 UTC.Controller

The next highest order of name spaced objects are those in the UTC.Controller namespace. These are objects that provide logic for what to do with the set of objects in UTC.UTCElement it has control over for a map. Taking a bottom-up approach, the lowest layer of the this set of objects are the SimEvents. The next layer up is the EventInterface, which provides a transport layer between the UTCJunctionController and UTCController classes for any generic SimEvent (using the GenericSerializer) and all associated sub-classes, as they will not be known at compile time. Finally, the top layer being the UTCController and UTCJunctionController object which provide a skeleton for developing your own traffic controller on top of the UCF.

UTC.Controller.EventInterface.SimEvents

The SimEvent object and any sub-classes produced by the user are at the very core of communication between a JCU and the main controller (in the case of the implementation in the next section, SCATS). The SimEvent object is very simple and provides the following features:

- An EventID Guid which keeps the events unique.
- A DateTime for when the current event is created.
- An SimulationID Guid which is kept for logging purposes.

The SimEvent class itself is abstract, meaning that there is no instantiation of this object directly. Instead it provides more of a skeleton for developing objects which extend this, such as EndOfPhaseSimEvent. This is responsible for sending PhaseData from the JCU to the main controller at the end of a phase. UpdatePhaseSimEvent will send the new phase data for a JPhase by its unique ID.

UTC.Controller.EventInterface

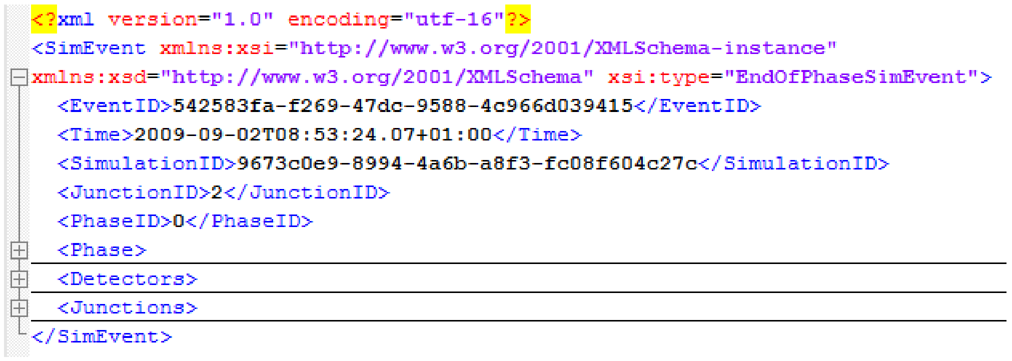
The EventInterface is quite a simple Object which is written on top of the abstract UTCController (and of course any of its sub-classes). It is responsible for providing a transport layer for sending and receiving generic SimEvents between any two UTCControllers. For Controllers that need two-way communications, it provides two methods:

- The sendSimEvent(Object simEvent) method is used to send a SimEvent to another Controller. This applies to any controller that extends the UTController, including the UTCJunctionController object. This method uses an Object as a parameter so the ThreadPool class can be be used to push simEvent's on to be sent.
- The receiveSimEvent(String simEvent) method is used to receive a SimEvent from another Controller. This applies to any controller that extends UTCController, including the UTCJunctionController object. This method uses a String as a parameter, as a custom XML Serializer had to be written to accommodate the serializing of objects that are not known at compile time.

GenericXMLSerializer The GenericXMLSerializer object is an extension of the .Net 2.0 libraries XMLSerializer which is needed to serialize and de-serialize objects by only having knowledge of its superclass (i.e the SimEvent object) at compile time. As we want to be able to write classes that extend SimEvent for whatever controller is being written by the user, it wouldn't make much sense to require the user to write an [XMLInclude()] for every new SimEvent they write. To avoid all of this confusion, the GenericXMLSerializer simplifies this by using the .NET Reflection package to build a list of all objects that extend the SimEvent at runtime and adds an array of Type that can be used when the XMLSerializer is instantiated. The XMLSerializer then has a reference to what each of the sub-classes look like at runtime and can serialize/de-serialize to/from XML easily one the classes themselves are marked at [Serializable()].

An example of the output of the GenericXMLSerializer object looks as follows:

The main observation to note is that the object itself is still a SimEvent and what differs between sub-classes is the it's xsi type, which in the case of Figure 5.1 is an EndOfPhaseSimEvent.



```
<?xml version="1.0" encoding="utf-16"?>
<SimEvent xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xsi:type="EndOfPhaseSimEvent">
  <EventID>542583fa-f269-47dc-9588-4c966d039415</EventID>
  <Time>2009-09-02T08:53:24.07+01:00</Time>
  <SimulationID>9673c0e9-8994-4a6b-a8f3-fc08f604c27c</SimulationID>
  <JunctionID>2</JunctionID>
  <PhaseID>0</PhaseID>
  <Phase>
  <Detectors>
  <Junctions>
</SimEvent>
```

Figure 5.1: An example of a EndOfPhaseSimEvent after being run through the GenericXMLSerializer.

As you can see, all data relevant to the superclass, the EventID, Time and SimulationID are serialized in with the data that is required for this particular SimEvent.

UTC.Controller.UTCController and UTCJunctionController

The UTCJController is the base class for describing any controller that is capable of using all of the UTCElements. Every UTCController has a UTCControllerEventInterface which it uses for transport between itself and another controller. It also has a Dictionary of Junction objects which are indexed by the Junction's ID. These Junction objects represent the current state of the junctions in the simulator for each time step.

The base UTCController classes itself is very simple and only has a handful of methods which are used within it. They are:

- Two methods, getEventInterface() and setEventInterface(UTCControllerEventInterface e) which are used to allow the UTCController to communicate with other controllers.
- A getJunctiondByID(int jID) which is used to look up the current junction which needs attention in the system.
- The processEvent(Object e) is the most important method in the UTCControllers, it is the method where a SimEvent enters the instantiated UTCController sub-class and is processed. It is a virtual method which is left completely blank in the base class, as there is nothing that can be done here until the user overrides this method to write the logic for their controller. The method uses an Object as a parameter so the ThreadPool class can be be used to push simEvent's on to be sent.

The UTCController's construction is intentionally simple, as it contains only the essential elements needed to start a controller. Any logic defined to make the controller work to the user's specifications is worked out when they write their controller.

Obviously, some of the default behaviour for Junction Controllers that is generic enough to write into its own class is not captured in the UTCController. This is why the construction of the UTCJunctionController object which extends UTCController is necessary. The UTCJunctionController is an object which describes the minimal functions required for a user to make their own Junction Controller logic. The UTCJunctonController has a number of methods, some of which are tied to every instance, and others which are overridden when writing a new junction controller. The ones which are overridden by the user when they write their own junction controller are:

- The atTimeStep(int jID) is a virtual method that is triggered for each Junction object at any time step. When the user implements their own junction controller, this method should be treated as what to do when it is the start of a new timeStep for JunctionID jID.
- The atDetectorPresence(int jID, int dID, int presence) is a virtual method that is triggered when a car is passing over the Detector with DetectorID dID. The DetectorID is an explicit parameter as there can be many detectors that are active for each phase.
- The atPhaseChange(int jID) is a virtual method which acts as a trigger for when the current phase for JunctionID jID is about to end. Any data needed about the currentPhase is contained inside the Junction object.
- The postPhaseChange(int jID) is a virtual method that is triggered just after a phase has been changed to the next in the list of active phases for JunctionID jID.
- The atEndCycle(int jID) is a virtual method that acts as a trigger for when the current cycle is about to end. Having the state of the Junction before a cycle ends is important if a record is needed about Junction jID.
- The postEndCycle(int jID) is a virtual method that acts as a trigger for when the current cycle has ended and the Junction jID is now ready for the next cycle (i.e detector counts have been reset).

