A Flexible Network Simulator for Multiple Server Virtual World Systems

Kevin Pulo

kev@cs.usyd.edu.au

January 8, 2001

Supervisor: Dr Michael E. Houle

Basser Department of Computer Science

The University of Sydney

Submitted in partial fulfillment of the requirements

for the degree of Bachelor of Science (Adv), (Honours)

Contents

Abstract

A *virtual world* is an artificial environment, created inside a computer, which mimics some aspect of the real world. Virtual worlds are often *multiuser*, which means that many people may be present in the virtual world simultaneously and may interact with each another and the virtual environment. With the advent of the global Internet connectivity, users from all over the real world may participate in multiuser virtual worlds and interact without any regard for geographic boundaries.

However, achieving believable realism is not easy and there are several factors which can hinder a user's virtual world experience. In order to avoid these kinds of problems and provide the best possible experience, the underlying network must be designed and implemented carefully.

This thesis presents a tool which can guide the design and implementation of virtual world systems to avoid potential problems. This is achieved by simulating the system in various scenarios and evaluating the performance of the system and the potential solutions.

Acknowledgements

I would like to thank my supervisor, Michael Houle for his enthusiasm and invaluable help throughout this project. It is much appreciated.

Thanks to the Computer Science 2000 Honours year that made hons so enjoyable. Special thanks to Audrey Lobo for her advice and ongoing support throughout the year. Finally, many thanks to my mother, father and brother, whose patience and tireless support have made this year manageable.

Part I

Introduction

1 Introduction

A *virtual world* is an artificial environment, created inside a computer, which mimics some aspect of the real world. They aim to be as realistic as possible, sometimes even to the point that the user becomes unaware that they are interacting with an artificial environment or cannot tell the difference between the virtual and real worlds. The realism is frequently achieved through the use of high resolution 3D graphics displays, but how the virtual world is used is more important than how it looks. If the user is to become completely immersed in the virtual world, it must be very intuitive and feel natural to use.

Next to the realism, the most important aspect of virtual worlds is that they are *multiuser*. This means that many people may be present in the virtual world simultaneously. Just as in the real world, they may interact with each other and with the virtual world, rather than merely interacting with a computer program. With the advent of the global connectivity provided by the Internet, users from all over the real world may become participants in a multiuser virtual world and may interact without any regard for geographic boundaries.

Two of the most popular types of multiuser virtual worlds present on the Internet are games, such as Quake, and chat rooms, such as IRC. Games will often provide fast and visually appealing action, such as combat against other players of the game. Chat rooms provide a means of allowing groups of people to talk to one another, in a similar fashion to a group of people chatting while sitting in a circle. They are usually text-based, rather than graphical, and so are an example of how the use of the virtual world is more important that how real it looks.

However, several factors can hinder the user's virtual world experience. The virtual world may pause for a short duration, disrupting what the user is doing. Such delays are called *lag*, and can be extremely frustrating from the user's perspective because the it can shatter the illusion being presented by the virtual world. A similar problem is when groups of users are accidentally removed from the virtual world, breaking the communication and interactions they are involved in.

In order to avoid these kinds of problems and provide the best possible experience for the users of the virtual world, the underlying network must be designed and implemented carefully. However, doing so is not always straightforward. Virtual world systems can be quite complicated, and as the number of simultaneous users increases, so do the potential problems.

This thesis focuses on a single class of virtual world systems, and presents a tool which can guide the design and implementation of these systems. For the problems which may be encountered, it allows potential solutions to be tested, evaluated and compared. This is achieved by simulating the system in various scenarios and evaluating the performance of the system.

2 Background

An important concept in networking is the *client-server model* [4]. In this model, clients utilise the network to *connect* to servers, creating a communication channel called a *connection* or *link*. Connected clients and servers communicate by transmitting small parcels of information called *packets* or *messages*. The asymmetry between clients and servers arises because clients initiate both the connections and communications with the server. The communications in a strict client-server model consist of clients sending *requests,* and servers responding to requests with *replies*. However, in practice servers often send information to clients which is unsolicited, but nevertheless useful. The client-server relationship is many-to-one, that is, many clients may be connected to a single server.

The client-server model is commonly found in application-level Internet protocols, such as those used for the World Wide Web, file transfers and the sending and receiving of email messages.

Virtual world systems are characterised by client-server systems which are aimed at allowing clients to interact with each other and with a virtual 'environment'. The two main types of virtual world systems are multiplayer Internet games, such as Quake, and multiuser Internet chat rooms, such as IRC.

In terms of server setup, there are three main types of virtual world systems.

1. *Single-server* systems consist of one server which all the clients are connected to.

2. *Multi-server* systems consist of multiple servers collected at a single Internet site.

3. *Distributed-server* systems consist of multiple servers distributed across the Internet.

Each of these systems offer advantages and disadvantages in comparison with the others.

Single-server systems are in general simpler than multi-server and distributed-server systems, and so are easier to implement.

Since single-server and multi-server systems are at a single location, they are generally easier to maintain and administer, and are more secure than distributed-server systems.

The tradeoff to the previous point is that single-server and multi-server systems suffer from a single point of failure. For example, if the site hosting the single-server or multi-servers loses power or its Internet link, then the entire virtual world becomes unavailable. However, it is extremely unlikely that all the servers in a distributed-server system would be unavailable simultaneously, which makes it more robust against these types of problems.

Multi-server and distributed-server systems have the advantage of redundancy. If a server is lost (due to an operating system crash, for instance), only the clients connected to that server are affected, while the rest of the virtual world may continue. It may even be possible to automatically add a spare replacement server within minutes of the original server loss.

In a single-server system, client interactions travel via the server. However, in multi-server and distributed-server systems, client interactions must be sent via more than one server if the clients are on different servers. This additional communication is said to be *interserver*, and it adds *latency* to the interactions. Latency is the time delay involved in the transmission of a packet, and so lower latencies are desirable. Since the network links between distributed-servers are usually Internet links, they are generally of higher latency than in multi-server systems which may employ expensive high-speed hardware to lower the interserver latency.

Distributed-server systems have the advantage that servers may be placed near the clients to lower the latency of messages between the clients and servers. However, the latency from clients to the single-server and multi-server site will on average be higher than well placed distributed servers.

Another advantage of multi-server and distributed-server systems is in the area of *scalability*. This is how well a server performs as its *workload* increases towards infinity, where the workload is usually related to the number and rate of clients being served. It is important during times of peak usage and because the ever-growing size of the Internet means that the number of clients will generally increase. The client limit of single-server systems is ultimately limited by the amount of hardware the single server machine can have. Multi-server and distributed-server systems do not have this limitation, as new servers can be added to handle extra clients, perhaps even without disrupting the virtual world. While this does improve the scalability of the system, it is now limited by the performance of the interserver network. This is because the amount of interserver traffic generally depends upon the number of clients.

As an example, multiplayer action games are usually single-server and limited to tens of clients, while IRC networks often consist of tens of servers and tens of thousands of clients. However, IRC has latencies on the order of seconds, while multiplayer action games typically require latencies less than half a second. In addition, in periods of high usage IRC networks suffer from *netsplits*, where the connection between a pair of servers is broken due to the congestion. These factors justify the choice of virtual world system for multiplayer action games and IRC chat networks.

In general, each client in a virtual world will not want to interact or communicate with every other client, particularly in large virtual worlds. The human user operating a client will usually find it impractical to deal with more than a small number of other clients at any given time. If these small groups of communicating clients can be arranged such that most of them are on the same server, then the performance and scalability of the system can be improved. This idea of increasing the performance of the system by exploiting the localisation of clients is the major motivation behind the project.

The *neighbourhood* of a client c is the subset of the clients which c may interact with. How strictly this definition is interpreted is not important; for example it may be weakened slightly to be only the clients c is expected to interact with. Performance gains occur when the neighbourhood is a strict subset of the clients, due to the smaller amount of interserver traffic.

In multi-server and distributed-server systems, each client is *assigned* to a particular server which manages the communications with that client. The act of assigning a client to a server is called an *assignment*. The collection of assignments for all clients is called an *allocation*, and refers to the global pattern of individual client assignments. This is because it refers less to where an individual client is assigned, and more to how the resources of the servers have been partitioned among the clients. An allocation is *static* if client assignments are fixed when clients connect, and are *dynamic* if client assignments are free to change while the client is connected. An *allocation strategy* is an algorithm for creating allocations; that is, it decides how each client is to be assigned. A *network scenario* refers to a particular situation in the network, specifically, a given setup of servers, clients, neighbourhoods and allocation, and how this varies over some time period.

3 Project Aims

3.1 Problems addressed

The project deals with multi-server and distributed-server systems and addresses two major problems related to finding allocation strategies which perform well. These problems are:

- The method for comparing allocation strategies and deciding which performs 'better' is not obvious.
- Developing good algorithms for allocation strategies is in general a difficult task.

3.1.1 Comparing allocation strategies

One possible method for measuring and comparing the performance of allocation strategies is an analytic analysis of allocation strategies. This would probably be in terms of the statistical nature of the input network scenarios. It would give good theoretical expectations of the strategies, but no real experimental results.

Another method is to actually implement the allocation strategies, either in the network itself or in a simulation, and then run both with 'expected' network scenarios. The amount of interserver latency incurred, for example, could be measured and compared directly. This would require some method of ensuring that the scenarios used for each strategy are sufficiently similar, such that any differences would not cause discrepancies in the interserver latencies obtained. Of course, the interserver latency may not always be the best measure of the performance of a particular allocation strategy. It would be useful to be able to evaluate and compare allocation strategies based on any metric, not just the interserver latency.

It is often the case that allocation strategies will perform well in certain types of scenarios and poorly in others. Thus it would be useful if strategies could be compared on a large number of 'typical' scenarios, perhaps randomly generated from some 'main' scenario.

3.1.2 Developing allocation strategies

Many problems presented by multi-server and distributed-server systems can be expressed as optimisation problems. These require that some quantity be optimised in addition to preserving some additional constraints. An example of a multi-server or distributed-server optimisation problem is to find an allocation which minimises the amount of interserver latency.

These problems are generally NP-hard, which means that any practical advances in solving them are made with *heuristics*. A heuristic is a general guide or rule for solving a problem, rather than an actual algorithm. Finding high quality and efficient heuristics is usually difficult because unlike algorithms, heuristics cannot be proved to be optimal. Instead, the quality of their optimisation must be ascertained experimentally by actually implementing and running the heuristic. This experimental step is also important in improving heuristics and developing new, better heuristics.

As such, the project should support the experimentation of heuristics for solving the various multi-server and distributed-server optimisation problems.

3.2 Aims of the project

This thesis presents a flexible simulation tool which facilitates the investigation into, and experimentation of multi-server and distributed-world systems. The simulation is a discrete-event simulation with a graphical interface. It is of sufficient generality and modularity that a wide range of systems can be modeled with a small amount of development effort. The modular design makes it extensible, so that new allocation strategies, neighbourhoods and cost metrics are easily created and evaluated. It also allows investigation into many different factors contributing to the performance of multi-server and distributed-server virtual world systems. These two benefits allow it to be applied to a wide range of multi-server and distributed-server virtual world systems and problems they face.

Throughout the thesis, these general features are applied to a realistic architecture for a large multi-server Internet game. The concentration is on the problem of minimising the interserver traffic for this given architecture.

Part II

Design

4 Software Requirements

The simulation tool is a discrete event simulation and as such it supports the operations present in these types of simulations. The most important of these operations is to efficiently process sparse events independently of the time periods between them, and it allows events to schedule new events, possibly causing a cascade of events.

The tool itself has several fundamental requirements:

1. The tool should allow the user to investigate and experiment with the performance of the simulation in various scenarios. These scenarios include:
   - Various arrangements of players and their behaviours.
   - Various client, server and other hardware configurations.
   - Various methods of dynamically allocating clients to servers.
   - Various methods of informing clients of the events occurring in the game.
2. The previous point implies that the tool must be able to effectively "score" the state of the simulation, and the outcome of the entire simulation, in order to effectively compare the performance of different simulation scenarios.
3. The tool should allow the use of "real-world" input, such as data recorded from actual gameplay. This will ensure that the user can be satisfied that the results are applicable to the actual game.
4. In addition to testing any already existing dynamic allocation strategies, the tool should support the development of new allocation strategies.

