Evaluation of Virtual World Systems

Kevin Pulo, Michael E. Houle
Basser Department of Computer Science
The University of Sydney, NSW, Australia, 2006
{kev,meh}@cs.usyd.edu.au

Abstract

A virtual world system is an artificial environment, created inside a computer, which mimics some aspect of the real world. These systems are multiuser, allowing many people to be present and to interact simultaneously in the virtual world. The performance of virtual world systems is important because the quality of the user's experience depends on the responsiveness of the system. This paper looks at issues involved in evaluating the performance of such multiuser virtual world systems. A flexible, object-oriented framework is presented for supporting these evaluations experimentally. As an example of its usage, this framework is applied to real virtual world system and some results are presented and discussed.

1. Introduction

Virtual world systems are characterised by client-server network systems aimed at allowing multiple clients to interact with each other and a virtual 'environment'. Virtual world systems span many different areas, from multiplayer games, Internet chat rooms and cellular mobile phone systems, to collaborative computing, video conferencing and tele-presence systems, to emerging e-commerce applications such as real-time auctions and real-time stock day trading.

Virtual world systems aim to be as realistic as possible. One of the most important aspects of this is the responsiveness of the system. This paper focuses on evaluating the performance of the system as a whole, but in particular the network and how it affects the responsiveness of the system.

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

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

In a single-server system, client interactions travel via the server. However, in multi- and distributed-server systems, client interactions must be sent via more than one server if the clients are connected to 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 reduce the interserver latency.

Of particular importance is the issue of *scalability*, which is how well the system performs as the number of clients increases. Single-server systems are limited in scalability by the amount of hardware a single machine can have, whereas multi- and distributed-server systems are limited by the interserver network. In addition, servers may be added to multi- and distributed-server systems without disrupting the virtual world, which further aids the scalability.

In general, each user in a virtual world will not want to interact or communicate with every other user, particularly in large virtual worlds. Rather, users tend to communicate with only a small group of other users at a time. Arranging the system such that most of the clients in a group are on the same server results in less interserver communications, which improves the performance and scalability of the system.

Various approaches to lowering the total amount of communication have been attempted. The SimNet system [1] works by broadcasting packets and allowing clients to receive only those packets they consider relevant. The NPSNET system [2] uses *Areas of Interest* to partition the clients into disjoint subsets, and then uses multicast groups to allow the clients in these subsets to communicate. Most client-server based systems developed to date are limited in the methods they implement or in their problem domain. For example, RING [3] uses visibility algorithms to compute potential visual interactions between clients, and VENUS [4] deals with issues encountered when using satellites for part of the communication between the clients and servers. Some systems aim to be more general, most notably the WAVES system [5] and its initial implementation, HIDRA [6], and the VOID shell [7], which provides an object-oriented toolkit for developing client-server virtual world systems. By contrast, the system presented in this paper provides the framework not for an actual virtual world system, but rather for a simulation of one. Furthermore, the emphasis is on using the simulation to experimentally evaluate the performance of the simulated system.

In this paper, the *neighbourhood* of a client is the subset of all other clients with which the client may interact. Performance gains can be achieved when the neighbourhood is a strict subset, particularly when the neighbourhood is of constant size.

In multi- and distributed-server systems, each client is *assigned* to a server which handles communication with the client. The collection of assignments for all clients is called an *allocation*. Allocations are *static* if client assignments are fixed when clients enter the virtual world, and are *dynamic* if client assignments may change over time. An *allocation strategy* is an algorithm for creating allocations; that is, determining each client's assignment.

This paper concentrates on these characteristic allocation problems found in multi- and distributed-server virtual world systems. In particular,

- allocation strategy performance — methods for evaluating the performance of allocation strategies and for comparing allocation strategies are not obvious, and
- developing allocation strategies — developing good algorithms for allocation strategies is generally a difficult task.

A flexible object-oriented framework for approaching these problems experimentally is presented. It allows a wide range of systems to be modelled and simulated without requiring large amounts of development effort. Input to the simulation may be artificially generated or may be actual data recorded from a real virtual world system.

Section 2 describes the main design of the framework. Section 3 describes the application of the framework to a real virtual world system as an example of its usage, and Section 4 presents and discusses the results obtained with this sample application. Finally, Section 5 discusses some possible improvements.