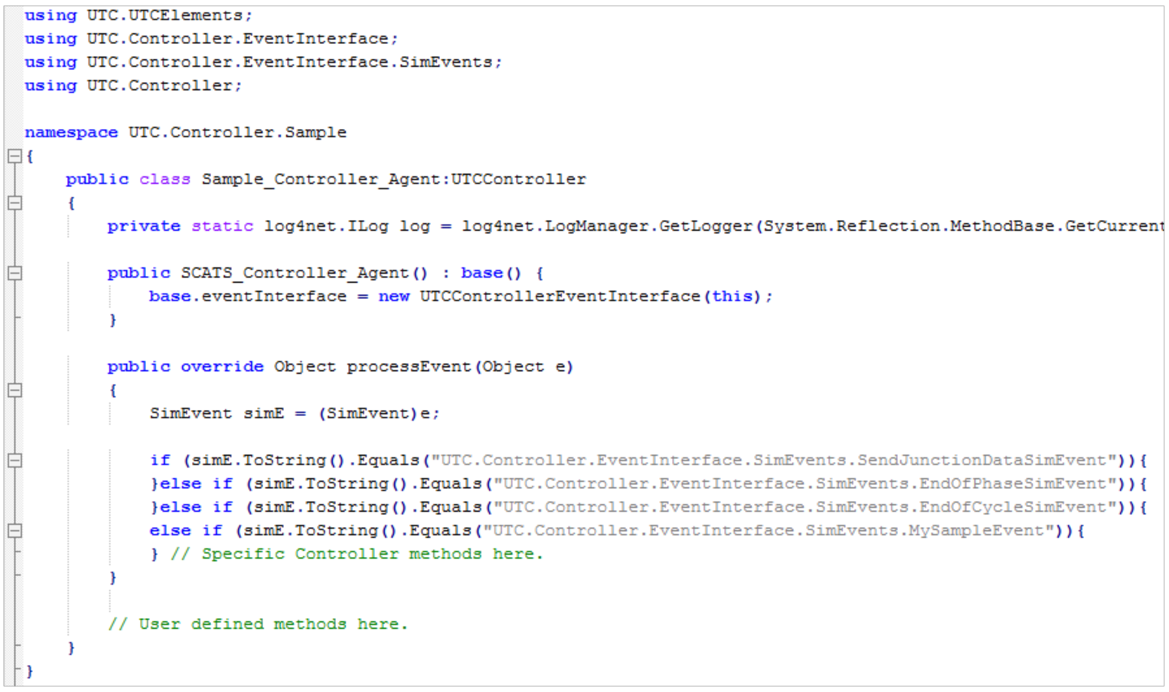
The methods that do the fundamental control that every junction controller uses, including sub-classes of the UTCJunctionController are as follows:

- The doDetectorStepForJunction(int jID) is a method that contains the logic for what to do for each Detector during a time step. It first gets each Detector that is relevant for the current JunctionID and CurPhaseID and checks to see if there is a car currently over the detector. It then calls the trigger atDetectorPresence() for sub-classes to use.
- The setNextPhaseForJunction(int jID) is responsible for setting the next active phase for JunctionID jID. It also sets the current phase cache in the Junction object to be this new active phase and sets the countdown to be the greenTime + amberTime + clearanceInterval.
- The endCycle(int jID) provides the logic for when the current cycle ends. Inside this it calls the atEndCycle() and postEndCycle() methods described above and resets all the junction data for the next cycle.
- The updatePhaseDataForJunctionID() method takes in an UpdatePhaseSimEvent which extends simEvent. Using the data inside this object, the phase data for a junction can be updated on a per phase basis, per cycle basis or whenever the user specifies.

- The most important method is the doTimeStep() which is concerned with what to do everytime the simulator does a new step. In this method, it interates through every object and calls the appropriate functions listed above.

5.1.3 Sample UTCController and UTCJunctionController skeletons

The end result of the these classes are that all functions written provide the minimum code to allow the monitoring and controlling of an adaptive Urban Traffic Control System without getting specific enough that it would hinder the development of one. As well, we have essentially made a model for UTCs that is event based, which will make the process of designing a controller much easier as we will only have to worry about what to do when an event occurs (such as when a Phase ends). The end result of the UTCController produces a skeleton example such as the one in Figure 5.2:



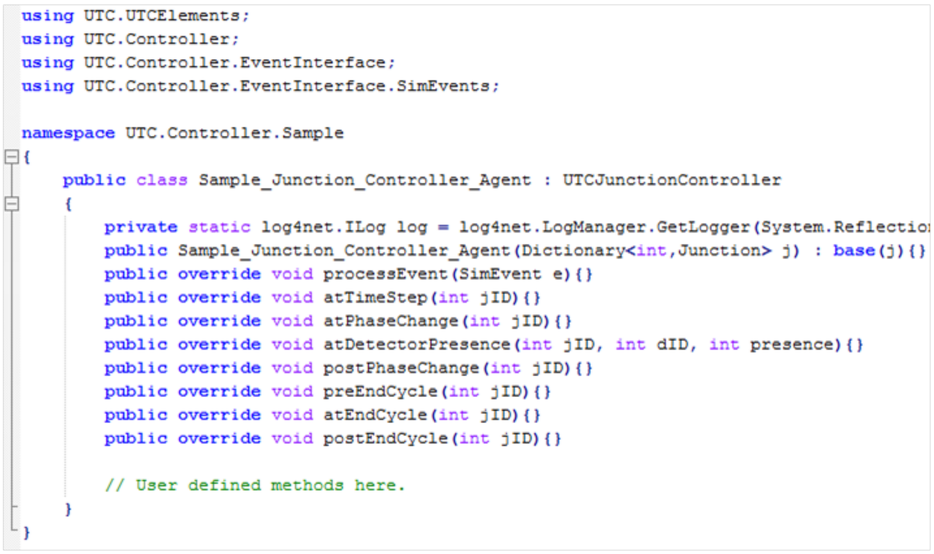
```
using UTC.UTCElements;
using UTC.Controller.EventInterface;
using UTC.Controller.EventInterface.SimEvents;
using UTC.Controller;

namespace UTC.Controller.Sample
{
    public class Sample_Controller_Agent:UTCController
    {
        private static log4net.ILog log = log4net.LogManager.GetLogger(System.Reflection.MethodBase.GetCurrent

        public SCATS_Controller_Agent() : base() {
            base.eventInterface = new UTCControllerEventInterface(this);
        }

        public override Object processEvent(Object e)
        {
            SimEvent simE = (SimEvent)e;

            if (simE.ToString().Equals("UTC.Controller.EventInterface.SimEvents.SendJunctionDataSimEvent")){
            }else if (simE.ToString().Equals("UTC.Controller.EventInterface.SimEvents.EndOfPhaseSimEvent")){
            }else if (simE.ToString().Equals("UTC.Controller.EventInterface.SimEvents.EndOfCycleSimEvent")){
            else if (simE.ToString().Equals("UTC.Controller.EventInterface.SimEvents.MySampleEvent")){
            } // Specific Controller methods here.
        }

        // User defined methods here.
    }
}
```

Figure 5.2: The UTCController skeleton for writing new controllers.

As you can see in Figure 5.2, to begin writing our own sample controller, we would start with this skeleton. The main controller only has to worry about what to do when taking in a particular SimEvent, then processing it and what SimEvent to send back. For example, if we wanted to implement a random traffic controller, when receiving an EndOfPhaseSimEvent, we should simply return an UpdatePhaseSimEvent to the junction controller. Once the logic between the main controller and junction controller matches up, traffic control can be achieved.