To support these fundamental requirements, the design of the simulation has several specific requirements.

1. The simulation should graphically show the clients and their relationships with each other, servers, and any other noteworthy game entities. Interactivity, though not necessary, would be a useful addition, because it is more suited to experimentation involving small changes in parts of scenarios, rather than on entire scenarios.

2. The simulation should be reasonably fast, preferably running much faster than realtime. This dictates several architecture choices, notably that a compiled language should be used.

3. The design must be modular and extensible, allowing components to be easily inserted and selected. This is required to support the experimentation of the different components, and the development of new ones. Object oriented languages support this idea well.

5 Overview

The simulation consists of many different parts, many of them interrelated. This section is presented as a brief overview of the entire system and subsequent sections will discuss the simulation in more detail.

Instances of entities exist in the simulation, and correspond directly to actual things in the real game. Entities are detailed in Section 9. There are three types of entities.

- Clients represent the machines that are connected to the game network, on which players are playing the game. The terms 'client' and 'player' are often used interchangeably, even though they are technically different — the client is the game program, running on the player's computer, and the player is the human using the client to play the game. Clients in the simulation store a 'player state', which encapsulates the player's location in the game and allows the location to be estimated.

- Servers represent the machines which collectively control the game, to which clients are connected. Servers maintain the state of the game world, allowing players to do things in the game and notifying them when things happen in the game. For example, all interactions between players are mediated by the servers. This ensures that no rules of the game are broken and allows the servers to inform other players of the interaction, among other things.

- Packets, sometimes called messages, represent small, atomic collections of information transferred between server and client entities. They are the only method of communication between clients and servers and approximately correspond to the actual networking packets transferred.

Entities are operated on by instantaneous events in the simulation. There are also various types of events, each performing different operations. The main events are to create and destroy entities, and to send and receive packet entities. Events are detailed in full in Section 8.

Events are "recorded" and stored in plain text logfiles, each of which encapsulates some aspect of an entire gaming session. This means that the types of information stored in them, such as events, entity identifiers, typenames, player states, costs, etc, each have an unambiguous plain text representation. Logfiles are detailed in Section 10. There are three different types of logfiles, characterised by their purpose and the types of events they can store.

- Player state logfiles record the events generated by the players as they play the game and are a complete record of their actions. These logfiles are obtained either by storing the events occurring in an actual game being played by human players, or by simulating such a game with a program that implements 'robot' players, or bots.

- Architecture logfiles record the architecture on which a game took place, for instance, the servers, network configuration, latencies, and so on. They are useful for testing how different architectures perform with a given player state logfile.

- Allocation logfiles record the assignment packets, which indicate the allocation of clients to servers throughout the simulation. Since they record only the assignments, they are a good way of comparing allocation strategies without revealing their details or algorithms.

An allocation strategy encapsulates a method for assigning clients to servers throughout the course of the simulation. The object-oriented design allows new strategies to be created and easily tested. Allocation strategies are detailed in Section 11.

A problem faced by both the actual game and the simulation is how to inform each client of the activities of the other clients in the game. Clients are distributed across the Internet and are expected to be connected to the Internet via dialup modems. This is a bottleneck in the link between the client and server, and means that clients lack the bandwidth to be told the activities of all the other clients. The direct implication of this is that each client is notified of the actions of a subset of the clients. This subset is called the 'neighbourhood' of the client and its member clients are the client's 'neighbours' or 'neighbourhood clients'. This subset should be chosen to be clients which are likely to interact with the client in the near future.

The concept of communication strategies solve this problem in a similar method to allocation strategies. Each encapsulates a method of choosing client neighbourhoods, and the design allows easy switching between them. The communication strategies used in the project choose neighbourhoods to be:

- The clients closer than a particular distance away from the given client.

- The u nearest neighbours of the given client (for some particular value of u).

Of these two, the former is expected to be more realistic. Communication strategies are detailed in Section 12.

Throughout the simulation, various operations and actions incur penalties, called *costs*. These costs are stored as a set of coefficients, where any particular cost is evaluated in terms of these coefficients. This means that it is easy to represent the cost of assigning a client to a server being proportional to the number of clients already on the server. This gives costs the flexibility to be used for almost any action in the simulation. Costs are detailed in Section7.

6 Basic Definitions and Data Types

This section introduces the basic data types, units, quantities and many of the terms and definitions which are used throughout the project.

6.1 Arena

The *arena* is the virtual world in which the game is played, the playing field. Players move around within the arena at various speeds, and can see their surroundings, including any other nearby players which are also in that area.

The size of the arena is determined by the game being played, and if this is not known, then the simulation should attempt to automatically resize the arena to accommodate all the clients in the game.

In the actual game, the arena is 3-dimensional. In the simulation the arena is the 2-dimensional projection of the real arena onto the ground. This is acceptable because most of the game is played near ground level, and using a 3-dimensional arena would complicate much of the simulation and algorithms to be used, for no real benefit. Points in the arena are 2-dimensional, with real-valued, Cartesian coordinates. Similarly, distances between points in the arena are also real-valued. The project uses C `doubles` (32-bit, signed) for coordinates and distances.

When dealing with real numbers, it is often desirable to use rational numbers instead. This stores numbers as a combination of numerator and denominator, in an effort to avoid rounding errors that are common in any limited precision representation of real numbers. However, they were rejected in favour of simpler, standard, fixed accuracy floating point real representations for two main reasons.

1. The set of rationals is a strict subset of the set of reals. It is often the case that positions and distances are irrational, usually due to an operation such as taking the square root, which is common in Euclidean space. These numbers cannot be represented as rationals, and therefore must be approximated, just as real numbers must be.

2. The actual implementation of rational numbers is usually complicated and brings with it a new set of technical problems, for example, having extraordinarily lengthy numerators and denominators can cause performance problems.

All coordinates and distances in the simulation are measured in the units of the decimetre (0.1 metres). This gives a virtually unlimited range and resolution with any floating point data type[1].

The arena is displayed graphically by the simulation in an arena window[2]. This is a schematic top-down view of the arena, with clients drawn as small crosses ('$\times$'). Each server is given its own unique colour, and all the clients are drawn in the colour of the server they are assigned to. Since the size of the arena isn't known *a priori*, the most extreme x and y values are remembered, and the display is resized so that its boundary remains 10% outside these values.

[1]Specifically, a range on the order of 10^{600} (that is, minimum and maximum values of $\pm 10^{300}$), and 15 decimal points of accuracy, that is, resolution of 10^{-15}.

[2]Naturally, this can be disabled for batch simulations, where the simulation is run without user-intervention to obtain its final outcome.

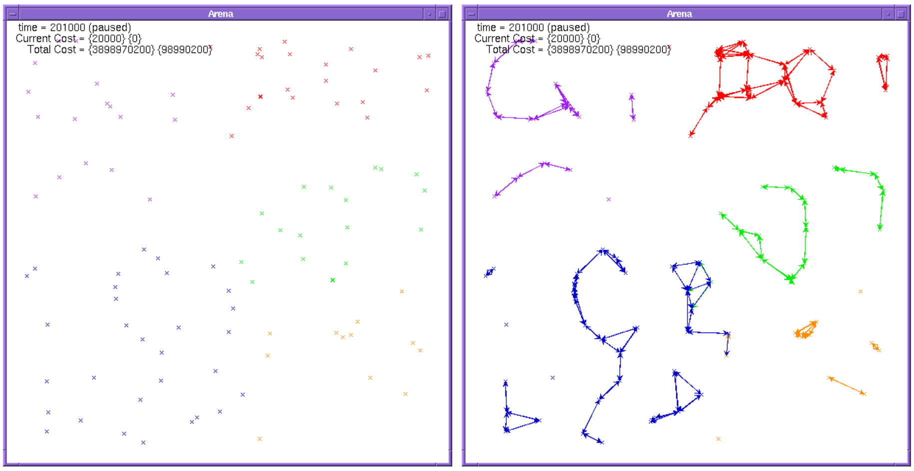


Figure 1: An example of the arena window, both with and without the neighbourhood edges.

If desired, the neighbourhoods of all clients can be drawn, clearly showing the overall effect of the selected communication strategy. The top left hand corner has a small status display, showing the current time and the current and total costs. Figure 1 shows an example of the arena window, both with and without the neighbourhood display.

6.2 Network topology

A possible realistic network topology is illustrated in Figure 2, and is the network topology which this thesis concentrates on. It is three-tiered, consisting of a standard client-server setup, with intermediary Front End Processors (FEPs) between the clients and servers. The FEPs reside on the same high-performance LAN which all the servers are on, which is connected to the Internet where the clients connect from. This LAN is referred to explicitly as the 'server LAN', and implicitly as 'the LAN', as it is the only LAN in the simulation. FEPs exist to serve as standard single points of connection for the clients. They allow clients to connect to a single FEP, but still be dynamically assigned to any server, avoiding the high cost associated that would be occur if clients were to reconnect themselves dynamically to the different servers.

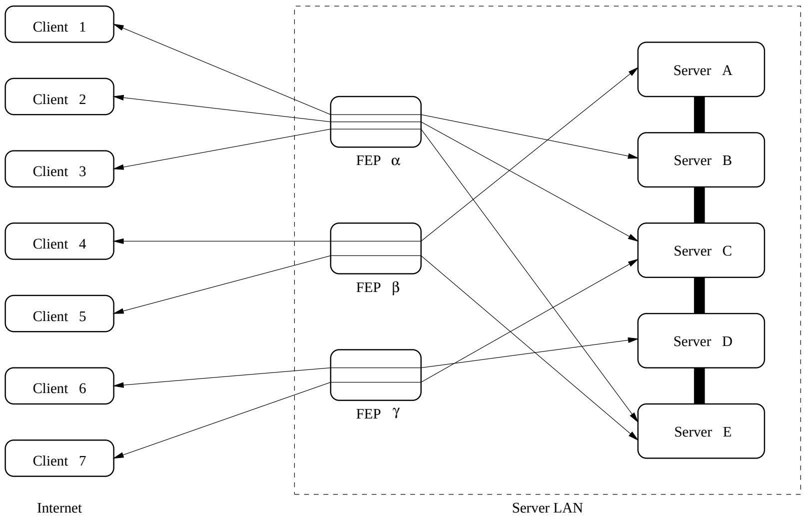


Figure 2: The actual network topology. Clients 1–7 are connected to Servers A–E via FEPs α–γ.

The LAN is expected to be very high performance, for example, switched Fast Ethernet (100Mbit/s) or Gigabit (1000Mbit/s), whereas clients are expected to be on (at worst) dialup modem lines (∼56kbit/s). We assume that all of this client bandwidth is available with very little additional latency, as most realtime multiplayer Internet games do.

Messages sent by clients to their assigned server are said to be 'upstream'. Similarly, messages sent by servers to the clients assigned to them are 'downstream'. Most upstream messages are clients informing their servers that their position in the arena has changed. Most downstream messages are servers notifying their clients of the position changes of other clients. Servers and FEPs are all implicitly 'connected' to one another, as they communicate directly on the LAN[3]. Clients, however, can only communicate with each other via their assigned servers. Figure 3 illustrates the path taken for messages between clients.

Depending on their available bandwidth, clients will typically send upstream messages at about 10Hz, and receive downstream messages regarding 10 other clients at 10Hz as well.

[3]As a result, the LAN would benefit greatly from being fully switched.

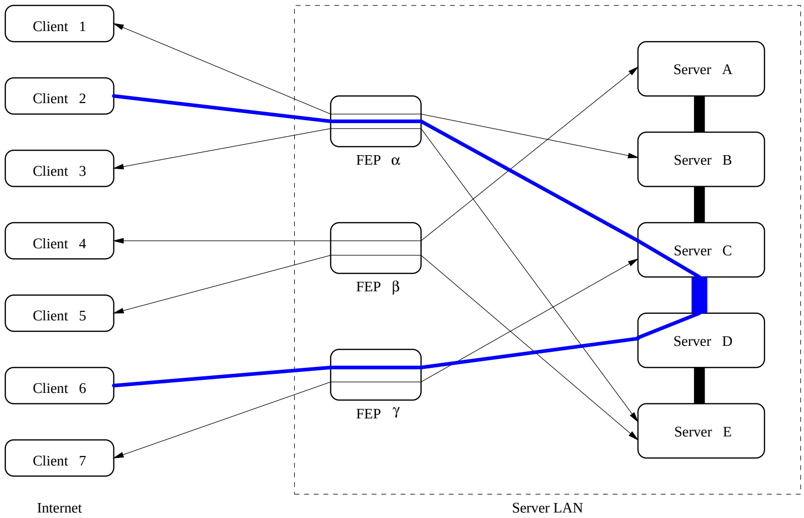


Figure 3: The path taken by messages between clients 2 and 6, via their assigned servers C and D, respectively.