2. Framework

The framework is composed of several main classes, each with an associated class hierarchy. They are Entities, Events, Logfiles, Costs, Communication Strategies and Allocation Strategies. The extensive use of object-oriented techniques, in particular the design patterns found in [8], mean that all of these class hierarchies are easily extended. This gives the framework the flexibility it needs to be applied to many different types of virtual world systems.

2.1. Entities

Instances of entities exist in the simulation, and correspond directly to actual things in the virtual world system. There are three types of entities: clients, servers and network packets.

A client entity represents a client connected to the main network of the virtual world system, which allows a user to participate in the virtual world. 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, which means that it may change throughout the simulation in order to best accommodate the changing state of the client.

Clients maintain their state in an object of the class ClientState. This uses the *Memento* design pattern [8] to store and transmit mutable information about the client, such as its location in the virtual world or perhaps the clients with which it is directly interacting.

A server entity represents a server in the network which gives clients access to the virtual world. The clients assigned to a server are called its *assigned clients*. Each server has a recommended number of clients, called its *capacity*. A server with more assigned clients than its capacity allows is said to be *overfull*, and the clients causing this situation are called *overfull clients*. Each overfull client contributes a scalar cost to the current cost, where costs are described in Section 2.4.

A packet entity represents a packet in the network. They are used to transfer atomic messages between servers and clients. While clients may only send packets to their assigned server, servers may send packets to any of their assigned clients or to any other server. If a server wishes to send a packet to a client not assigned to it, the packet must first travel to the client's assigned server.

Packets record only their source entity, not their destination (although they are still received). The destination is implied by the allocation, making packets independent of allocation. This is important as it allows several allocation strategies to be compared on a single scenario.

There are two specific types of packets, UpdateState packets and Assign packets. UpdateState packets carry notification of a given client's change of state. They store an instance of the client's new ClientState object. Incremental storage could be achieved by using the *Composite* design pattern [8] to recursively store the differences in state with respect to some other ClientState object. This other ClientState object could be explicit or implicit, such as the client's last known state.

Assign packets carry notification of a client's assignment to a particular server. They are sent from the server which has determined the assignment (usually the server to which the client is being assigned). Exactly where the packets are sent depends upon the method used to inform servers of client assignment changes. This is handled by selecting an assignment cost which reflects the method being used; these costs are described in Section 2.4.

2.2. Events

The simulation is based on instantaneous events, that is, events which occur at a particular time with no duration. (Non-instantaneous events can be modelled by having instantaneous events when they start and finish.) These events are stored in a priority queue based on increasing event time, and 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.

The available event types are the creation and destruction of an entity, the sending and receiving of a packet entity, and setting the value of a cost.

2.3. 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 fields include the time of the event, the identifier of the entity being acted upon, the type of event. 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 common text-manipulation utilities 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.

Client state logfiles. Client state logfiles record the state of the clients as the virtual world progresses. A client state logfile is a complete record of the states of the clients. A client state logfile can be "played back" (by executing its events) to reconstruct an exact replay of the virtual world from which it was recorded. It records the actions of clients (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 data sent from the clients to the servers.

The specific events allowed are the creation of client and UpdateState packet entities, the destruction of entities, and the sending and receiving of packets.

Architecture logfiles. Architecture logfiles record the architecture used to host a virtual world system. This architecture is the server-side "hardware" setup on which the events of client state logfiles may be executed. It includes the servers, their network connections, their associated costs, limits and other properties particular to the setup.

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 is useful for testing situations such as how the system performs when a server is destroyed at a certain point in time.

The specific events allowed are the creation of server entities, the destruction of entities, and setting the values of costs.

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

The specific events allowed are the creation of Assign packet entities, the destruction of entities, and the sending and receiving of packets.

Downsampling. The event and logfile design has the disadvantage that logfiles can become very large and slow to process if they have a high resolution. This is avoided by *downsampling* the logfile, that is, not recording every event to the logfile. For a given packet, the number of *missing equivalent packets* is recorded. This is the number of functionally equivalent packets omitted from the logfile due to downsampling.

Client estimation is the process of modelling the client's state when downsampling means that a definitive state is not available. This is usually accomplished by the interpolation or extrapolation of known data, and may be as simple as a linear model. The *Strategy* design pattern [8] is used with the ClientEstimator class to define an interface for obtaining estimates of client states. A ClientState object stores an instance of a ClientEstimator object, which decouples the estimation model from the state itself. This allows for seamless estimation of data lost to downsampling.