Figure 5.3 shows the slightly more complex skeleton for writing our first junction controller:

As you can see, this has a couple more in-built methods to UTCController class which it extends. In the case of this controller, all we have to do is fill in the logic for how the junction



```
using UTC.UTCElements;
using UTC.Controller;
using UTC.Controller.EventInterface;
using UTC.Controller.EventInterface.SimEvents;

namespace UTC.Controller.Sample
{
    public class Sample_Junction_Controller_Agent : UTCJunctionController
    {
        private static log4net.ILog log = log4net.LogManager.GetLogger(System.Reflectio
        public Sample_Junction_Controller_Agent(Dictionary<int,Junction> j) : base(j){}
        public override void processEvent(SimEvent e){}
        public override void atTimeStep(int jID){}
        public override void atPhaseChange(int jID){}
        public override void atDetectorPresence(int jID, int dID, int presence){}
        public override void postPhaseChange(int jID){}
        public override void preEndCycle(int jID){}
        public override void atEndCycle(int jID){}
        public override void postEndCycle(int jID){}

        // User defined methods here.
    }
}
```

Figure 5.3: The UTCJunctionController skeleton for writing new controllers.

controller should react to the functions shown. To write the simplest junction controller that will just assign phases randomly, we need only assign a new set of active phases to the junction using the atEndCycle() if we want to process this locally. If we want the main controller to generate this, we could potentially send a EndOfCycleSimEvent and let the main controller update the active phases on its end. Another alternative would be to write our own Randomly-GeneratePhasesSimEvent in which the junction controller would send its current phase's data to the main controller at the end of its phase and the the main controller would reply with just this phase updated with new phase data.

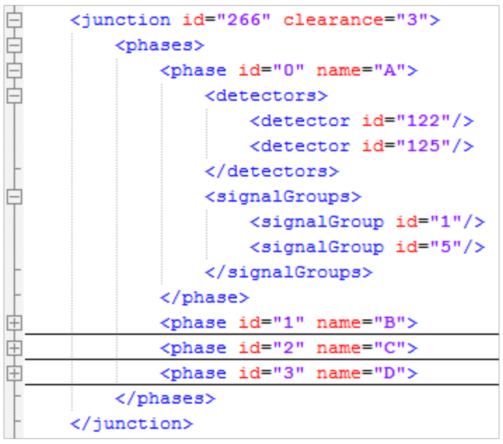
There are many ways in which traffic control can be done using the UCF's UTCController and UTCJunctionController classes. It is up to the user to choose which way they plan on doing so and to ensure the solution is not wasteful on resources, as transmission of data across web services can be very costly.

5.2 VISSIM Component

The VISSIM component exists as UTC.Simulator.VISSIM to allow the extraction of data from the VISSIM simulator, combined with the configuration and phaseData XML files and the UCF translate its content into objects the UCF can use for traffic monitoring and control on the JCU and main controller side. It has three elements: the PhaseDataBuilder which uses the phaseData.xml file to build a relationship between what SignalGroups in VISSIM correspond to which phase they are in, the JunctionDataBuilder which is used for building junction data from Vissim's COM Server and combines it with the information from the PhaseDataBuilder to produce a set of UTCElements that are ready to be used for simulation, and the VissimWrapper which puts this all together to run simulations that use the embedded junction controller DLL written by the user.

5.2.1 PhaseDataBuilder

The PhaseDataBuilder is a C# class which reads an XML file responsible for providing a map of what SignalGroups are active, as well as a map of which detectors should be active during a particular phase. Once this data is parsed into two Dictionary objects, one with a mapping of JunctionID to another Dictionary of JPhase objects and the second a Dictionary which is a mapping of PhaseID to another Dictionary of JDectector objects. The PhaseData.xml file itself appears as the following format:



```
<junction id="266" clearance="3">
    <phases>
        <phase id="0" name="A">
            <detectors>
                <detector id="122"/>
                <detector id="125"/>
            </detectors>
            <signalGroups>
                <signalGroup id="1"/>
                <signalGroup id="5"/>
            </signalGroups>
        </phase>
        <phase id="1" name="B">
        <phase id="2" name="C">
        <phase id="3" name="D">
    </phases>
</junction>
```

Figure 5.4: An example PhaseData XML file for a junction.

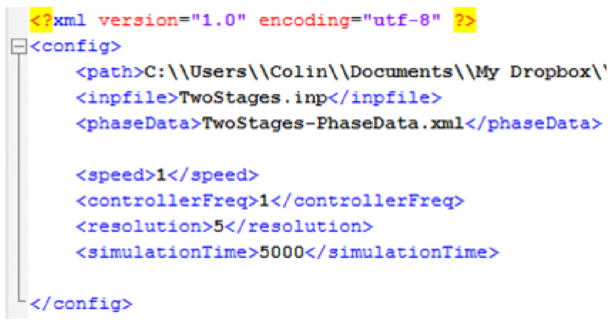
As you can see the format is extensible enough to allow the capturing of phase data. The 'A' Phase for Junction 266 has two Detectors with DetectorID 112 and 115 in which SignalGroupID 1 and 5 are active. This phase is a very simple one, showing the first phase in a simple Junction when two opposite approaches are active. In the simulator itself the SignalGroups are broken in to a set of SignalHeads which represent a single traffic light. For example there are two SignalHeads for an approach to turn left and to go straight through the junction; in both cases they should use the same detector.

5.2.2 JunctionDataBuilder

The JunctionDataBuilder is a C# class which works with the Vissim_COMSERVERLib DLL to build data from the current simulation into UCF objects which can be used for simulation. This DLL provides a COM interface which provides services for starting, executing and retrieving results from a Simulation while on a per-time-step basis, allowing a look at the state of each object. The end result of this object will allow for a Dictionary object of Junctions which are indexed by JunctionID that includes data from the PhaseDataBuilder

5.2.3 The VissimWrapper

The VISSIM wrapper is the main object which is used to run simulations with VISSIM. When the VISSIM Wrapper is run, it reads a configuration XML file which gives the following details:



```
<?xml version="1.0" encoding="utf-8" ?>
<config>
    <path>C:\\Users\\Colin\\Documents\\My Dropbox\'
    <inpfile>TwoStages.inp</inpfile>
    <phaseData>TwoStages-PhaseData.xml</phaseData>

    <speed>1</speed>
    <controllerFreq>1</controllerFreq>
    <resolution>5</resolution>
    <simulationTime>5000</simulationTime>

</config>
```

Figure 5.5: An example Config XML file for running simulations with the Vissim Wrapper.

The config XML contains a number of attributes which it uses to run simulations. The first, *path* is the absolute path to where the VISSIM .inp map file is located. The *inpfile* tag is used to poinpoint exactly where the map file is located in the path directory. The *phaseData* tag is the location of where the PhaseData.xml (which is used with the PhaseDataBuilder object) file is located in this directory. The next couple of tags are to do with how the simulation will run, the *speed* tag is as a parameter to the Vissim object in the COMServer for how fast the simulation should attempt to run (1 being real-time). The *controllerFrequency* tag is how many times per second objects in the simulation such as The Detecors should sense for vehicles passing over. The *resolution* tag is used for how many steps there should be in a simulation second (5 being a stepsize of 1/5 of a second) and the *simulationTime* is used for how long the simulation should run.

The Vissim Wrapper includes the objects which the simulator provides such as the Vissim object, the Simulation object and its sub-classes and the Evaluation object which contains evaluation data.