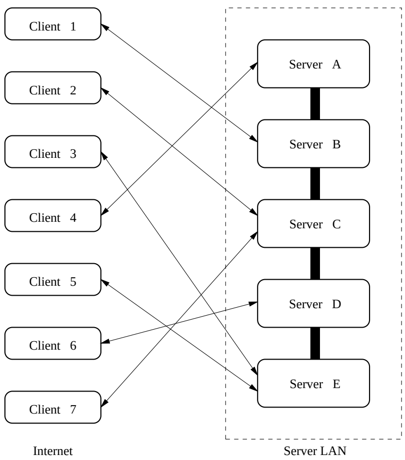


Figure 4: The network topology used by the simulation. Clients 1–7 are connected to Servers A–E with the same relationships as in Figure 2, but the FEPs α–γ have been removed.

The project's network topology differs quite substantially from that described above. Effort has been taken at many levels to avoid the disclosure of which particular server any given client is assigned to, without losing the information about where and when packets are sent, the network load, and so on.

The network is two-tiered, with only servers and clients. Servers are connected to each other via the local server LAN. FEPs have been neglected because for our purposes they are essentially transparent proxies for the clients, and do not affect the amount of traffic sent over the server LAN or the traffic's destination. The cost of traffic between clients and FEPs is neglected, because it is a completely unavoidable cost. However, the cost of traffic between FEPs and servers is recorded, because even though it is also unavoidable, it affects the amount of traffic in the server LAN. Thus, the LAN is directly connected to the Internet at large, rather than (essentially) firewalled by the FEPs, since this security aspect doesn't affect the simulation in any way.

Clients may only send packets to their assigned server. This server is not known explicitly to the packet, even though its receipt by the server is acknowledged. Servers send packets to any client; if a client is assigned to a different server, then the packet must first travel to the client's server, from which point it travels to the client. As the servers are all on the LAN, these inter-server packets travel directly between servers without any intermediate servers.[4] Client to client interactions make up less than approximately 1% of the total traffic to and from clients, meaning that they have only an extremely small effect on the cost of any particular allocation strategy. Thus they have been neglected by the simulation, although the design allows for them to be added.

[4]This requires that either each server know which server to send a given client's packet to, or some form of broadcasting method is used. The former method is formidable, as it would (at first glance) require notifying $O(n)$ servers when a client changes assigned servers. However, the exact problem of deciding which method to use reduces to choosing an appropriate assignment costs, which will be described in Section 9.5.2.

6.3 Time

Points in time are of integer type, to avoid floating-point accuracy problems (particularly with comparisons, which are important in the simulation's priority queue). The start of a simulation is time 0 (the origin against which other times are measured), and so it is tempting to define time as unsigned. However, the difference between two points in time has the same units as the points themselves, thus these durations should also be stored as time variables, and durations may easily be negative. Also, time-based costs (described in Section 7) may be negative. Hence, times are signed.

Next, the resolution and limits of time must be considered. Both affect the size of the integers used to store times. The smaller the resolution, and the larger the limits, the greater the size of integer required. Durations (and time-based costs, Section 7) are easily accumulated, thus the upper limit of time must actually be much larger than the expected maximum point in time.

The two competing time resolutions are milliseconds (1ms $= 10^{-6}$s) and microseconds (1μs $= 10^{-9}$s). A time resolution of 1ms is not considered fine enough, because fast networks routinely deal with sub-millisecond latencies. Further, if many sub-millisecond times are rounded to 0 or 1ms and then added, the accumulated error can be quite large[5].

The two competing integer sizes are 32-bit, a standard `long int` type, and 64-bit, not always supported and the GNU gcc extension `long long` type (see also Section A.1.4).

Table 1 shows the maximum time values possible for these integer sizes and time resolutions. As can be seen, 32-bit integers place quite a limit on the largest possible time or duration value, even at millisecond resolution. However, programming with 64-bit integers can be slow, non-portable and is generally to

[5] For an explicit example, 10000 durations of 10μs each add up to 0.1s. If the 10μs are rounded to 1ms, then they add up to 10s. This error is 2 orders of magnitude, since the rounding from 10μs to 1ms is 2 orders of magnitude.

Rounding sub-millisecond times to 0 may occur when taking the difference of two times which are the same when rounded. In this case, the total is always 0.

Time resolution	Maximum time value	
	32-bit	64-bit
1000μs (1ms)	49 days	584 000 000 years
100μs (0.1ms)	4.9 days	58 400 000 years
10μs (0.01ms)	11.9 hours	5 840 000 years
1μs (0.001ms)	1.19 hours	584 000 years

Table 1: Maximum time and duration values for varying integer size and time resolution.

be avoided if 32-bit integers will suffice.

There are technical issues involved in attempting to record events which occur at sub-millisecond resolution, however it is better to allow recording at sub-millisecond resolution when it isn't possible, than it is to not allow sub-millisecond resolution when it may be possible.

Thus, times and durations are expressed in microseconds (μs) using 64-bit `long long` variables.

6.4 Data

In addition to maintaining time-based details of the simulation, details of the data in the simulation must be maintained. In particular, the traffic in various parts of the network must be maintained, so that, for example, the simulation is accurate during periods of network congestion. Every packet which travels across the network will record its size, in addition to any other parameters it needs to record. This data type is called 'DataSize', is also a 64-bit `long long` integer, and is in units of bytes. It is signed because, like times, DataSizes may also store differences in sizes, and these can be negative. The 64-bit integer size was chosen as a result of similar considerations to those for times, that is, accumulation of many individual data sizes. It gives an effectively infinite upper limit of 8 million terabytes and lower limit of -8 million terabytes.

6.5 Identifiers and Keywords

At any point in time, the simulation may contain many entities of various types. These entities need to communicate with one another; this requires that they somehow identify one another. When the simulation is running, internally it is sufficient to have a reference or pointer variable to another entity, which essentially identifies the entity by its memory location. However, this is not a persistent identification mechanism, and so is useless for identifying entities between instances of the simulation, such as in logfiles. The 'identifier' type is used to identify individual instances of the various other data types in the simulation, usually entities. Since identifiers are used to identify individual instances within the simulation, an individual's identifier must be unique for the lifetime of the individual, to avoid any possible ambiguity.

Given this criteria, identifiers are arbitrarily long strings of alphanumeric characters, with some punctuation also allowed. Specifically, identifiers are those strings which match the Perl-style regular expression '`[A-Za-z0-9_-.,]+`'.

'Keywords' are similar to identifiers, however, their role is closer to that of keywords in programming languages. Keywords are used to identify types in the simulation, rather than individuals. For example, entity and event types are keywords. Since keywords identify individual types, they must also be unique for the entire duration of the simulation, to avoid ambiguity.

6.6 Collections

In addition to standard collection data types such as arrays and linked lists, several collections are used extensively throughout the project.

The 'PointSet' is a collection of points in a 2D plane, and which is optimised for efficient results of geometric queries, such as finding nearest neighbours and range searches.

As described in Section 6.5, a very common operation is to look up the individual of a given identifier, or the type of a keyword. These are implemented using dictionaries, where the keys are identifiers or keywords. The 'IdTable' and 'KeywordTable' types are these generalised dictionaries, with the values being any type desired[6]. IdTables and KeywordTables are the most commonly used collections of objects identified by Identifiers (such as Entities), and types which have Keywords (such as the various Entity types like EntityClient, EntityServer, and so on).

7 Costs

7.1 Introduction

In the course of running the network, actions incur penalties. These penalties are called *costs* and can affect various system resources. Costs are made up of *cost components*, each of a particular type. In the simulation, costs have two components, *TimeCosts* and *DataCosts*. TimeCosts indicate that an action has taken some period of time, or (equivalently) that an action has been delayed by some period of time. DataCosts indicate that an action has caused some network traffic. In general, a cost will have both time and data components, although either or both may be zero.

During the simulation, "current" and "total" costs are maintained. The current cost is a measure of the cost present in the system at the current point in time, that is, the current time delay of actions currently executing and the amount of traffic in the server LAN. The total cost is the integral of the current cost over time, that is, between each timestep from t_p to t_c, the quantity $c_c \times |t_c - t_p|$ is added to the total cost, where c_c is the current cost at time t_c. This gives rise to a stepwise (rectangular) overestimate of the integral, as shown in Figure 5. This means that the total cost is in fact an upper bound on the overall cost of the simulation. A tighter upper bound can be found by using integral approximation methods such as the trapezoidal method or Simpson's

[6] Due to C++ limitations with templated typedefs, these are actually implemented as macros.

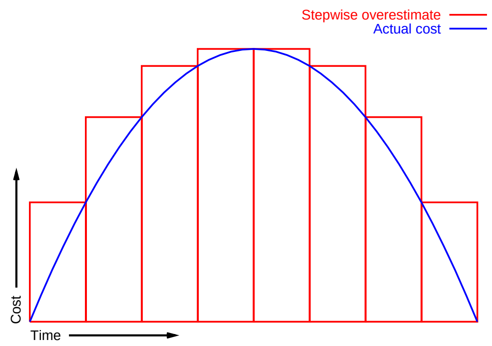


Figure 5: Cost overestimation. The area under the red stepwise cost approximation function is greater than the area under the blue smooth actual cost function.

rule.

The current cost can be separated into unavoidable and avoidable parts. As described in Section 6.2, the unavoidable cost is comprised of those costs which must always be incurred, such as upstream client packets, and the avoidable cost is comprised of those costs which are a result of a particular allocation or communication strategy, and so could conceivably be improved (or even eliminated entirely) by a better strategy. The motivation for this is that network saturation can be the result of either avoidable or unavoidable costs. When caused by unavoidable costs, it is simply a limitation of the available network architecture, and so is of no further concern. However, when it is caused by avoidable costs, the network saturation may be the result of a poor strategy. This means that the network usage recorded is not the true network traffic required, but rather has been saturated at the network bandwidth. This generally causes the traffic to "smear" forward in time, since a saturated network induces lag (time delay). Figure 6 illustrates the situation, where the actual network traffic includes C, but the traffic recorded doesn't include C, and is more like B alone. The distinction between avoidable and unavoidable costs is a natural one to make, and is very useful in detecting when recorded traffic has been distorted as a result of the underlying unknown strategies involved.

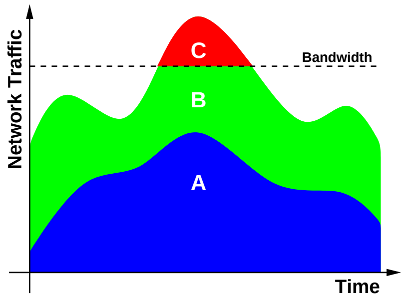


Figure 6: Network saturation from avoidable costs. A indicates the unavoidable cost, B indicates the avoidable cost resting on the unavoidable cost, and C indicates that portion of the avoidable cost which is lost due to saturation.

7.2 Cost Internals

Different kinds of costs depend upon different factors. Different instances of costs will depend on their factors in differing quantities. For example, the cost of sending a packet may depend linearly upon the size of the packet (and is thus different for differently sized packets), whereas the cost of a client joining the network may be constant. Thus costs need to be specified with adequate flexibility to accommodate this situation.