2.4. Costs

In the course of running the simulation, actions incur penalties. These penalties are called *costs,* and are made up of two components, *TimeCosts* and *DataCosts.* TimeCosts indicate that an action has taken some period of simulated time, or (equivalently) has been delayed by some period of simulated time. DataCosts indicate that an action has caused some amount of network traffic. As the simulation runs it maintains a "current" cost and a "total" cost. The current cost is a measure of the cost present in the system at the current point in the simulation, and the total cost is the integral of the current cost with respect to simulation time.

Different kinds of costs depend upon different factors. Different instances of costs will depend on these factors in differing quantities. For example, the cost of sending a packet may depend linearly upon the size of the packet, whereas the cost of a client joining the network may be a constant. In order to accommodate these situations, costs are specified as an array of coefficients. The number of coefficients and the values they are in terms of are specific to each kind of cost. The actual value of an instance of a cost is found by "flattening" the cost, that is, multiplying each of the cost's coefficients by the value it is in terms of, and then summing these.

Using the *Template Method* design pattern [8], the Cost class implements costs generally while allowing subclasses to specify the number of coefficients and what they are in terms of. There are presently 3 types of costs, the ScalarCost, UpdateStateCost, and AssignCost.

The ScalarCost has only one coefficient; it is the special case of a constant cost which doesn't depend upon anything.

The UpdateStateCost is incurred when a client notifies its server that its state has changed. It is specified in terms of the number of neighbours, the number of neighbours assigned to other servers, and the number of distinct servers the neighbours are assigned to.

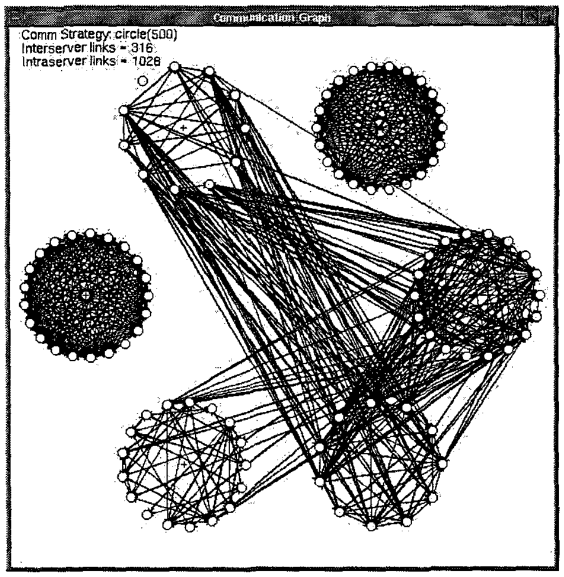
The AssignCost is incurred when a client is assigned to a server. For a client c being deassigned from server s_d and assigned to server s_a, the AssignCost is specified in terms of the number of clients assigned to s_a (excluding c), the number of clients assigned to s_d (excluding c), the total number of clients, and the total number of servers.

2.5. Allocation strategies

The Strategy design pattern [8] is used for implementing allocation strategies. The AllocationStrategy class abstractly defines the interface of an allocation strategy and is subclassed for the actual implementations of the various allocation strategies.

Allocation strategies are notified of the execution of events by being passed the executed event. The base AllocationStrategy class provides an internal interface which determines the event type, and then passes the appropriate information from the event to an event-specific method. This means that concrete implementations of allocation strategies need only override the methods for the event types they are interested in.

Allocation strategies 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, as described in Section 2.3. They are added to the currently running simulation (usually scheduled for immediate execution) and may also be stored in an allocation logfile on disk.



Comm Strategy: circle(500)
Interserver links = 316
Intraserver links = 1028

Figure 1. An example of a communication graph. Each circle of nodes and the edges incident to those nodes are drawn in the same colour.

2.6. Communication strategies

Communication strategies are methods for determining the *neighbourhoods* of all clients. They are similar to allocation strategies; they use the Strategy design pattern [8] to abstractly define the communication strategy interface, and they are notified of the execution of events. However, the notification method returns no value. Rather, the communication strategy can be queried about the neighbourhood of any given client. Neighbourhoods are represented simply as an unordered collection of clients.