5.2.4 Controlling Traffic using only VISSIM_COMServerlib

The traditional method used to control traffic involves writing a specific controller DLL in C++ which would contain the logic for the junctions. To simplify the process, it is possible to control the traffic using only the COM interface. A couple of issues do arise when attempting to do so:

- The user has to take care of exactly when to switch the next signalGroup active. This is done by first setting all of the signalGroups RED_END attribute to be the time that the current phase ends.
- The active signalGroups are then set to a RED_END of the current timeStep and the GREEN_END to the phaseTime minus the clearance interval for the Junction.

- When the phaseCountdown in the UCF Junction object hits zero all SignalGroups are reset and the process it continued.
- For this to work properly, the CYCLETIME for the entire simulation has to be set to the entire simulation time which is located in the config XML under *simTime*.

Although not very complicated to implement, it saved a lot of time that would be spent looking that their Signal Controller DLL skeletons. It also allows the full UCF implementation to be in a single language, C#.

5.3 SCATS on the Urban Cloud Framework

The implementation of SCATS on the UCF starts with the two skeletons defined in the UCF section above. From the UTCController and UTCJunctionController objects we can define our own objects that extend these. For this portion on the implementation, they will be known as the SCATS Main Controller (SMC) and the SCATS Junction Controller (SJC).

5.3.1 SCATS Main Controller (SMC)

The SCATS Main Controller (SMC) is the simpler defined skeleton of the two controllers. All interactions between itself and the SJC will be done exclusively through SimEvents. This means that we only have to worry about the SimEvents that are introduced to the controller via the overridden processEvent(Object e) method which takes in a SimEvent on a ThreadPool from the controllers EventInterface and when finishing processing returns another SimEvent to the SJC as a result. For instance, if we receive a EndOfPhaseSimEvent from the SJC, we can process the phase using the SCATS algorithms and return an UpdatePhaseSimEvent back to the SJC for processing on its side.

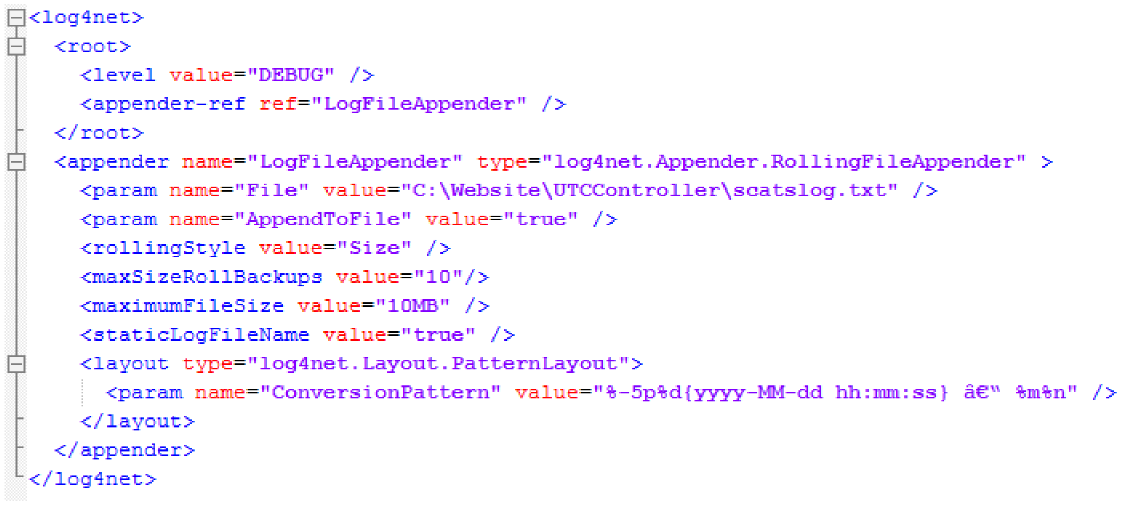
The processEvent(Object e) method therefore becomes the critical point in the definition of a controller. Inside this method, we first cast the generic object (this must be an object, so this method can be used with ThreadPools) into a SimEvent and then check the SimEvents Type and cast it to the appropriate one. The sections of code inside these statements then become a place to start defining controller logic.

SMC Instantiation on an single machine

Speaking first from within a single thread in the IIS server, every time the web service is called it starts a completely new instance of SMC. Therefore, the constructor has to call S3 to get a copy of the Junctions with which it is working. To retrieve the correct set of Junctions from a large set, they are indexed on S3 in the *scats-data* bucket with their own unique key of the SimulationID which is encapsulated in every SimEvent. Also important to the SCATS algorithm is the last two states of the Original Volume (OV), Degree of Saturation (DS) and Reconstituted Volume (VK). These states are combined with the current state to give the smoothed result for the three. It is important to smooth these results as massive fluctuations between phases/cycle are possible during events like ambulances passing by. With this set of data ready, this current instance is now ready to serve as a SMC.

Logging

The use of logging is very important for systems whose top layer is a web service. As there is no specific interface set up in .Net for handling where the output of the *Console.WriteLine()* goes, the use of an enterprise ready logging environment is required. For this, Apache's log4net libraries are perfect for our purposes. It provides a number of different logging priorities such as INFO, DEBUG, ERROR and FATAL and is easily integrated into the IIS webserver, by simply adding a single line to the Web.config file and the following lines for how the log should output and where:



```
<log4net>
  <root>
    <level value="DEBUG" />
    <appender-ref ref="LogFileAppender" />
  </root>
  <appender name="LogFileAppender" type="log4net.Appender.RollingFileAppender" >
    <param name="File" value="C:\Website\UTCController\scatslog.txt" />
    <param name="AppendToFile" value="true" />
    <rollingStyle value="Size" />
    <maxSizeRollBackups value="10"/>
    <maximumFileSize value="10MB" />
    <staticLogFileName value="true" />
    <layout type="log4net.Layout.PatternLayout">
      <param name="ConversionPattern" value="%-5p%d{yyyy-MM-dd hh:mm:ss} â€" %m%n" />
    </layout>
  </appender>
</log4net>
```

Figure 5.6: The log4net configuration file.

All logging using the log4net DLL is completely thread safe. As there will undoubtedly be a large number of SMC's running on the same machine, this is very important so we can be positive that we have captured all necessary data.

Bundling the SMC AMI

Once the code for the SMC was finished, the web service was published on the single Windows Server 2003 Amazon EC2 instance so it could initially be tested with the Simple 2-Stage Junction map. Once the tests looked positive, it was time to bundle the AMI, so I could launch any number of these instances to start the scalability testing. In order to bundle an Amazon AMI, you need to install the Amazon EC2 AMI tools package for Windows which is written completely in Java, so the JRE has to be installed as well and the paths set for *JAVA_HOME*, *EC2_HOME*. Then, using the *ec2-bundle-instance* command, inputting my AWS Access and Secret Key as well as a name for the AMI's manifest *scatsami/utcontrollerbundle.manifest.xml* which is stored in the scatsami bucket in S3. Bundling can take quite some time, in the range of about 20 to 40 minutes, but once done the AMI is sitting in the S3 and can easily be launched by calling its manifest.

Launching the full SMC Service

In order to launch the full scalable SMC Service we need to integrate the SCATS AMI with a Load Balancer, set up its Launch Configuration and configure the Auto-Scaling Group.

Using the Elastic Load Balancer (ELB) a new instance was created called *ScatsLB*. The ELB is set up to automatically balance incoming traffic evenly across the Availability Zones which the instances run. In our case the ScatsLB runs over two zones and listens on ports 80 and 443 for incoming HTTP/HTTPS connections.

The Launch Configuration for the application, called *SCATSLauncher* has the ID of the SMC AMI which also includes a list of the Security Groups to which it is attached. The LC uses the default security group (which opens ports for Remote Desktop) as well as a custom security group, *UTCControllerSG* which allows for incoming traffic on ports 22 for having SSH support for each individual box and 80 and 443 for allowing HTTP/HTTPS traffic through.