The solution chosen is to specify costs as an array of n coefficients, that is, the TimeCost and DataCost components are each an array of n coefficients. The actual value of an instance of a cost is found by "flattening" the cost, which involves multiplying each of the cost's coefficients by the value of the corresponding variable in the cost instance, and then adding these together. That is, for a cost component (set of coefficients) $c_1, c_2, \ldots, c_n$ and instance variables $V_1, V_2, \ldots, V_n$ with values $v_1, v_2, \ldots, v_n$, the flattened cost is $c_1v_1 + c_2v_2 + \cdots + c_nv_n$. The value of n and the instance variables $V_1, V_2, \ldots, V_n$ are particular to each different type of cost. The particular cost type determines the value of n and the variables against which these coefficients are multiplied are determined.

Costs are represented in plain text by each of their components separated by whitespace. The components are listed in a well-defined order, which is Time-

Cost followed by DataCost. Each component takes the form $\{c_1, c_2, \ldots, c_n\}$, meaning that costs have the representation $\{t_1, t_2, \ldots, t_n\}\,\{d_1, d_2, \ldots, d_n\}$.

7.3 Kinds of Costs

A special kind of cost is the Scalar Cost, which has only one coefficient for both TimeCost and DataCost. These represent the special case of a constant cost which depends upon nothing. As such, they are useful in many places where one just needs to record a single, independent cost, for example, the result of flattening a cost instance is a scalar cost, and the current and total costs are also scalar costs.

There are two other main kinds of costs; those incurred when a client updates its state, discussed in Section 9.4.2 (with the update state packet entity), and those incurred when a client is assigned to a server, discussed in Section 9.5.2 (with the assignment packet entity).

7.4 Template Costs

The simulation maintains a set of *template costs*. The various cost-incurring actions are identified by their unique identifier, and each is mapped to their corresponding template cost. The template costs are the actual cost coefficients used for these actions, and so are multiplied against the values of the cost instance variables, as described in Section 7.2. The template costs may be any kind of cost, although some kinds of costs are defined specifically for one template cost.

7.4.1 Template Cost Instances

The set of template costs present in the simulation are now briefly described.

- The 'updateState' template cost is a cost of the type 'updateState', described in Section 9.4.2. It is evaluated and added to the current cost whenever a client updates its state.

- The ‘assign’ template cost is a cost of the type ‘assign’, described in Section 9.5.2. It is evaluated and added to the current cost whenever a client is assigned to a server.

- The ‘overfullServer’ template cost is a scalar cost which is the penalty incurred whenever a server’s capacity is exceeded. It is described in detail in Section 9.2.2.

8 Events

The simulation is event-based, and as such centers around events which occur at instantaneous points in time. The events are then stored in a priority queue based on increasing event time, and events are taken in turn from the head of the queue and executed. The execution of an event can add more events to the priority queue.

In general, an event will manipulate a single object in the simulation. This object, usually an entity, is referenced by an identifier which is present in all events. If an event doesn’t need to reference anything its identifier can easily be padded with a short dummy value. Entities are discussed in detail in Section 9.

Since supporting events which last for a non-zero duration would add considerable complexity to the simulation, if such events are needed they are instead modeled by separate “begin” and “end” versions of the event, which are used to delimit the extent of the event in time.

8.1 General Format

As described in Section 10, events are stored in simple, plain text logfiles, with one event per line. Events have the following general plain text representation:

event_time id event_type [*event_options* ...]

- The *event_time* field is the time of the event. This is common to all types of events and is of relevance when sorting the events, hence it appears first.

- The *id* field is the identifier the event is dealing with.

- The *event_type* field is a keyword identifying the type of event being recorded.

- Finally, the remainder of the fields in the event specify options and details of the event; their format is specific to the particular type of event (as specified by *event_type*).

The following sections describe the main types of events, how they function and their format.

8.2 Entity Creation

The `create` *event_type* specifies the creation of an entity. It has a general form, but takes several specific forms, depending on exactly which type of entity is being created. These specific variants are detailed in the sections of the individual entities, Sections 9.1 – 9.5.

Its general plain text representation is:

event_time *entity_id* `create` *entity_type* [*creation_options* ...]

- The *entity_id* field specifies the identifier of the entity which is to be created.

- The *entity_type* field is a keyword identifying the type of entity which is to be created. The creation of specific entity types is detailed below.

- The remaining fields form the *creation_options*, these are optional fields which are particular to the specific entity type being created. Typically they are used to specify some kind of initial state for the newly created entity.

8.3 Entity Destruction

The `destroy` *event_type* specifies the destruction of an entity. Its plain text representation is:

event_time *entity_id* `destroy`

The *entity_id* field identifies the entity to be destroyed and removed from the simulation. Exactly what happens to any dependencies of the entity (such as clients of a removed server), depends upon the type of entity being destroyed and its specific destructor.

8.4 Sending and Receiving Packet Entities

The `send` *event_type* specifies that the packet identified by *packet_id* is commencing its journey across the network from its source to its (unknown) destination. Its plain text representation is:

event_time *packet_id* `send`

Packets may only be sent once, after they are sent, they should be received and subsequently destroyed.

The `receive` *event_type* specifies that the packet identified by *packet_id* is ending its journey across the network. Its plain text representation is:

event_time *packet_id* `receive`

A packet can only be received if it was sent at some time in the past.

8.5 Template Cost Specification

The `setcost` *event_type* specifies the value of a template cost. (Template costs are described in Section 7.4.) Its plain text representation is:

event_time *template_cost_id* `setcost` *cost_type* *cost_value*

- The *template_cost_id* field identifies the template cost being set.

- The *cost_type* field identifies the type of cost the template cost is to be set to.
- The *cost_value* field is the actual value of the cost. Costs are represented in plain text as described in Section 7.

For example, to set the 'overfullServer' template cost to a scalar cost with time cost of 10000 and no data cost, the following event may be used:

```
0 overfullServer setcost scalar {10000} {0}
```

9 Entities

An entity is an individual 'object' in the simulation. The simulation has three different types of entities; clients, servers and packets. Each entity in the simulation is identified by a unique identifier (identifiers are described in Section 6.5).

Entities are created, destroyed, and generally manipulated with events, as described in Section 8. However, since entity creation events are specific to each type of entity, these events are discussed with their respective entity type in the following sections.

9.1 Client Entities

A client entity represents a client of the game, connected to the main game network, which allows a player to play the game. Clients are assigned to a particular server called their 'assigned server', and all of the client's communication is with this server alone. A client's assigned server is dynamic, meaning that it may change throughout the simulation in order to best accommodate the client's changing state.

Clients maintain the state of the player (the 'player state'), which is simply their location in the arena. The state could be extended to cover other information, such as the player's velocity, but for the purposes of the simulations under investigation, the position alone is sufficient. Section 13 describes how this state is modeled between the points in time when it is definitively known.

9.1.1 Client creation events

The `client` *entity_type* in a creation event indicates that a client entity is to be created. Its plain text representation is:

event_time *client_id* `create` `client` *initial_player_state*

The only creation option, *initial_player_state*, is the initial state of the client when it joins the game at the time of the event. It is the plain text representation of a player state, which is simply its location, as a pair of floating point coordinates. The coordinates are surrounded by parentheses and separated by a comma in the usual convention. For example, `(795.472,209.301)`. If the player state is extended to store other information, this information can be stored after the location, separated by whitespace. Since the fields defining a player state are always known, they can be unambiguously parsed[7].

9.2 Server Entities

Server entities represent servers in the network. The set of clients assigned to a server are called its 'assigned clients'.

Servers have a capacity in terms of the number of clients they can be assigned. This capacity is not a hard constraint, any number of clients can be assigned to a server. A server with more assigned clients than its capacity allows is said to be *overfull*. A cost is incurred when a server is overfull which will be described in Section 9.2.2.

[7] If optional fields are desired then the plain text representation can be modified to keep player states unambiguously parsable.

9.2.1 Server creation events

The `server` *entity_type* in a creation event indicates that a server entity is to be created. Its plain text representation is:

event_time *server_id* `create server` [*client_capacity*]

There is a single creation option, the client capacity of the server. If it is omitted, it defaults to 32.

9.2.2 Overfull server costs

A server which is overfull has exhausted some of its resources, such as CPU cycles, RAM, or network bandwidth. This means that the server will not perform to its best, and this is accounted for with the 'overfullServer' template cost (template costs are described in Section 7.4). This is a scalar cost which is added to the current cost for each client which is above the capacity limit of a server. These clients are called *overfull clients*, as they are the clients causing the servers to be overfull.

We write the number of clients currently assigned to the i-th server as a_i and its capacity as c_i. For m servers, the total number of overfull clients is given by

$$n = \sum_{i=1}^{m} \max(a_i - c_i, 0).$$

During the simulation the current cost always contains the quantity $n \times t_{of}$, where t_{of} is the 'overfullServer' template cost.

9.3 Packet Entities

Packet entities represent packets in the network. They are used to transfer atomic messages between servers and clients. Clients can only send packets to the server they are currently assigned to. Servers may send packets to any client or server, but if they send a packet to a client which is not assigned to them, the packet must first travel to the client's assigned server.

As described in Section 6.2, packets record only their source entity, so as not to reveal the allocation strategies in use. However, packets are still sent and received, so that they are in the network for a finite amount of time. This makes sense, because we are interested in what information is being sent and when, but it doesn't matter particularly to which server or client it is being sent to. In any case, the exact server or client involved can be deduced from the current allocation.

There are two specific types of packets: UpdateState packets and Assign packets. UpdateState packets are used by clients to notify their assigned server that their player state has changed, and Assign packets are used by servers to indicate to other servers that a client's assignment has changed. Both are discussed in their specific sections, 9.4 and 9.5 respectively.

9.3.1 Packet creation events

Packet entities are created by creation events with *entity_type* of `packet_`***`type`***, where ***`type`*** is a keyword specifying the type of packet to be created. Its plain text representation is:

event_time *packet_id* `create` `packet_`***`type`*** *src_entity_id* *missing_equivalents* [*packet_options* ...]

The *src_entity_id* field is the identifier of the entity (client or server) from which the packet is originating.

The *missing_equivalents* field is an integer indicating how many other packets equivalent to this one have been omitted due to packet downsampling. As described in Section 13, performance concerns mean that these other packets have not been explicitly created, but due to cost concerns their existence must still be acknowledged and accounted for. This field indicates how many of these similarly typed missing packets are being represented by the packet. When no downsampling is in use, this field is always 0.

The remaining fields form the *packet_options*, these are any optional fields particular to the specific type of packet being created.

9.4 Update State Packet Entities

Update state packets are packets which carry notification of a given player's change of state.

Conceptually, they are used both upstream, for a client to notify its server of a state change, and downstream, for a server to notify a client of a state change for some other client. However, for performance reasons (in both the recording of logfiles and the actual simulation), only the upstream packets are explicitly present, downstream packets are handled implicitly by considering their network cost in update state costs, discussed later in Section 9.4.2.

They store a complete copy of the player's new, updated state. It may be feasible to transmit only incremental changes in state, however this wouldn't be robust in the presence of packet loss, which is often a real concern in multiplayer Internet games[8].

9.4.1 Update State packet creation events

The `packet_update_state` *entity_type* in a creation event indicates that an update state packet entity is to be created. Its plain text representation is:

event_time *packet_id* `create` `packet_update_state` *src_entity_id* *missing_equivalents* *new_player_state*

The *src_entity_id* and *missing_equivalents* fields are the same as for general packets.

The only packet option, *new_player_state*, is the updated state of the player which the server is being notified of. It has the same player state plain text representation described in Section 9.1.1[9].

[8] The packet loss is usually due to the use of an unreliable network transport (such as UDP) for speed.

[9] As such, it may actually be more than one field, but as detailed in Section 9.1.1 this is

9.4.2 Update State Costs

This is the cost incurred when a client notifies its server of a change in its state. The variables it is in terms of are:

1. One — that is, the first cost coefficient is a constant.
2. The number of neighbours.
3. The number of neighbours assigned to other servers.
4. The number of distinct servers the neighbours are assigned to.

A typical state update cost might be something like `{100,50,50,0}`. The first value indicates the (constant) size of the update state packet. The second and third values are used to specify the downstream packets sent to the neighbours informing them of this state update. The second value indicates the size of the packet sent to each neighbour, the third value is the interserver journey of the packet for those neighbours which are not on the same server as the updating client. The fourth value of zero indicates that the cost does not depend on the number of distinct servers in the neighbourhood. However, this could be used to send only a single interserver packet to each server in the neighbourhood which then forwards it on to the appropriate clients, rather than possibly many nearly identical packets to each server, which is an inefficient use of the LAN.