The simulation displays a communication graph, in which the nodes represent clients, and two nodes are linked by an edge if the corresponding clients are neighbours. The graph is drawn as shown in Figure 1, where each circle of nodes are the clients assigned to a particular server, and are drawn in the same colour. This allows the user to very easily see areas with many interserver edges or intraserver edges. This layout scales poorly as the number of clients and servers increases, but better layouts are possible. The problem of finding good layouts for communication graphs is outside the scope of this paper.

3. Experimental design

3.1. Virtual world system

As an example of how the framework may be used, this section describes its application to a sample virtual world system, a large online multiplayer game presently under development at a leading game studio.

Development in multiplayer games is currently tending towards increasing from only tens of simultaneous players up to tens of thousands. This means that they are an emerging source of real-time virtual world systems for large numbers of clients.

The game is spatial in nature, with players represented by avatar characters in an immersive first-person perspective 3D environment. The simulation displays the locations of players on a 2D map, called the arena, illustrated later in Figure 3 in Section 3.6.

3.2. Network topology

The network topology is two-tiered, with a group of servers on a high-speed Local Area Network (LAN), and clients connecting to these servers via the Internet, illustrated in Figure 2. Note that for client 2 to interact with client 6, packets must travel via servers C and D.

This is a particularly natural and simple topology, Funkhouser [9] gives a more detailed investigation of various network topologies for virtual world systems, including experimental results.

3.3. Input data

Recorded logfiles of actual gameplay were not available at the time of publishing. The technical aspects of creating such logfiles is a separate problem which lies outside the scope of this paper. As a result, the input data was generated with a small and simple simulation of player behaviour, called *bots*.

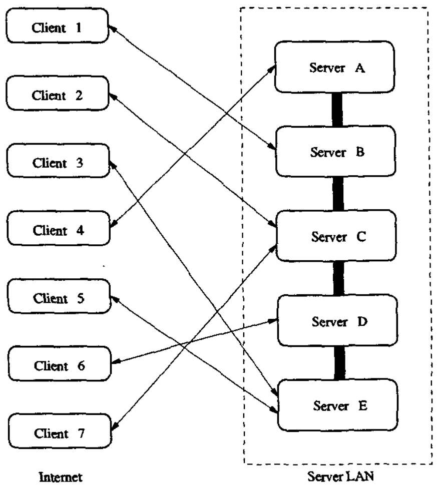


Figure 2. The network topology used by the simulation. Clients 1–7 are directly connected to Servers A–E.

A set 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 at that station for a random period of time between that station's minimum and maximum wait times. While at a station, the bot wanders randomly, staying within the station's "wander distance". When a bot has finished waiting, it randomly chooses an adjacent station and travels to that station, where the process repeats.

3.4. Client state logfiles

Three station sets were used, with varying station and edge densities. Illustrations of the station sets give only limited insights for their relatively large size, so to save space they are instead described qualitatively.

The `city` station set consisted of 9 stations in 3 groups, each in a corner of the arena. There were several edges within each group, an edge between the first and second groups, and an edge between the second and third groups.

The `wells10` station set consisted of 10 stations in 4 groups distributed across the arena. There were several edges within each group and two edges between each group and two of its neighbouring groups.

The `wells50` station set consisted of 50 stations distributed evenly throughout the arena, with the edges creating a triangulation of the stations. This gives a constant degree for each station on average. Figure 3 was created using `wells50`.

For each station set, a client state logfile was generated with 100, 200, 500, 1000 and 2000 bots, giving a total of 15 client state logfiles. Each had a duration of 150 seconds and was downsampled to a resolution of 10 seconds.

3.5. 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 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. 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.

The UpdateStateCost depended equally on the number of neighbours and the number of neighbours on other servers, which models the UpdateStatePackets being sent to the neighbours via their assigned servers. The AssignCost depended equally on the number of clients in the assigned and deassigned servers, which models the clients being notified of the change in assignment. The OverfullServerCost was a substantially large constant per overfull client.

3.6. Allocation strategies used

Two allocation strategies were compared. The first is a simple random allocation strategy, which statically assigns clients to a random server, giving quite bad allocations. The second is an allocation strategy called *cellular*, which gives better allocations.

The cellular allocation algorithm is based on the k-Means clustering heuristic [10]. For each server, a *representative* point is maintained which is the mean of the locations of the server's assigned clients. Each client is assigned to the server whose representative point it is nearest. This divides the arena into 'cells', shown in Figure 3, where each server controls the clients in the cell containing its representative. Assignment events occur when clients change cells. Initially, clients are randomly assigned to empty servers until every server has at least one client.