The use of two Auto-Scaling Groups are required for the SMCs to run in the two Availability Zone for which they were set up. These are called *ScatsG1* and *ScatsG2*. These groups are associated with both the *ScatsLB* and *SCATSLauncher* for a minimum number of instances launched set to 1 and a maximum number set to 10.

5.3.2 SCATS Junction Controller (SJC)

The SCATS Junction Controller Agent is built by extending the skeleton for the UTCJunctionController to include the algorithms needed to implement the Junction Controller (JC) for SCATS.

The JC is quite simple for SCATS. At the end of every phase it will send back details for the number of vehicles which have gone over the appropriate detectors and the non-occupancy time for these detecors, which is the amount of time in seconds an approach during the phase did not have a car over the detectors. There is also a recording of the maximum flow over the detectors, being the largest number of cars that have ever passed over the detector during a phase and the average occupancy, which is defined as the number getting the average number of cars that have passed over during the maximum flow.

Starting a new simulation

As the SJC is embedded directly into the Vissim Wrapper, it is instantiated when it is called inside this program (along with the Vissim COM Server objects). In the current state, the SJC is explicitly defined in the Vissim Wrapper, breaking the modularity aspect as noted in the design section. There just wasn't enough time left to go back and change it to a generic UTCJunctionController which type is checked at runtime.

When the SJC is instantiated, the first object set is the eventInterface which is used to communicate with the SMC. The SJC then gets a set of UCF objects passed to it representing the Junctions in the simulation it will be controlling. This is set as a Dictionary of Junction objects which is indexed by JunctionID. At this stage the SJC immediately registers this initial data set with S3 so it can be passed on the SMC for processing at a later time. The simulation is saved in a *scats-data* bucket with the object being stored by the SimulationID Guid, which is created at the start of every simulation. The SMC can then directly reference the Junction data by using the SimulationID, which is passed with every SimEvent.

Calling SMC at the end of a phase

The SJC overrides the atPhaseEnd(int jID) function and for each of the JDetector objects in the currently active phase extracts this data and forms an EndOfPhaseSimEvent with this data, the JPhase object which represents the current phase and the JunctionID which is used on the MC side. It then publishes this finished event to the ThreadPool in its eventInterface to be sent to the SMC. Sometime later it should receive an UpdatePhaseSimEvent on it's threadpool from the eventInterface, which will process this event and update the active phase for the next traffic cycle.

Failure of the SMC

If the SCATS Junction Controller (SJC) attempts to call the SCATS Main Controller (SMC) and fails, the SJC has a fallback mechanism which is built into SCATS itself. If this occurs, the JC keeps a set of background phases to which it will automatically set its current set of active phases and continue to use these until communications between the JC and the MC are restored. As SCATS is a distributed system, it was built with fault-tolerance in mind.

Chapter 6

Evaluation

The section outlines how the implementation of SCATS on top of the UCF meets the definition of feasibility as described in the Design section. The first section outlines the tests that were done in order to explore the various questions raised. These questions can be broken down into proving that the system is Scalable, Reliable, Real-Time Capable, Fault Tolerant and Secure. There will also be an evaluation of costs associated with the project.

6.1 Feasibility

As described in the Design section, feasibility is measures in terms of Scalability, Reliability, Real-Time Performance, Fault-Tolerance, Security and Cost.

6.1.1 Scalability

The criteria for evaluation of Scalability is based on not placing limits on the number of Junctions that can be simulated on VISSIM and controlled with the MC and JC. To show this in action, the design of two tests were done to show scalability with the simulation of a large map and taking the map out of the equations with a dedicated script.

Scalability Test A

The first test of Scalability was to simply test a large map on the cloud to see if the MC is able to handle the load from the Junctions in the simulation. For this test, the use of a 46 Junction map representing the Urban Freeway in Redmond, US was used.

In order to use a map of this size for the simulation there needed to be quite a powerful machine needed, as at full load the map would simply run too slow. Using a Intel Core2 2.4Ghz with 4GB of RAM as the test machine, it brought the simulation speed up to an average speed of about *0.8 - 1.3* at full load.

On the MC end, the 46 junctions were not producing nearly enough of a load in either of the Availability Zones which would cause the Auto-Scaling Groups to trigger the addition of instances. Even bringing down the number of Availability Zones to a single one (which meant that the Load Balancer was serving a single machine) was not enough to trigger a scale. As well, the introduction of a larger map would be too slow to show how many junctions even a single instance can handle, so another method needed to be devised in order to show scalability of the MC itself, which would exclude the simulator from the equation.

Scalability Test B

The second test was designed to take the simulator out of the scalability test, as it was the bottleneck. Instead, a Perl-based scripting client was used. This client was developed to send large quantities of mock SimEvents to test what the capacity of the instances are. Starting with the same configuration as Scalability Test A, it includes two SMC's spread over two availability zones.

To set up the test, a mock PhaseData XML file with the SimulationID of 00000000-0000-0000-0000-000000000000 and with a data set representing 10,000 junctions at approximately 8.5kb each. The script would then pick a random number between 0 - 9999 and start a thread which sent a mock EndOfPhaseSimEvent with that random number in the place of the JunctionID. It would then check the result it got back from the SMC just to make sure it there were no errors in the data it was expecting back.

The results of this test show that after 4 minutes each Availability Zone attached to the ELB started a new instance as a direct results of the Auto-Scaling Groups triggers to start a new instance based on triggers defined to scale when the processing power of the server exceeded 60 percent and network traffic exceeded 1Mbit/s over a three minute period. The results show that the average number of SMC's controlled by a single instance is *approximately 500 junctions* with an average round trip time of *94ms*. This of course is averaged over the four servers which were active during that time it was measured. Another 4 minutes after the script was stopped, the ELB scaled each AutoScalingGroup back down to it's initial size.

6.1.2 Reliability

One of the main goals of Amazon Web Services is to provide a service that is guaranteed to be reliable by providing Availability Zones and Data Centres on two different continents. Any fault in a availability zone does not affect the applications running in another. The Elastic Load Balancer (ELB) can be used to push data across availability zones evenly, and in each of these zones Auto Scaling can be used to ensure there is always enough instances to meet the demand of the service. The SCATS Main Controller (SMC) uses these mechanisms to ensure that there is always a service available. For the SCATS Junction Controller (SJC), reliability isn't really an issue as it is tied in directly with the simulator and if of availability of the simulator would not allow for any simulation at all.

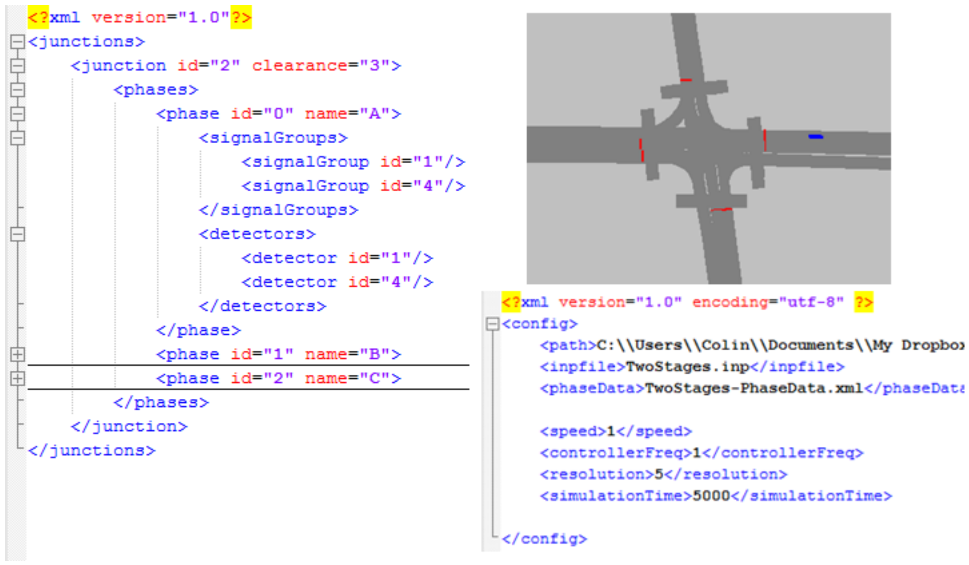
6.1.3 Real-Time Performance

One of the requirements to give a good indication of feasibility is that the Junction Controller (JC) is capable of sending data to the Main Controller (MC) and retrieve results back in enough time to work in at least a real-time environment. Anything faster than real-time is not necessary to proving feasibility, but is quite desired as it demonstrates that simulations can be controlled from the cloud.