9.5 Assign Packet Entities

Assign packets are packets which carry notification of a client's assignment to a particular server. They are sent from the server which has decided to do the assignment, which is usually the server to which the client is being assigned.

Exactly where the packets are sent to depends on the exact method used to inform servers of client assignment changes. Due to the design of packets

not a problem as it can always be unambiguously parsed.

the exact destinations aren't known, but their number is; it is controlled by the assignment cost, as described in Section 9.5.2.

9.5.1 Assign packet creation events

The `packet_assign` *entity_type* in a creation event indicates that an assign packet entity is to be created. Its plain text representation is:

event_time *packet_id* `create` `packet_assign` *src_entity_id* *missing_equivalents* *assigned_client* *assigned_server*

The *src_entity_id* and *missing_equivalents* fields are the same as for general packets.

There are two packet options, *assigned_client* and *assigned_server*. The *assigned_client* field is the identifier of the client being assigned, the *assigned_server* field is the identifier of the server that *assigned_client* is being assigned to. This assignment overrides and replaces any previous assignments, so prior to the assignment the client is deassigned from its previous server, *deassigned_server*.

The packet is sent from the server which has made the decision to allocate. Generally this is *assigned_server*, but it could be any server. For example, a server may exist specifically to make allocation decisions, in which case all assign packets would be sent from this server.

9.5.2 Assignment Costs

This is the cost incurred when a client is assigned to a server. The variables it is in terms of are:

1. One — that is, the first cost coefficient is a constant.
2. The number of clients assigned to *assigned_server*, excluding *assigned_client*.
3. The number of clients assigned to *deassigned_server*, excluding *assigned_client*.
4. The total number of clients.

5. The total number of servers.

A typical assignment cost might be something like {100,200,300,0,0}. The first value indicates the (constant) overhead involved in assigning a client (for example, notifying the client's FEP of the new server to direct communications to). The second and third values reflect the notification of the affected clients of the assignment change. The fourth value of zero indicates that assignments don't cause all clients to be notified. Similarly, the fifth value of zero indicates that assignments don't cause all servers to be notified.

Typically not every client or server will need to be notified of assignments. However, situations which require this flexibility are conceivable. For example, if every server knows the assignment of every client, then when an assignment occurs all the servers must be notified.

10 Logfiles

Logfiles are simply collections of events stored in plain text files. They are used in both input and output modes; input for reading a set of events to be used in the simulation, output for recording (logging) the events which are executed by the simulation.

Logfiles have one event per line, with fields of the events separated by whitespace. The exact format of events is detailed in Section 8.1. Blank lines (including those containing only whitespace), ill-formatted lines and lines beginning with the hash symbol ('#') are ignored.

This file format means that logfiles are simple and easily parsed by both machine and human; it allows them to be scrutinized by hand when necessary, and also allows the application of standard Unix text-manipulation utilities (such as `grep`, `sort`, `sed`, `awk`, etc) to the logfiles.

There are three different types of logfiles, characterised by the types of events they may store and their intended purpose in the simulation.

10.1 Player state logfiles

Player state logfiles record the state of the game players. A player state logfile is a "complete" record in that it could be "played back" to reconstruct an exact replay of the game from which it was recorded. It records what the players have done (particularly their movements), without any concern for issues peripheral to this, such as the network or strategies which happen to be in use. This means that it is essentially a record of the upstream data from the clients to the servers, because the clients notify the servers of what they are doing in the game.

The specific events allowed are:

- Creation of client entities (`create client`)
- Creation of update state packet entities (`create packet_update_state`)
- Destruction of entities (`destroy`)
- Sending of packets (`send`)
- Receiving of packets (`receive`)

10.2 Architecture logfiles

Architecture logfiles record the architecture used to play a game. The architecture is the server-side "hardware" setup on which player state logfile events may be played back on. It includes the servers themselves and the server LAN network, and their associated costs, limits and other properties.

It is usual to have an initial "setup" phase of the simulation in which only architecture events are executed. However, in keeping with the discrete event simulation design, the architecture may be changed at any point in the simulation. This can be useful for testing situations such as how the system performs when a server is destroyed (for example, it crashes) at a certain point in time.

The specific events allowed are:

- Creation of server entities (`create server`)
- Destruction of entities (`destroy`)
- Setting of template costs (`setcost`)

10.3 Allocation logfiles

Allocation logfiles record the assignments of clients to servers. It is the output of an allocation strategy, and can thus be used instead of an allocation strategy. In this way, allocation logfiles hide the actual allocation strategy used, recording only the decisions it makes. This allows allocation strategies to be compared without needing to disclose their methods.

The specific events allowed are:

- Creation of assign packet entities (`create packet_assign`)
- Destruction of entities (`destroy`)
- Sending of packets (`send`)
- Receiving of packets (`receive`)

For example, the allocation logfile events to assign client `CLI_0034` to server `SRV_010` would look like

```
 1000 PKT_00632 create packet_assign SRV_010 CLI_0034 SRV_010
 1010 PKT_00632 send
11010 PKT_00632 receive
11020 PKT_00632 destroy
```

11 Allocation Strategies

A substantial goal of the project is to allow comparisons between various allocation strategies. This suggests the *Strategy Design Pattern* (or *Strategy DP*) [1] for the design of allocation strategies. The `AllocationStrategy` class defines the interface of an allocation strategy; that is, what it means to be an allocation strategy. The `AllocationStrategy` class is subclassed for the actual implementations of the various allocation strategies. The Strategy DP means that any particular subclass of `AllocationStrategy` may be used when an allocation strategy is needed.

11.1 Interface

Allocation strategies are notified of the execution of events. The allocation strategy is given the event, and it then determines which type of event it is and what action should be taken, if any. The external interface is a single `notify` method with an `Event` object as its sole parameter. An internal interface is provided by way of a `notify_`*`type`* method for each type of event, with parameters particular to the type of event. This means that the work of determining the event type and extracting the useful information from the event is implemented only once by the base `AllocationStrategy` class. Particular allocation strategy sub-classes need only override the notification methods for the event types they are interested in.

Allocation strategies decide to make assignments based on the information they receive in notifications. When a strategy is notified of a change, it returns the (possibly empty) set of chosen assignments. These assignments are stored as allocation logfile events, described in Section 10.3. The events are usually scheduled for immediate execution. They are added to the currently running simulation, and may also be stored in an allocation logfile.

The interface allows allocation strategies to augment the arena display (described in Section 6.1) with useful graphical information about how the allocation is being determined.

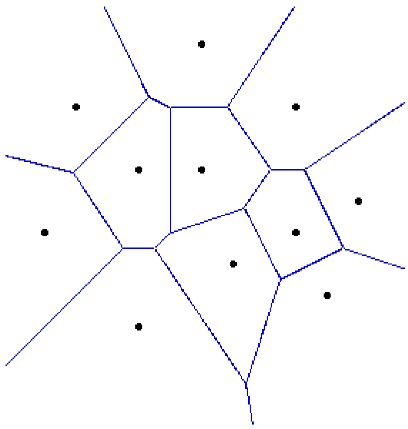


Figure 7: An example of a Voronoi diagram. Sites are shown as black dots and Voronoi edges are shown in blue.

11.2 Default Allocation Strategy

The default allocation strategy used by the simulation is a simple and straightforward allocation strategy called *cellular*. It is relatively efficient and gives reasonably good results, making it a good benchmark for more advanced allocation strategies. Part III discusses the results of experimentally testing the cellular allocation strategy.

11.2.1 Background

The *Voronoi diagram* [2] is a geometric structure which captures information about the proximity of points in the plane to various identified points known as *sites*. Each site lies within its associated *Voronoi region* — the set of all points which are closer to that site than to any other. The boundaries of the Voronoi regions, called *Voronoi edges*, are those points equidistant from two sites. The points where Voronoi edges meet, called *Voronoi vertices*, are those points equidistant from three or more sites. Figure 7 shows an example of a Voronoi diagram.

The Voronoi diagram is useful because it can be used to efficiently answer proximity queries. The two proximity queries most useful in this application are to find the nearest site to a query point q (that is, which Voronoi region q lies

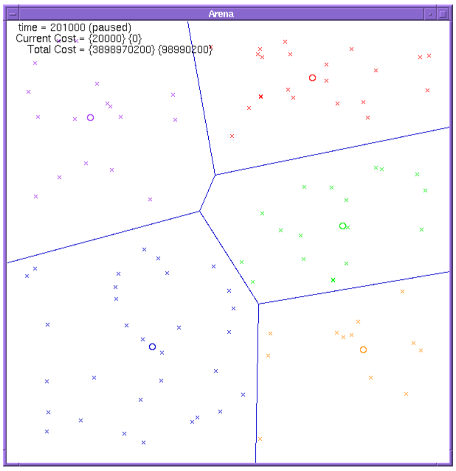


Figure 8: An example of how the Cellular allocation strategy divides the arena into cells, with each cell handled by a server. The Voronoi edges are in blue and the small open circles are the server representatives.

in), and to find the *nearest neighbour* of a site s, which is the site t closest to s.

For n sites, the Voronoi diagram is of size $O(n)$ and takes $O(n \log n)$ time to construct. After this initial preprocessing step it takes only $O(\log n)$ time to find the nearest site to a query point and the nearest neighbour of a site [2].

11.2.2 Algorithm

The cellular allocation algorithm is based on the k-Means clustering heuristic [5]. For each server, a *representative* point is maintained which is the average of the locations of the server's assigned clients. The Voronoi diagram of the server representatives is used to ensure that all clients are assigned to the server with the closest representative.

As the clients move around the arena, the representative points of the servers also move, since they are determined from the locations of the clients. This causes the Voronoi diagram to change, and if this causes some clients to change Voronoi regions, then these clients must have their assignment changed as their nearest representative point has changed.

Initially, clients are randomly assigned to empty servers until every server has a single client.

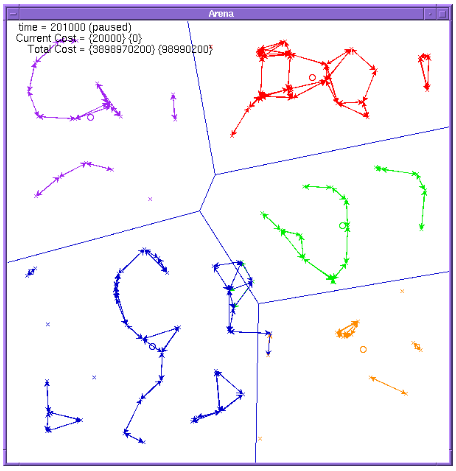


Figure 9: The same as Figure 8 but also showing the neighbourhood of each client. The neighbourhood edges which cross Voronoi edges correspond to interserver traffic.

As shown in Figure 8, the Voronoi diagram effectively divides the arena into *cells*, where each cell is handled by a single server. Figure 9 is the same as Figure 8 but with edges showing the neighbourhood of each client. Neighbourhood edges which cross Voronoi edges will cause interserver traffic whenever the clients involved update. This display can be particularly insightful, as it allows problem areas of the current allocation to be examined directly in the arena.

11.3 Future Allocation Strategies

As mentioned in Section 11.2, the Cellular allocation strategy is just one of many possible allocation strategies which may be implemented. This section describes one of the possible allocation strategies which is expected to be promising.

11.3.1 Combining Allocation Strategies

The Strategy Design Pattern allows allocation strategies to be combined in novel and interesting ways.

For example, an allocation strategy called `Dual` could be written which uses n other "sub"-allocation strategies, called `Sub1`, `Sub2`, ... `Sub`n, all of which are arbitrary allocation strategies. Whenever `Dual` is notified of an event it in turn notifies each of the `Sub`i, for $1 \leq i \leq n$, of the event. None of the `Sub`i are aware of this intervening "layer". `Dual` then receives the assignments from each `Sub`i resulting from the event, and chooses to return the "best" of these assignments.

Comparison of assignments would defined in some way, and perhaps even a parameter of `Dual`. The most obvious assignment comparators would be related to the foreseeable cost and impact of the assignments, but some may be entirely unrelated to the cost, such as how recently the entities involved were previously assigned.