3.7. Communication strategies used

Two communication strategies are implemented and used.

The circle communication strategy chooses the neighbourhood to be the clients which lie within a circle of radius r centered on the location of the client. The value of r is a parameter of the strategy, and was chosen to be 0.1, 0.15 and 0.2 times the width of the station set. (Each station set had an aspect ratio of approximately 1.)

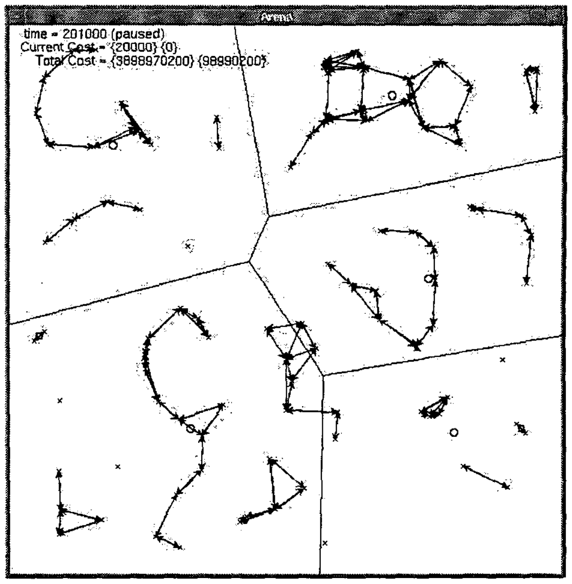

Arena
time = 201000 (paused)
Current Cost = {20000} {0}
Total Cost = {3898970200} {98990200}

Figure 3. An example of how the cellular allocation strategy divides the arena into cells, with each server managing a single cell. Clients are represented as small crosses ('×'), representative points small open circles ('○'), and clients are joined with an edge where they are neighbours.

The u-nearest communication strategy chooses the neighbourhood to be the u nearest neighbours [11] of the client. The value of u is a parameter of the strategy, and was chosen to be 5 and 10.

3.8. Simulation execution

For each pair of client state and architecture logfiles, the simulation was run for all combinations of the allocation and communication strategies described in Sections 3.6 and 3.7.

The experiment had 5 independent variables — station set, number of clients, server load, allocation strategy and communication strategy. The number of scenarios is

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

The simulation was run on each of these scenarios, and the resulting allocation logfile 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 overfull clients.

4. Results and discussion

The 5 independent and 5 dependent variables give the results a total of 10 dimensions. This high dimensionality means that analysing the results directly is both prohibitive and outside the scope of this paper. Instead, smaller sections of lower dimensionality have been analysed. A sample of the most interesting results is presented, illustrating some of the insights possible with the framework.

4.1. Total costs

Since the cost system has been designed to combine many factors, one of the best results to consider is how the total cost progresses throughout the simulation.

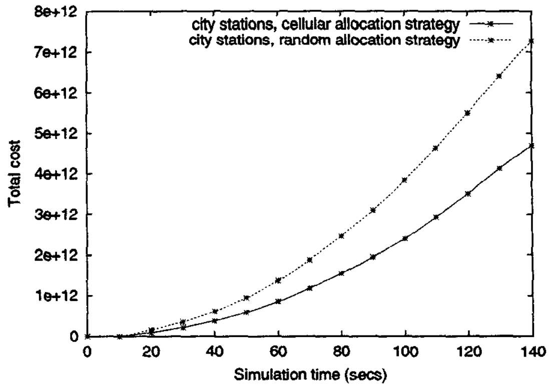
Consider the scenarios using the `city` stations, 500 clients, circle communication strategy with radius of 0.1 times the dimensions, server load of three-quarters full (24) and both the cellular and random allocation strategies. Figure 4(a) shows the progression of the total cost throughout the simulation. 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 4(b) 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 becomes 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.