Real-Time Performance Test

As the simulator can be the obvious bottleneck when doing tests for speed, it makes sense to only simulate the smallest unit that is possible, a single Junction. For this test, the Simple 2Stage Junction map was used to show this.

The map itself consists of a single junction, which contains three phases. The clearance interval is set to 3 seconds, all phases are set to a default of 30 seconds of greenTime and 3 seconds of amberTime which they will run on the first cycle before being changed by the SMC. The first two phases allow for the opposite sets of signal groups to be active and a third to allow the flow of traffic on the filter lights.



```xml
<?xml version="1.0"?>
<junctions>
    <junction id="2" clearance="3">
        <phases>
            <phase id="0" name="A">
                <signalGroups>
                    <signalGroup id="1"/>
                    <signalGroup id="4"/>
                </signalGroups>
                <detectors>
                    <detector id="1"/>
                    <detector id="4"/>
                </detectors>
            </phase>
            <phase id="1" name="B">
            <phase id="2" name="C">
        </phases>
    </junction>
</junctions>
```

```xml
<?xml version="1.0" encoding="utf-8" ?>
<config>
    <path>C:\\Users\\Colin\\Documents\\My Dropbo
    <inpfile>TwoStages.inp</inpfile>
    <phaseData>TwoStages-PhaseData.xml</phaseDat

    <speed>1</speed>
    <controllerFreq>1</controllerFreq>
    <resolution>5</resolution>
    <simulationTime>5000</simulationTime>

</config>
```

Figure 6.1: The set of configuration data for a simple junction.

In this test, the *speed* tag in the config XML file is ignored in order to allow the simulator to run as quickly as possible. The frequency at which the controllers sense is set to one second, the resolution of the simulation to five steps per-simulation second and the simulation time to 5000 seconds. Taking a record of the average simulation speed and the average amount of time taken to send off an EndOfPhaseSimEvent to receive a new UpdatePhaseSimEvent, the results show that the maximum speed the simulation can be run at is **7.4** times normal speed with an average of **5.9**.

This is more than is required to run simulations in real-time, as we are looking for a value larger than one, but the values do show the overhead of using the COM interface and the Vissim abstraction for running simulations, as a normal simulation will run at up to 40 times normal speed. To avoid a complete halting of the simulation whenever a SimEvent needs to be sent to the SMC and time taken to receive the Update, the simulator is threaded to use the next current active phase, which if not updated in time will be the phaseData from the last cycle. This effect shows that for very fast simulation there is the possibility of the junction's phase data being off by one cycle, which is not really a problem for the SMC as the snapshot of data that was sent to the SMC is consistent with that point in time.

6.1.4 Fault-Tolerance

The problem with fault-tolerance is that it has to be handled on a per-controller basis. As SCATS itself exists as a three tier system in the real world, the designers in the Road Traffic Authority, New South Wales had to consider fault-tolerance when they were designing the system. As a result the Junction Controllers have the concept of background phases built-in, which are used as a fall back mechanism should the link between the Junction Controller (JC) and the Regional Manager (RM) go down at any point. So although the implementation of SCATS on top of the UCF is fault-tolerant, it is unwise to say that any UTC algorithm you can think of is as well. The eventInterface is capable of throwing a FaultSimEvent back to the ThreadPool for the JC, it is just using this SimEvent in the overridden processSimEvent() method in the JC itself wisely that allows for fault tolerance.

On the SCATS Main Controller (SMC), the fault of an instance will be resolved by simply starting another one in its place. As the SMC do not keep persistent data locally and instead use the Amazon's Simple Storage Service (S3) and SimpleDB for long term data storage, this is not much of a problem. SimpleDB and S3 themselves are distributed systems and fault-tolerance on their end is handled by using active replication of resources. This can cause consistency issues, with the possibility of getting a stale copy of data if accessed too quickly. However it has internal mechanisms for combatting this. Fault-tolerance is also related to Reliability, this is explained above.

6.1.5 Security

Security is taken very seriously in Amazon, first with low-profile data centres, military grade beaming and two security checkpoints to get into any server room. Taking physical security out of the equation, it must be ensured that data communications between between a JC and MC are kept secure and consistent.

The first step taken by Amazon Web Services (AWS) for their components such as S3, SimpleDB, SQS is that to access these web services you need your amazon access and secret key. This is used as an authentication mechanism for accessing objects and buckets in S3, getting and putting in SimpleDB and sending/retrieving and deleting events from SQS. The transport of data from these web services use Transport Layer Security in the form of HTTPS on port 443. As for the consistency of the data as it travels across a network, the include an MD5 hash of the entire object which is being passed so the client can check this first to ensure the data is safe.

The same ideas have been applied to the custom web service between the SJC and the SMC, with the exception of the MD5 for consistency checking. As the UTCElements are serialized in the EventInterface before it is sent off (the web service received SimEvents as serialized strings and are then de-serialized when received), the addition of an MD5 hash to this would be quite easy to add on but time constraints prevented this from showing up in the implementation.

6.1.6 Cost

Yet another concern of feasibility is that of cost. In this section there are two types of cost that are appropriate to answer the question of using SMC to provide Traffic Control as a Service (TCaaS) and as a Traffic Simuation as a Service (TSaaS).

Based on the findings of the Scalability Test B, we can work under the facts that a single Windows instance will cost 0.25 per hour used. To ensure the availability of this service we will need two Availability Zones for traffic control, this means for a city the size of Dublin, which contains some 750 SCATS enabled Junctions, there would need to be at least 4 instances running to provide a feasible SMC. This adds up to approximately 8500 + the bandwidth usage of S3 and SimpleDB to control a city the size of Dublin. A final figure would be substantially less than 10,000 per year to run.

In doing so, the Dublin City Council would not have to worry about the maintenence of the servers, the management of them and the power they consume currently. It must be noted that this is most certainly not a silver bullet answer as there a lot of logisitical problems that must still be answered. This is merely used as an example.

From a TSaaS perspective, a service could be set-up (similar to the ASP model) in which the user could download a client for sending their traffic data to the TSaaS and will be billed accordingly.

Chapter 7

Conclusion

This section summarizes the work that was carried out to achieve the objectives set out by this dissertation. Following this, a list of users who might be interested in the implementation of the UCF and why, concluding with a list of ideas that can be implemented to move the UCF project forward.

7.1 Achievements

The main objective of this dissertation was to answer the question of whether an implementation of an Urban Traffic Control System is feasible on a Cloud Computing infrastructure. As a test case, an implementation of P.R Lowrie's SCATS algorithms [46] was used to show this. In the search of the definition of Cloud Computing, the concept of Everything as a Service arose, leading to the conclusion that an Infrastructure as a Service (IaaS) was the best tier to use for design. Going in search for an IaaS capable of meeting my needs, Amazon Web Services demonstrated itself as being more than suitable for an implementation of a UTC algorithm.