Another example is an allocation strategy which uses a `SubA` allocation strategy when certain conditions in the simulation are true, and a `SubB` allocation strategy when these conditions are false. This allocation strategy would be quite simple to write, but would be excellent for investigating and developing 'hybrid' allocation strategies which are combinations of other, more specialised allocation strategies. A good example is a strategy which uses different sub-allocation strategies depending on whether the simulation contains many or few clients.

12 Communication Strategies

The set of clients which are informed of a given client's actions is called the *neighbourhood* of that client, and the informed clients are called the *neighbours* or *neighbourhood clients*.

As described in Section 6.2, clients send upstream packets to their assigned server and receive downstream packets from it. The upstream data consists of notifications of the player's actions and the downstream data consists of the actions of other players the client is being informed of.

The amount of upstream data is essentially fixed, as the client is always informing the server of its actions, but the amount of downstream data depends upon the number neighbourhoods this client is a part of. In addition, since the neighbourhood relationship is one to many, clients must receive more downstream data than they send upstream.

12.1 Communication Strategies Implemented

The project implements two communication strategies. For a given client c,

- the 'circle' communication strategy chooses neighbourhoods to be the clients which lie within a circle of radius r centered on the location of c (where r is a parameter of the strategy), and
- the 'unearest' communication strategy chooses neighbourhoods to be the u nearest neighbours [2] of c (for some particular value of u).

The 'circle' communication strategy appears to be the more promising of the two strategies. This is because the 'unearest' strategy suffers from the problem that in sparse areas, some of the neighbours may be considerable distances away. This means that they are more likely to be on other servers, and may therefore cause unnecessary interserver traffic.

A possibility which was not explored, but is expected to be promising, is a hybrid combination of both 'circle' and 'unearest'. This would choose neighbourhoods to be those clients which are both a u nearest neighbour of c and also lie within a given circle centered on the location of c. This would avoid choosing nearest neighbours which are considerable distances away, avoiding the problem 'unearest' has in sparse areas.

12.2 Interface

The `CommStrategy` class uses the *Strategy Design Pattern* [1] to abstractly define the interface used by communication strategies for choosing the neighbourhood of a client.

In a fashion similar to allocation strategies, communication strategies are notified of the execution of events. However, unlike allocation strategies, notification of communication strategies has no return value. Instead, the communication strategy can be queried at any point in time, which returns the neighbourhood of the queried client entity. Neighbourhoods are represented simply as a collection of clients.

12.3 Communication Graph

The neighbourhoods of all the clients form a graph structure, where the nodes are the clients and two clients u and v are joined by a directed edge uv if v is in the neighbourhood of u. This graph is called the *communication graph*.

It would be possible, and indeed probably preferable, to have communication strategies maintain such a graph, since that is what they are in essence doing with the queries they provide. They would then provide read-only access to this graph, to allow the neighbourhood of a client to be found.

A communication graph is not explicitly maintained because it would add substantial complexity to the project, while the current system of querying the neighbourhoods of individual clients is adequate. If it is desired, the communication graph can be constructed by simply querying the neighbourhood of each client.

12.4 Communication Graph Layout

The communication graph is displayed in its own window. It is laid out in a fashion which highlights the two types of edges: those joining clients assigned to the same server (intraserver edges), and those joining clients assigned to different servers (interserver edges). This allows the user to see problem areas relatively easily, and to allow the cause of these to be determined by relating back to the arena display.

The layout is quite simple, and is illustrated in Figure 10. The clients assigned to a given server are arranged in a circle (of fixed radius) and coloured the same as their server. Each of these circles is then arranged with its center on

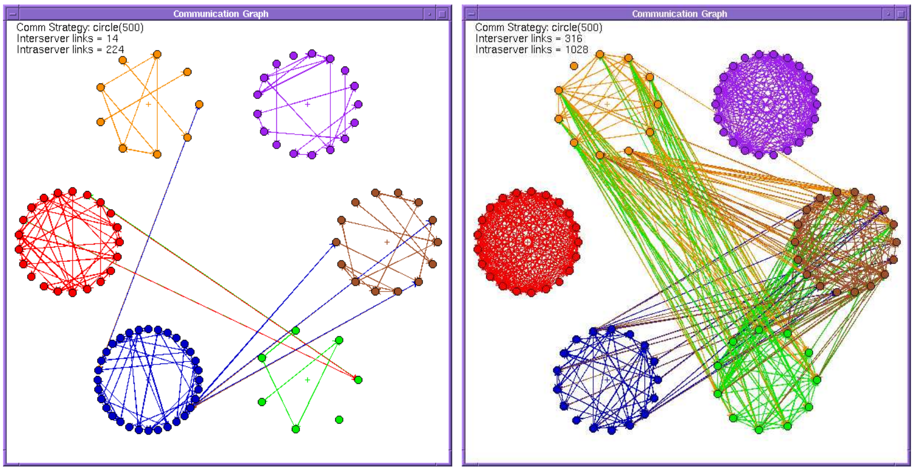


Figure 10: Examples of communication graphs.

the circumference of a larger circle. The clients are then joined to their neighbourhood clients with directed edges. This arrangement means that intraserver edges are always contained within their circle, and so intraserver edges of different servers won't cross, and also that interserver edges will always pass through the central region. The main benefit of this layout are that it conveys a lot of information at a glance. It is easy to see how populated a given server is (how many clients it has around its circumference), how dense it is (how many intraserver edges it has), and of course how many interserver edges there are (as these are to be avoided).

This layout scales poorly as the number of clients and servers increases. It is expected that better layout could be designed given sufficient development effort, however this is outside the scope of this project.

13 Client Movement Modeling

As noted earlier in Section 10, logfiles can be very large if they have a high resolution. This introduces some substantial performance issues. For example, a typical situation could be 1000 clients updating their state 10 times per second. To create, send, receive and then destroy an update state packet typically requires about 140 characters in the logfile. This results in a logfile which grows at a rate of about 1.3Mb/s. Both reading and writing this rate of data is a formidable task. In addition, the priority queue operations should be taken into consideration (and are typically $O(\log n)$); these also slow the system.

This problem is handled by the *modeling* of the client movement. This involves downsampling the state update packet stream, that is, recording only every n-th packet (referred to as the key packets[10]) for each client and discarding the rest. At the time of key packets, the state of the client is definitively specified by the contents of the packets; between key packets a mathematical model is used to approximate the state of clients. The model not only allows the state to be interpolated and extrapolated from the key packets, but also allows intersection style proximity queries to be made, for example, to determine if two clients are within a certain distance of each other during a given time interval.

A large disadvantage of these modeling techniques is that they can make the computational complexity of updating the state of a set of player much worse. When modeling is not used, techniques such as Voronoi diagrams (Section 11.2.1) allow the set of clients "close" to some other client can be found in $O(\log n)$ time. However when modeling is used, the approximated positions of all n clients must be compared to the client's approximated position, which takes $O(n)$ time. An expected future development to this would be a hybrid system, which uses the efficient techniques to discover the set of clients whose approximated positions should be compared explicitly. This would give the benefits of modeling without the poorer computational complexity.

13.1 Implementation

The `PlayerState` class, briefly introduced in Section 9.1, is the core of the modeling system. In addition to defining the interface for obtaining and setting the state, it also abstractly defines the interface for estimating the state at any

[10]By analogy with key frames in computer animation.

point in time, estimating the derivative of the state at any point in time (that is, the client's velocity), estimating the intersection of the state with lines and other geometric constructs, and estimating the minimum distance between two clients in a given time period. The `PlayerState` class itself is not strictly abstract, as it implements the linear model, but it may easily be subclassed, allowing any model of the client movement to be substituted.

13.2 Details of the model used

In general, the performance of the model will be related to its sophistication. A default linear model is implemented, however it can easily be substituted for more sophisticated models if required.

The linear model makes the assumption that clients only travel in a straight line between any two points a and b. More sophisticated models, such as quadratic, cubic, or higher polynomial functions, would increase accuracy, but would require disproportionally more programming time, and so the linear model is satisfactory.

We use vector notation, where boldface indicates a vector, and positions are represented as vectors from some origin.

The linear model stores the "current" position $\mathbf{p}_c$ and the corresponding time t_c, and the "previous" position $\mathbf{p}_p$ and time t_p of the client. This allows the velocity to be estimated at any point in time by $\mathbf{v} = \frac{\mathbf{p}_p - \mathbf{p}_p}{t_c - t_p}$. The estimation of the position at any time t is given by $\mathbf{p}_p = \mathbf{p}_p + \mathbf{v} \times (t - t_c)$. Proceeding from these elementary estimations, it is then possible to estimate the intersection of the client with various geometric structures, such as lines, rays and circles.

The estimated minimum distance between two clients is more interesting. Here we have the situation of two clients a and b moving at arbitrary velocities $\mathbf{v}_a$ and $\mathbf{v}_b$ between two points in time t_i and t_f, and we wish to find their minimum separation during that time. We examine the vector $\mathbf{m}$ joining the position of the two clients at time t, clearly this will be of minimum length when it is perpendicular to $\mathbf{v}_a$ and $\mathbf{v}_b$. Here $\mathbf{a}_i$ and $\mathbf{b}_i$ are the positions at time t_i of a and b respectively, and $\mathbf{a}_f$ and $\mathbf{b}_f$ are the positions at time t_f of a and b respectively, the vector $\mathbf{m}$ and the time t at which it occurs is given by

$$\begin{aligned} t &= \frac{(\mathbf{b}_i - \mathbf{a}_i) \cdot \mathbf{v}_a}{(\mathbf{v}_a - \mathbf{v}_b) \cdot \mathbf{v}_a} \\ \mathbf{m} &= (\mathbf{a}_i - \mathbf{b}_i) + t(\mathbf{v}_a - \mathbf{v}_b) \end{aligned}$$

There are three distinct cases:

- $t \leq t_i$, that is, $\mathbf{m}$ is perpendicular before the time period. In this case, the distance between the clients at the start of the time period, $|\mathbf{a}_i - \mathbf{b}_i|$, is the result.
- $t_i < t < t_f$, that is, $\mathbf{m}$ is perpendicular within the time period. In this case, $|\mathbf{m}|$, the length of the vector $\mathbf{m}$, is the result. It is acceptable for $\mathbf{m}$ to be perpendicular at more than one time between t_i and t_f, as all such $\mathbf{m}$ vectors will be of equal length.
- $t_f \leq t$, that is, $\mathbf{m}$ is perpendicular after the time period. In this case, the distance between the clients at the end of the time period, $|\mathbf{a}_f - \mathbf{b}_f|$, is the result.

14 Bots

As noted in Section 10, the input of the simulation is abstracted by using logfiles, which may come from any source. The two main anticipated sources are recordings of actual gameplay and generated test scenarios involving *bots*. Bots are simple, automated clients which are simulated for the sole purpose of generating logfiles. They are used because recording actual gameplay can be both difficult and time-consuming.

14.1 Bot Behaviour

Naturally, a bot corresponds to a player (and client) in the simulation. The bot behaviour is the way in which bots move around the arena. The simulation uses a simple and straightforward scheme to acheive bot movements which are considered similar to those expected in some real-world scenarios.

A number of *stations* are defined at various locations in the arena, and these stations are joined by directed edges. Initially, the bots are distributed uniformly throughout the arena, and each bot travels in a straight line to its nearest station. When a bot reaches a station, it remains ("waits") at that station for a random period of time between the station's minimum and maximum wait times. While at a station, the bot moves randomly ("wanders"), not traveling further than the station's wander distance from the station's location. When a bot has finished waiting at a station, it randomly chooses one of the adjacent stations and travels to that station. In addition, at any point the set of stations and their adjacencies and properties can be changed, which makes the bots once again move to their nearest station and continue from there, as they did initially.