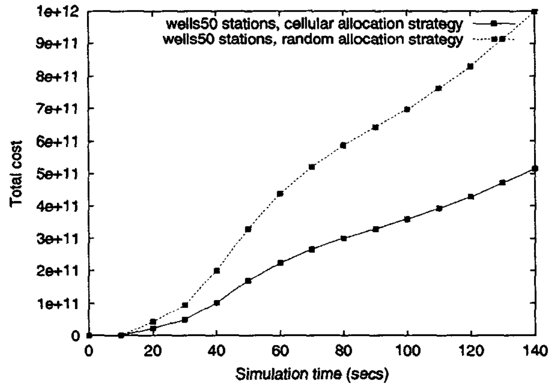
Finally, Figure 4(c) shows the data from Figures 4(a) and 4(b) on a single set of axes. This clearly shows that the cost for both allocation strategies (and their difference) is much less in the `wells50` scenario compared to the `city` scenario. As before, this can be attributed to a more even distribution of clients in `wells50` compared to `city`.

4.2. Overfull clients

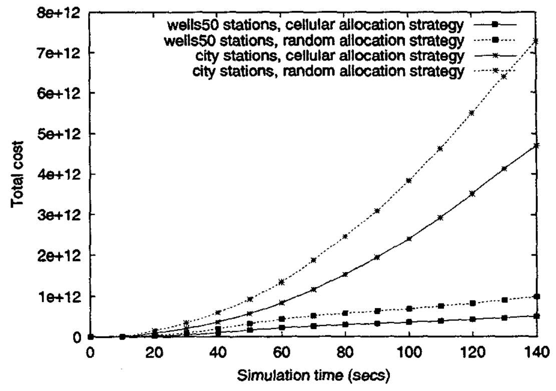
One consideration which is not present in the cellular allocation strategy is the capacity of servers. The result is that the cellular strategy might allow a small number of servers to dominate, allocating many clients to those servers and making them badly overfull.



(a) city stations

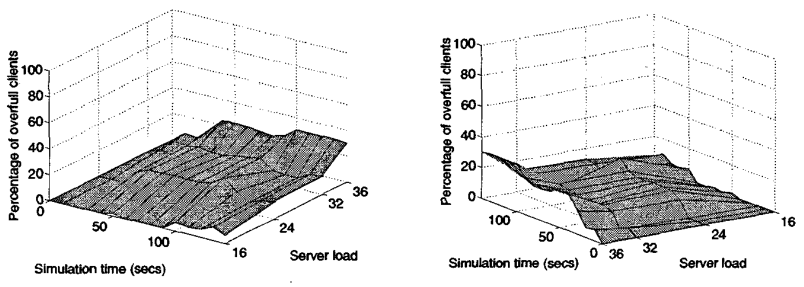


(b) wells50 stations



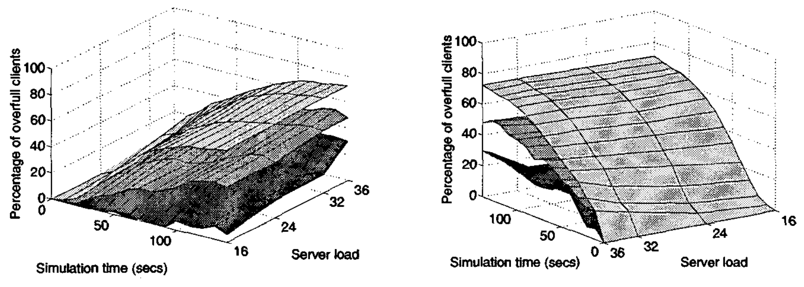
(c) both city and wells50 stations

Figure 4. Total cost versus simulation time for the cellular and random allocation strategies.



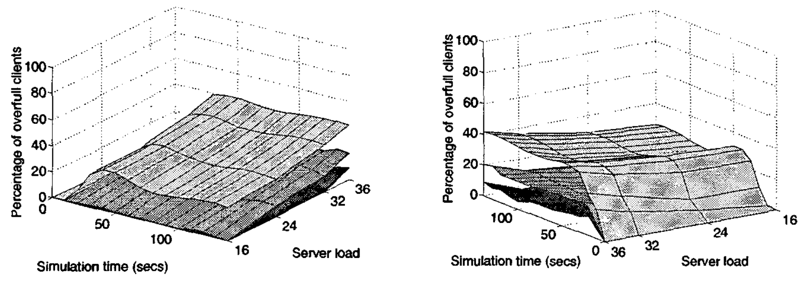
(a) front view (b) back view

Figure 5. The percentage of overfull clients versus simulation time, versus server load, for 100 clients using the city stations.