The achievements of this dissertation are outlined as follows:

- The UCF provides a very genericized platform for writing new urban traffic controllers.
- The entire framework and implementation is written in one language, C# on top of Microsoft Windows Server 2003, the .Net Framework 3.5 and IIS 7.0.
- The controllers, and indeed the users, do not need to know any details of the environment it's controlling. This means that you do not have to write your algorithms with VISSIM, ITSUMO or the real world in mind.
- The implementation of SMC on the cloud looks promising in terms of being highly scalable, secure, reliable and faster than real-time. Although it must be noted that fault-tolerance is inherently built into SCATS, so other algorithms must be written with this in mind.

7.2 Potential Usage

There are two main categories of people which would benefit from the work carried out in this dissertation. These are:

- Urban Traffic Control Researchers.
- Parties interested in Traffic Control as a Service (TCaaS).

Urban Traffic Control Researchers will benefit from using the UCF as a platform for building and testing their own traffic control algorithms. As the main controller is capable of any number of simulators without problems due to its design, it can be used to test many different cases at once using the same control algorithm. The researcher also only has to think about their controller's problem in UTC terminology instead of the new terminology that is introduced in traffic simulators like VISSIM. This cuts the learning curve and allows the reasearcher to focus completely on the control algorithm. With this implementation, there are licensing issues with using VISSIM on the cloud as it must be able to access a dongle server which allows it to run. An alternative would be to go in search of a different simulator and write an interface from it to the UCF; this will allow for traffic simulation on the cloud.

For parties interested in Traffic Control as a Service (TCaaS), the use of higher bandwidth communications such as 3G/GPRS or Fibre between the Junction Controller and the Main Controller allows traffic to be controlled from the cloud. This means any number of cities in close enough proximity of a cloud (like Amazon Web Services, or even a custom central cloud for a region) allows for every city to use this service. This allows for the aggregation of massive amounts of live and archived data, which could in turn be used for further research. As well, the algorithm by which the traffic is controller could be switched depending on which is most suitable for the current conditions.

7.3 Future Work

There are several features that have not been implemented in the current state of the Urban Cloud Framework. The UCF is left very open-ended as there can be a lot done with a Framework that allows Traffic Simulations and Control in a centralised place.

7.3.1 Loosen the coupling between UCF and VISSIM

In the UCF's current state, the notion of a generic UTC.UTCSimulator object that VISSIM would extend is not currently implemented. Building this would allow for any simulator to be plugged into the UCF in a more modular way (i.e, just by loading a DLL representing the simulatiors component, in our case the VISSIM_Component). The current state of the UCF is very tightly coupled with some of Vissim's objects, such as the SignalGroup ID's in the phaseData.xml file for each map. SignalGroups, although representing groups of signals that are active together, are unique to VISSIM and to ensure a completely modular implementation of UCF a generic class for these should be made.

7.3.2 Dynamic Loading of Controller DLL's

In the current implementation the loading of DLLs is very static in the sense that the SCATS Junction Controller class which extends the UTCJunctionController is defined in the VISSIM_Wrapper class. An approach that is more in-step with the design ideology with this framework would be to instead use an instantiation of a generic UTCJunctionController which is looked up at runtime (perhaps using Reflection). This would be very desireable to the UCF, as the source code would perhaps not be available to everyone and to move forward with the notion of Traffic Simulation as a Service (TSaaS) and Traffic Control as a Service (TCaaS).

7.3.3 A Simulation Queue

From the perspective of it being used as a simulation framework (or TSaaS), the introduction of a queued job system, perhaps integrating it into Amazon's Simple Queue Service and S3 for results, would be a another useful challenge that is unfortunately out of the scope of this dissertation. Although the initial state of the Junctions is stored in S3 for every simulation, more data about the simulation, in fact an entire trace of the simulation, can be stored in S3 or SimpleDB for later analysis. The inclusion of a SimulationID Guid in the abstract SimEvent class is exactly what this was intended for, as it allows for data to be aggregated on a per-simulation basis.

Appendix A

Abbreviations

Short Term	**Expanded Term**
3G	3rd Generation (International Mobile Telecommunications-2000)
AMI	Amazon Machine Image
APN	Access Point Name
AWS	Amazon Web Services
CL	Cycle Length
CMS	Control Management Centre
COM	Component Object Model
DDOS	Distributed Denial of Service
DLL	Dynamic Link Library
DS	Degree of Saturation
DS0	Digital Signal 0
EC2	Elastic Computing Cloud
ELB	Elastic Load Balancer
GPRS	General Packet Radio Service
HPC	High Performance Computing
IaaS	Infrastructure as a Service
J2C	Junction to Controller
JC	Junction Controller
JCU	Junction Controller Unit
JDO	Java Data Objects
MC	Main Controller
MITM	Man in the Middle Attack
MPI	Message Passing Interface
OV	Original Volume
PaaS	Platform as a Service
RCL	Recommended Cycle Length
REST	Representational State Transfer

Short Term	**Expanded Term**
RM	Regional Manager
S2J	Simulator to Junction
SaaS	Software as a Service
SCATS	Sydney Co-ordinated Adaptive Traffic System
SCL	Stopper Cycle Length
SCOOT	Split Cycle Offset Optimisation Technique
SJC	Scats Junction Controller
SLA	Service-Level Agreement
SMC	SCATS Main Controller
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SQS	Simple Queue Service
TCaaS	Traffic Control as a Service
TLS	Transport Layer Security
TSaaS	Traffic Simulation as a Service
UCF	Urban Cloud Framework
UTC	Urban Traffic Control
VK	Reconstituted Volume
WSDL	Web Service Description Language
XML	Extensible Markup Language

Bibliography

[1] Traffic Control Systems Handbook, FHWA-SA-95-032. Federal Highway Administration. 1996.

[2] Nagel K., Schrekenberg M. 1992. A Cellular Automaton model for Freeway Traffic, Journal Physics France, pp. 2221–2229.

[3] Dia, H. 2002. An agent-based approach to modelling driver route choice behavior under the influence of real-time information. Transportation Research Part C: Emerging Technologies, 10-5/6:331-349.

[4] Lighthill, M. J. and Whitham G. B. (1955). On kinematic waves: Ii. a theory of traffic flow on long crowded roads. Proceeding of the Royal Society A, 229:317-345.

[5] Weiss, A. 2007. Computing in the clouds. netWorker 11, 4 (Dec. 2007), 16-25. DOI= `http://doi.acm.org/10.1145/1327512.1327513`

[6] 2009. Cloud Computing: An Overview. Queue 7, 5 (Jun. 2009), 3-4. DOI= `http://doi.acm.org/10.1145/1538947.1554608`

[7] Geelan J. (2009), Cloud Computing Journal, Twenty-One Experts Define Cloud Computing, `http://cloudcomputing.sys-con.com/node/612375`

[8] Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J., and Brandic, I. 2009. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Gener. Comput. Syst. 25, 6 (Jun. 2009), 599-616. DOI= http://dx.doi.org/10.1016/j.future.2008.12.001

[9] The Definition of Utility Computing, Oxford English Dictionary

[10] Dupre, Frank. (2008), Utility (Cloud) Computing...Flashback to 1961 Prof. John McCarthy, `http://computinginthecloud.wordpress.com/2008/09/25/utility-cloud-computingflashback-to-1961-prof-john-mccarthy/`

[11] Web archive (2006), About NetCentric, `http://web.archive.org/web/20060209160435/http://www.netcentric.com/`