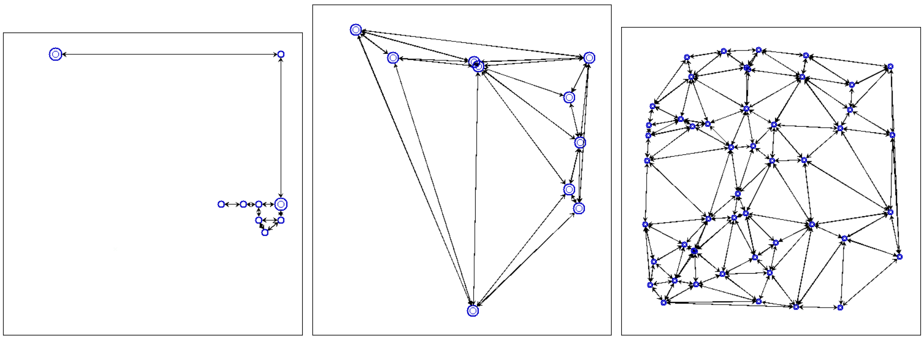
An easy and standard way of obtaining a set of stations and their adjacencies is to take a random set of points and then use their Delaunay triangulation as the adjacency graph. The Delaunay edges can either all be made bidirectional, or randomly assigned one of the forward direction, the backward direction, or bidirectional.

Station files are plain text files storing a set of stations, their properties and the edges between them.

14.2 Implementation

Bots are implemented by simulating their behaviour separately from the main simulation. This bot simulation inputs a station file and outputs a player state logfile containing the events for the movement of the bots.

In the future, performance increases could be achieved by embedding the bots into the main simulation, instead of generating an entire logfile which must then be loaded entirely into the simulation. This would mean that the bot events could be added to the simulation event priority queue progressively throughout the simulation, rather than loading all of the bot movement events at the beginning of the simulation. This would result in less events in the priority queue on average, which would allow the queue to perform better. However this was viewed as a secondary goal, as it provides no more functionality than is already provided by the logfile method.



(a) city (b) wells10 (c) wells50

Figure 11: Station files used in bot generation.

Part III

Results

15 Experimental Design

15.1 Player State Logfiles

The player state logfiles used for the experiments were generated using the bots described in Section 14. This is because no recordings of actual gameplay were available from any known sources and making such recordings was well beyond the scope of the project.

The `city`, `wells10` and `wells50` station files were used; Figure 11 shows their spatial layouts. For each station file a player state logfile was generated for 100, 200, 500, 1000 and 2000 bots, giving a total of fifteen player state logfiles. Each has a duration of 2.5 minutes and was downsampled such that clients update every 10 seconds.

15.2 Architecture Logfiles

The *server load* is defined as the average number of clients per server. Each server was given a capacity of 32, and the server load was given the values of

Template Cost	Cost Type	Time Component	Data Component
updateState	updateState	{100,50,50,0}	{100,50,50,0}
assign	assign	{100,200,300,0,0}	{100,200,300,0,0}
overfullServer	scalar	10000	0

Table 2: Template cost instances used in the experimentation architecture logfiles.

16, 24, 32 and 36. This tests the cases of when the servers are expected to be half-full, three-quarters full, completely full, and an eighth overfull. Thus, the number of servers in a given architecture logfile is simply the number of clients desired divided by the server load. Twenty architecture logfiles were generated, one for each combination of 100, 200, 500, 1000 and 2000 clients with server loads of 16, 24, 32 and 36[11]. The template cost instances (Section 7.4.1) used are shown in Table 2. (For details on the cost instance variables used for the 'updateState' and 'assign' cost types, refer to Sections 9.4.2 and 9.5.2, respectively.)

15.3 Simulation Execution

For each pair of player state and architecture logfiles, the simulation is run with various combinations of allocation and communication strategies.

Two allocation strategies are used,

- the cellular strategy described in Section 8, and
- a random allocation strategy, mainly used for debugging purposes, which statically assigns clients to a random server.

The communication strategies used are the two described in Section 12.1 — the 'circle' strategy, for various values of the radius r, and the 'unearest' strategy, for various values of u. For the `city` based scenarios r is given the values 500, 1000 and 1500, whereas for the `wells10` and `wells50` based scenarios r is given

[11] Since none of the client numbers are divisible by the server loads, the number of servers actually used is the ceiling of the result of the division. This means that the server load is actually an upper bound on the average number of clients per server.

the values 5000, 10000 and 15000. This is because `wells10` and `wells50` have larger dimensions than `city` and so the larger circle radii compensate for the clients being more sparsely distributed (on average). The values of u used are 5 and 10, for all three station files.

To summarise, the experiment has 5 independent variables — the station file, number of clients, server load, allocation strategy and communication strategy. The number of scenarios is given by

$$\begin{aligned} &3 \text{ stations} \times 5 \text{ numbers of clients} \times 4 \text{ server loads} \times \\ &2 \text{ allocation strategies} \times 5 \text{ communication strategies} \;\; = 600 \text{ scenarios.} \end{aligned}$$

The simulation is run on each of these scenarios, and the resulting allocation logfile is recorded. In addition, the following 5 dependent variables are recorded at each point in time during the simulation — the current cost, total cost, number of interserver links, number of intraserver links, and the number of clients causing their server to be overfull. However, since the current and total costs each consist of a scalar time and data component, the number of dependent variables is actually 7.

16 Results and Discussion

The results recorded are 12 dimensional, as they have 5 independent variables and 7 dependent variables. This high dimensionality means that it would be extremely difficult and impractical to directly analyse the entire set of results. Instead, smaller, lower dimensional sections of the results are analysed. Small shell scripts and Unix utilities are used to manipulate the results, obtaining the desired information in a format suitable for plotting with *Gnuplot*. A sample of the most interesting results is presented here, illustrating some of the insights possible with the project.

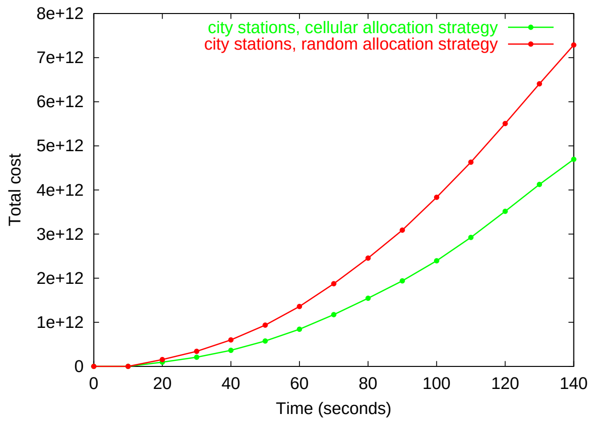


Figure 12: The total cost versus time for the cellular and random allocation strategies, using the `city` stations.

16.1 Total costs

Since the cost system has been designed to take into account several pertinent factors, one of the best results to consider is how the total cost progresses throughout the simulation, and compare this for different allocation strategies.

Consider the scenarios using the `city` stations, 500 clients, 'circle' communication strategy with radius of 500, server load of three-quarters full (24) and both the cellular and random allocation strategies. Figure 12 shows the progression of the total cost throughout the simulation. It can be clearly seen that the costs of both allocation strategies are accelerating during the simulation, but the cellular strategy is doing so at a slower rate than the random one. This is as expected, since the random allocation strategy employs no method of avoiding interserver communications.

Figure 13 shows the same scenario except using `wells50` stations rather than `city`. Again, the cellular strategy is better than the random strategy, suggesting that this may be the case consistently. In addition, after approximately one minute of simulation, the acceleration of the costs of both strategies begins to slow and become linear. This may be attributed to the fact that clients in `wells50` tend to be more evenly distributed than in `city`, since `wells50` has a more even distribution of stations.

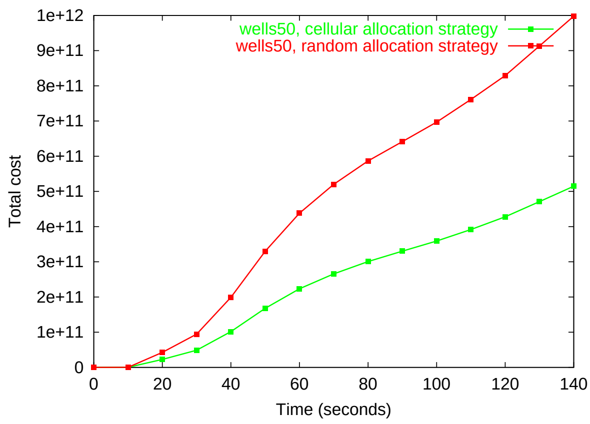


Figure 13: As in Figure 12, the total cost versus time for the cellular and random allocation strategies, except using the `wells50` stations rather than `city`.

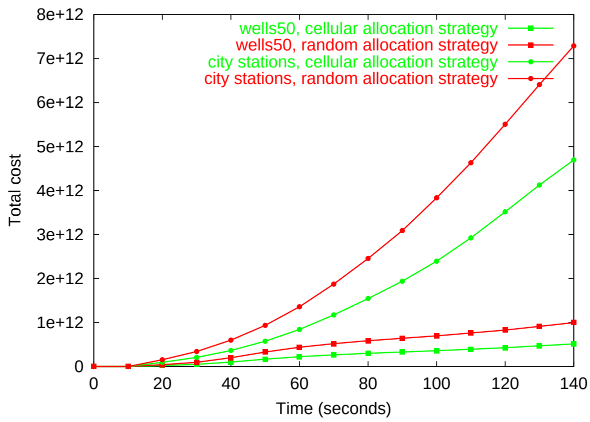
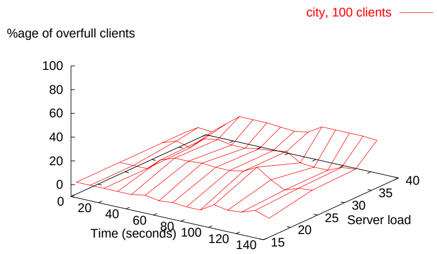


Figure 14: The plots from Figures 12 and 13 on the same set of axes.



(a) front view

(b) back view

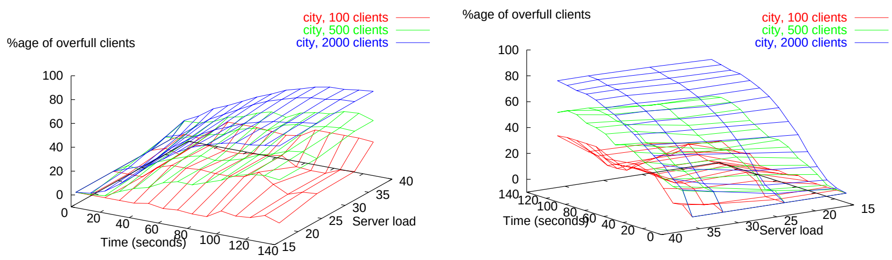
Figure 15: The percentage of overfull clients versus time, versus server load, for 100 clients using the `city` stations.

Finally, Figure 14 shows the data from Figures 12 and 13 on the same set of axes. This clearly shows that the cost for both allocation strategies is much less in the `wells50` scenario compared to the `city` scenario. As before, this can be attributed to a more even distribution of clients in `wells50` compared to `city`. Also noteworthy is that although Figures 12 and 13 appear to indicate that the cellular algorithm is better than the random strategy by a greater margin, Figure 14 shows that this is an artifact of the differing y scales in Figures 12 and 13.

16.2 Overfull clients

One consideration which is not present in the cellular allocation strategy is the capacity of servers. The result is that the cellular strategy may allow a small number of servers to dominate, allocating many clients to those servers and making them badly overfull. 'Overfull clients' are those clients which cause servers to be overfull, as described in Section 9.2.2.

For the case of the cellular allocation strategy, `city` stations, 100 clients and radius 500 circle communication strategy, Figure 15 shows how the fraction of clients which are overfull, changes as the simulation progresses and for the various server loads. This shows that this fraction of overfull clients slowly



(a) front view (b) back view

Figure 16: The percentage of overfull clients versus time, versus server load, for 100, 500 and 2000 clients using the `city` stations.

(a) front view (b) back view

Figure 17: The percentage of overfull clients versus time, versus server load, for 100, 500 and 2000 clients using the `wells50` stations.

increases as the simulation progresses and as the servers become increasingly loaded.

Figure 16 adds the scenarios for 500 and 2000 clients to the plot shown in Figure 15, and shows results which are surprisingly pronounced. It can be seen quite clearly that as the number of clients increases, the fraction of these clients which are overfull increases quite rapidly as the simulation progresses. In fact, at the end of the simulation approximately 70% of the 2000 clients are overfull. This strongly suggests that the cellular allocation strategy does indeed perform poorly where server capacities are concerned, as the number of clients increases.

Figure 17 is similar to Figure 16 except that it uses the `wells50` stations. Here we see slightly different behaviour. As the number of clients increases so does the fraction of overfull clients, but not to the degree found in the `city` stations. This time, approximately 40% of the 2000 clients are overfull at the end of the simulation. This indicates that perhaps the cellular allocation strategy, whilst performing poorly with respect to server capacities, appears to perform somewhat better when the clients are more evenly distributed.

Also, a similar observation to one found in Section 16.1 can be made. As described in Section 16.1, the total cost associated with `wells50` stopped accelerating and continued linearly, the fraction of overfull clients for each of the 100, 500 and 2000 client scenarios stops increasing and remain constant after a short initial period. This can also be attributed to the even distribution of clients, since when the clients are evenly distributed it is unlikely that the number and size of large servers can continue to grow. It is expected that further investigation could reveal if these observations are related to the similar cost-related observations from Section 16.1.

Part IV

Appendices

A Implementation

This appendix describes the reference implementation of the simulation and the software engineering issues which arose during its development.

A.1 Architecture

The project is written in C++, using the LEDA library and GNU Unix tools.

The combination of C++ and LEDA was chosen because it fits the three main requirements of an implementation of the design described in Part II. These requirements are:

1. To have good practical speed, suggesting a compiled language rather than an interpreted one.
2. To be object oriented.
3. To have a high quality algorithmic library freely available. The first and second requirements also apply to this library.

A.1.1 LEDA

The *Library for Efficient Data types and Algorithms*, commonly known as *LEDA* [3], is the main algorithmic library in the simulation. It provides efficient implementations of many common algorithms and their related data structures, particularly those related to graphs and geometry. It is written in C++ and is a mature library with a large userbase. In addition, it allows the use of various specialised extension libraries, such as the CGAL computational geometry library.

The main LEDA data types used by the simulation are

- Linear lists, resizable arrays, dictionaries and priority queues,
- Arbitrary precision integer and rational numbers,
- Geometric objects such as points, vectors, lines, line segments, rays, circles, and so on,
- Graphical windowed displays, and
- *Pointsets*. Pointsets are collections of points in a plane on which a dynamic Delaunay triangulation is maintained. This efficiently provides operations such as computing the Voronoi diagram and various nearest neighbour and range searches.

A.1.2 Naming Convention

All of the files and classes in the project follow a standard naming scheme. The scheme makes it easier to identify classes, determine the class hierarchy, and in the automatic code generation system.

Each word in a class name begins with a capital letter, the rest is lowercase, and no underscores are used. For example, `PlayerState`.

Subclasses only ever append to their parent's name. For example, `EntityPacketAssign` inherits from `EntityPacket`, which in turn inherits from `Entity`. This means that all the subclasses of Entity match the (Perl-style) regular expression `^Entity.*$`, which becomes most useful in the build process (Section A.2) and when generating code (Section A.3).

The class `ClassName` is stored in its own pair of of files, `ClassName.cc` and `ClassName.hh`. The `hh` file is called the header file and stores the class definition, and the `cc` file is called the implementation file and stores the implementation of the methods defined in the header file[12]. Adopting this convention is particularly

[12] The exceptions to this are templated classes, templated functions and nested classes. Current C++ compilers generally use what is termed *template instantiation*, which means that the implementations of templated types must be included in the header file, such that they are compiled for every particular use of the templates.

useful when generating code with M4 (Section A.3.2).

A.1.3 Abstractly Named Types

As is good software engineering practice, the data types used in the simulation are abstractly named in a single common header file, `common.hh`. This means that variables are typed according to their intended usage, rather than their actual type.

For example, identifier variables are of the type `Id_t` and keyword variables are of the type `Keyword_t`, even though both identifiers and keywords are implemented as `String` objects. This allows identifiers to change type independently of keywords, and with only minimal impact on the rest of the code. For example, identifiers may be changed to be integers by changing `Id_t` to be defined as an integer type rather than as a `String`. The only code which must then be modified is that which assumes `Id_t` is actually a `String`.

A.1.4 GNU tools

The GNU Unix tools were chosen because they aid development in many ways, provide features (both language and other) not available elsewhere, they are essentially ubiquitous, and they are easily compiled and installed on a wide range of systems.

The project makes use of the following GNU tools and features.

g++ The GNU C++ Compiler, version egcs-2.91.66 or higher. The GNU compiler is needed for its support of 64 bit '`long long`' integers.

make The GNU make system, version 3.77 or higher. GNU make (as opposed to other versions of Unix make) is needed for the system Makefile because GNU make has many convenient features available which aid in automatic code generation (Section A.3).

m4 The GNU M4 macro processor, version 1.4. M4 is used for automatic code generation, as described in Section A.3.2. GNU M4 is used because it has many advantageous features over traditional versions of Unix M4.

getopt The GNU getopt from GNU libc 2.0 or higher. GNU's getopt system was used for reading command line options; it allows the more descriptive GNU 'long options' to be used. However, it would still be fairly easy to convert back to standard getopt.

A.1.5 Include Guards

Include guards are a simple and commonly used C and C++ language mechanism. They wrap each header file with the code

```
#if !defined(INCLUDED_CLASSNAME_HH)
#define INCLUDED_CLASSNAME_HH
...
#endif
```

This stops the contents of any header file being included more than once, and thus avoids any redefinition errors.

However, the situation in the simulation is not this simple. Adjacency is the situation where two classes refer to one another using pointers or references. In this case, a C++ *forward declaration* must be used. This indicates to the compiler that a class of the given name will be defined at some point in the future. Until this point in time, only pointers or references of the forward defined class may be used. The include guards in the project take into account forward definitions as well as standard include guard features. They have the general form

```
// CLASSNAME.hh
```

```
class CLASSNAME;  // Forward-define CLASSNAME

#if defined(DEFER_CLASSNAME)
#undef DEFER_CLASSNAME
#else
#if !defined(INCLUDED_CLASSNAME_HH)
#define INCLUDED_CLASSNAME_HH

#include "OTHERCLASS.hh"
#define DEFER_ADJACENTCLASS
#include "ADJACENTCLASS.hh"
...

class CLASSNAME {
    ...
};

#include "ADJACENTCLASS.hh"
...

#endif
#endif
```

In this method, forward definitions are achieved by defining `DEFER_CLASSNAME` prior to including `CLASSNAME.hh`, and then doing a 'full' include of `CLASSNAME.hh` after the definition of the class.

Using this method means that classes simply include any other class they depend upon. If the dependency is only with references and pointers, then the included class should first be deferred to obtain the forward definition, with the full inclusion appearing after the class definition.

A.2 Build Process

A.2.1 The Makefile

The build process is controlled by the project's Makefile. It contains definitions for system dependent items, such as the C++ compiler and its options, library locations and any optimisations.

A.2.2 Dependencies

Incremental building is where only the parts of the simulation which have changed are rebuilt, which makes recompiling much faster. However, if class `A` uses class `B` and the `B.hh` file is changed, then both `A.cc` and `B.cc` must be recompiled. The dependency system automatically works out these dependency relations, and encodes them in Makefile format.

Makefile dependencies are not transitive; if `A` depends on `B` and `B` depends on `C` then `A` depends on `C` only if it is explicitly listed as a dependency. The C++ compiler `-M` option will process a `.cc` file and output its complete Makefile dependencies, consisting of all the files it includes. However, the M4 code generation techniques described in Section A.3.2, mean that not all the source files exist when the system begins building. To avoid this problem, as each `.cc` and `.hh` file is listed as a dependency it is 'touched' by the Makefile. This updates its timestamp and fools make into thinking that this file has been changed. This is equivalent to finding the transitive closure of the dependency graph. Generated files which don't exist when they are listed as a dependency are generated.

A.3 Generating Code and Reducing Code Repetition

Code repetition is reduced in the simulation through the use of *generated code*. This is code which exists in a templated form in terms of various parameters. When the simulation is built this code is expanded into fully complete code which is subsequently compiled. This means that different pieces of code which are identical except for a few details appear only once. This has the usual benefits associated with reduced code repetition, such as improved maintainability, lower complexity and easier extensibility.

Code is generated in the simulation in two different ways, with C preprocessor macros and with the GNU M4 macro processor. They are applied to different situations, with different requirements, and will be discussed in the following subsections.

A.3.1 The C preprocessor

The C preprocessor (CPP) allows the definition of macros which are similar to functions except that their contents textually replace the macro 'call'. For example, the macro

```
#define COORD(x, y, w) ((x) + (y)*(w))
```

can be used as

```
matrix[COORD(j-k, i, width)]
```

and the CPP will replace this text with

```
matrix[((j-k)+(i)*(width))]
```

`IdTable` and `KeywordTable`

The implementation of the `IdTable` and `KeywordTable` 'classes' use this idea. These classes are simply dictionaries with particular types of keys (identifiers and keywords, respectively) and any type of value. Unfortunately C++ does not allow templated typedefs[13], so these types are implemented as the macros

[13] However, a workaround exists:

```
template <typename T> class Table {
    typedef leda_dictionary<Id_t, T*> Id;
```

```
#define IdTable(T) leda_dictionary<Id_t, T##* >
#define KeywordTable(T) leda_dictionary<Keyword_t, T##* >
```

For example, the 'class' `IdTable(Event)` is actually of the type `leda_dictionary<Id_t, Event*>`, which is an `IdTable` of `Event`s. This follows the type abstraction ideas in Section A.1.3.

Inherited constructors and instantiators

In C++, constructors are not inherited. This means that each derived class must duplicate the definition and implementation of the constructors of the base class. This leads to very poor code repetition, and the possibility of having derived classes which aren't properly substitutable. CPP macros are used here to generate the definitions and implementations of the constructors and instantiators for derived classes of the major base classes: `Event`, `Logfile`, `Entity` and `AllocationStrategy`. The result is that two lines in the definition and implementation of derived classes gives them a standard, consistent method of construction and instantiation.

For example, the definitions for the constructors of the various `Event`-derived classes are given by the macro

```
#define EVENT_CONSTRUCTOR_DEFINITIONS(CLASSNAME) \
public: \
CLASSNAME(); \
CLASSNAME(Time_t time, Id_t id); \
CLASSNAME(Time_t time, Id_t id, WordList options); \
CLASSNAME(const CLASSNAME &e); \
virtual ~CLASSNAME();
```

```
        typedef leda_dictionary<Keyword_t, T*> Keyword;
};
```

which is subsequently used as `Table<Event>::Id` for an `IdTable` of `Events`. This is considered unacceptably complicated in light of the macro solution.

The derived class `EventDerived` defines its constructors by including the line

```
EVENT_CONSTRUCTOR_DEFINITIONS(EventDerived)
```

in its definition.

A.3.2 M4

The M4 macro processor is a very powerful text processing utility. It allows text to be composed from macros, in a similar fashion to the CPP. However, macros may be recursive and M4 provides many useful built in macros for common tasks; using M4 is quite similar to programming. M4 is used to generate entire source files from a single 'template' M4 file; as mentioned in Section A.2 the project Makefile contains rules for automatically generating these source files when they are needed.

M4 files are used in two distinct ways in the project. The first is to provide central *registries* of prototypes of various classes, the second is to provide derived classes which are identical except for some small details.

References

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* (Addison-Wesley, 1995)

[2] J. O'Rourke, *Computational Geometry in C, Second Edition.* (Cambridge University Press, New York, 1998)

[3] K. Mehlhorn, S. Näher, *LEDA: A Platform for Combinatorial and Geometric Computing.* (Cambridge University Press, Cambridge, 1999)

[4] A. S. Tanenbaum, *Computer Networks, Second Edition.* (Prentice Hall, Englewood Cliffs, 1988)

[5] V. Estivill-Castro and M. E. Houle. "Robust Distance-Based Clustering with Applications to Spatial Data Mining." *Algorithmica (Special Issue: Algorithms for Geographical Information).* To appear in 2000. `http://www.cs.usyd.edu.au/~meh/papers/gisclust.ps.gz`