[12] Gruener, Wolfgang. (2008), Tom's Hardware, Dell's Cloud Computing Trademark Application Criticized, http://www.tomshardware.com/news/Dell-cloud-computing,6049.html

[13] Markoff, John. (2001), The New York Times, Internet Critic Takes on Microsoft, http://www.nytimes.com/2001/04/09/technology/09HAIL.html

[14] Sullivan, Danny. (2006), Search Engine Strategies Conference, Conversation with Eric Schmidt hosted by Danny Sullivan, http://www.google.com/press/podium/ses2006.html

[15] 3tera Company, About, Company Section, http://www.3tera.com/Company/

[16] Jones, M. Tim (2008), Cloud Computing with Linux, Cloud computing platforms and applications, http://www.ibm.com/developerworks/linux/library/l-cloud-computing/

[17] Chapell, D. 2008. A Short Introduction to Cloud Platforms, An enterprise-oriented view, http://www.davidchappell.com/CloudPlatforms--Chappell.pdf

[18] Brodkin, Jon. (2009), Cloud Interoperability Remains Wispy but Progress Being Made, http://www.cio.com/article/496610/Cloud_Interoperability_Remains_Wispy_but_Progress_Being_Made

[19] Distributed Management Task Force (2009), Open Cloud Standards Incubator Charter, http://www.dmtf.org/about/cloud-incubator/CloudIncubatorCharter2009-04-16.pdf

[20] Open Cloud Manifesto (2009), Open Cloud Manifesto: Dedicated to the belief that the cloud should be open, http://www.opencloudmanifesto.org/Open%20Cloud%20Manifesto.pdf

[21] Amazon Web Services, What is AWS?, http://aws.amazon.com/what-is-aws/

[22] Palankar, M. R., Iamnitchi, A., Ripeanu, M., and Garfinkel, S. 2008. Amazon S3 for science grids: a viable solution?. In Proceedings of the 2008 international Workshop on Data-Aware Distributed Computing (Boston, MA, USA, June 24 - 24, 2008). DADC '08. ACM, New York, NY, 55-64. DOI=http://doi.acm.org/10.1145/1383519.1383526

[23] Amazon Elastic Compute Cloud (2009), Getting Started Guide, API Version 2009-04-04, http://awsdocs.s3.amazonaws.com/EC2/2009-04-04/ec2-gsg-2009-04-04.pdf

[24] Amazon Elastic Compute Cloud (2009), Developers Guide, API Version 2009-04-04, http://awsdocs.s3.amazonaws.com/EC2/2009-04-04/ec2-dg-2009-04-04.pdf

[25] Hazelhurst, S. 2008. Scientific computing using virtual high-performance computing: a case study using the Amazon elastic computing cloud. In Proceedings of the 2008 Annual Research Conference of the South African institute of Computer Scientists and information Technologists on IT Research in Developing Countries: Riding the Wave of Technology (Wilderness, South Africa, October 06 - 08, 2008). SAICSIT '08, vol. 338. ACM, New York, NY, 94-103. DOI=http://doi.acm.org/10.1145/1456659.1456671

[26] The Xen hypervisor website, What is Xen?, http://www.xen.org

[27] Amazon Simple Storage Service (2009), Getting Started Guide, API Version 2009-04-04, http://awsdocs.s3.amazonaws.com/S3/20060301/s3-gsg-20060301.pdf

[28] Amazon Simple Storage Service (2009), Developers Guide, API Version 2009-04-04, http://awsdocs.s3.amazonaws.com/S3/20060301/s3-dg-20060301.pdf

[29] Amazon Simple DB (2009), Getting Started Guide, API Version 2009-04-15, http://awsdocs.s3.amazonaws.com/SDB/2009-04-15/sdb-gsg-2009-04-15.pdf

[30] Amazon Simple DB (2009), Developers Guide, API Version 2009-04-15, http://awsdocs.s3.amazonaws.com/SDB/2009-04-15/sdb-dg-2009-04-15.pdf

[31] Amazon Elastic MapReduce (2009), Getting Started Guide, API Version 2009-03-31, http://awsdocs.s3.amazonaws.com/ElasticMapReduce/20090331/emr-gsg-20090331.pdf

[32] Amazon Elastic MapReduce (2009), Developers Guide, API Version 2009-03-31, http://awsdocs.s3.amazonaws.com/ElasticMapReduce/20090331/emr-dg-20090331.pdf

[33] Apache Hadoop Framework Site, http://hadoop.apache.org/

[34] Amazon CloudWatch (2009), Developers Guide, http://awsdocs.s3.amazonaws.com/AmazonCloudWatch/latest/acw-dg.pdf

[35] Amazon Auto Scaling (2009), Developers Guide, http://awsdocs.s3.amazonaws.com/AutoScaling/latest/as-dg.pdf

[36] Amazon Elastic Load Balancer (2009), Developers Guide, http://awsdocs.s3.amazonaws.com/ElasticLoadBalancing/latest/elb-dg.pdf

[37] Amazon Simple Queuing Service (2009), Developers Guide, http://awsdocs.s3.amazonaws.com/SQS/latest/sqs-dg.pdf

[38] Amazon Web Services (2008), Overview of Security Processes, http://s3.amazonaws.com/aws_blog/AWS_Security_Whitepaper_2008_09.pdf

[39] Google AppEngine Developer Guide: What is Google AppEngine? http://code.google.com/appengine/docs/whatisgoogleappengine.html

[40] Google AppEngine Developer Guide: Datastore http://code.google.com/appengine/docs/python/datastore/

[41] Schofield, J., Google angles for business users with ‘platform as a service’, The Guardian, http://www.guardian.co.uk/technology/2008/apr/17/google.software

[42] What is the Windows Azure Platform?, http://www.microsoft.com/azure/whatisazure.mspx

[43] Sims, A.G.; Dobinson, K.W., ”The Sydney coordinated adaptive traffic (SCAT) system philosophy and benefits,” Vehicular Technology, IEEE Transactions on , vol.29, no.2, pp. 130-137, May 1980 URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1622746&isnumber=34047

[44] Aldridge Traffic Controllers Website, Dublin City Council SCATS Contract Awarded (2009), http://www.aldridgetrafficcontrollers.com.au/Breaking-News/Dublin-City-Council-SCATS-Contract-Awarded/default.aspx

[45] M. Dineen, Real-time display of Dublin traffic information on the web, Department of Computer Science, University of Dublin, Trinity College, Ireland, M.Sc. Thesis, September 2000

[46] Lowrie, P.R., The Sydney Co-ordinated Adaptive Traffic System principles, methodology, algorithms, Proc. IEEE International Conference on Road Traffic Signalling, London, pp67-70, 1982.

[47] Traficon, Traffic Video Detection Cameras Website, http://www.traficon.be/

[48] SCATS 6, Functional Description Manual, http://www.aldridgetrafficcontrollers.com.au/ArticleDocuments/47/ATC_An_Introduction_To_The_New_Generation_Scats_6%20_5_%203_.pdf.aspx

[49] Roads and Traffic Authority of New South Wales, Austrailia, ”SCATS Message Formats”,(RTA-TC-226),June 1999

[50] Fehon, K., Chong, R., Black, J., Adaptive Traffic Signal System for Cupertino, California, April 2003

[51] Hunt P.B., Robertson D.I., Bretherton R.D., Royle M.C., The SCOOT On-Line Traffic Signal Optimisation Technique, International Conference on Road Traffic Signalling, IEE, pp.59-62. London, UK, 1982.

[52] The SCOOT Urban Traffic Control System Website, http://www.scoot-utc.com/

[53] VISSIM 5.10-03 COM Interface Manual, PTV Vision, 2008