(a) front view (b) back view

Figure 6. The percentage of overfull clients versus simulation time, versus server load, using the city stations. The darkest shade is 100 clients, the next darkest is 500 clients and the lightest shade is 2000 clients.



(a) front view (b) back view

Figure 7. The percentage of overfull clients versus simulation time, versus server load, using the wells50 stations. The darkest shade is 100 clients, the next darkest is 500 clients and the lightest shade is 2000 clients.

Consider the case of the cellular allocation strategy, `city` stations, 100 clients and circle communication strategy with radius of 0.1 times the dimensions. Figure 5 shows how the fraction of clients which are overfull changes as the simulation progresses and for the various server loads. It shows that this fraction of overfull clients slowly increases as the simulation progresses and as the servers become increasingly loaded.

Figure 6 adds the scenarios for 500 and 2000 clients to the plot shown in Figure 5, 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 where server capacities are concerned, the cellular allocation strategy performs poorly as the number of clients increases.

Figure 7 is similar to Figure 6 except that it uses the `wells50` stations. Now as the number of clients increases so does the fraction of overfull clients, but not to the degree found with `city` stations. This time approximately 40% of the 2000 clients are overfull at the end of the simulation. This suggests that perhaps the cellular allocation strategy performs somewhat better when the clients are more evenly distributed.

Another observation is that the fraction of overfull clients for each of the 100, 500 and 2000 client scenarios stops increasing and remains constant after a short initial period. This is similar to the observation made in Section 4.1 where the total cost associated with `wells50` stopped accelerating and continued linearly. It can also be attributed to the even distribution of clients, when it is unlikely that the number and size of large servers can continue to grow.

5. Future work

5.1. Combining allocation strategies

One of the strengths of the Strategy design pattern [8] is that it allows its components to be easily combined. An allocation strategy called Multi could be written which maintains a set of other "sub"-allocation strategies. Whenever Multi is notified of an event it in turn notifies the sub-allocation strategies, receiving a set of assignments from each. It then chooses which set of assignments to return.

This decision step could also make use of the Strategy design pattern [8], allowing the exact method used to be determined by an "AssignmentComparator" object. For example, these comparators may choose a set of assignments by examining their immediate cost, some longer term projected cost estimate, how recently the clients involved were previously assigned, or unrelated conditions such as the total number of clients.

5.2. Better allocation strategies

An obvious and easy improvement which could be made to the cellular allocation strategy is to only allow a client to join a server if it does not cause that server to become overfull. It would be interesting to see how this would affect the performance of the system.

More sophisticated allocation strategies would also be useful. For example, [10] describes a clustering method which is substantially better than that used by the cellular allocation strategy, and so this could be used as a basis for a better grouping of clients onto servers.

5.3. Hybrid communication strategies

Both the circle and u-nearest communication strategies have advantages and disadvantages. The circle method has the disadvantage that the number of neighbours depends on the density of the clients, while it has the advantage of a constant maximum neighbour distance. The u-nearest method has the disadvantage of the maximum neighbour distance depending on the density of the clients, while it has the advantage of a constant number of neighbours.

Based on this, a hybrid communication strategy combining the best aspects of both may be useful. For example, the strategy may take only the first u neighbours (or less) which are within a given radius r. This would have the advantage of both a constant number of clients and a constant maximum neighbour distance, and so it would be interesting to see how well it performs.

5.4. Interactivity

The simulation displays the arena and communication graphs graphically. The usefulness of the simulation in exploratory situations could be greatly improved by adding graphical statistics displays and interactivity features. This would allow the user to modify the simulation as it is running, and to dynamically see the effects of various operations. The possible operations could be as diverse as adding or removing a server, changing allocation strategies, or forcing a set of clients to behave in a particular way.

6. Acknowledgements

Prof. Peter Eades and Prof. Bernard Pailthorpe provided invaluable feedback and helpful advice with the writing of this paper.

References

[1] J. Locke, *An Introduction to the Internet Networking Environment and SIMNET/DIS*, Technical Report, Computer Science Department, Naval Postgraduate School, August 1993.

[2] M. Macedonia, M. Zyda, D. Pratt, D. Brutzman, P. Barham, *Exploiting reality with multicast groups: A network architecture for large-scale virtual environments*, Proceedings of IEEE Virtual Reality Annual International Symposium (VRAIS) 1995, pp 2–10.

[3] T. A. Funkhouser, *RING: A Client-Server System for Multi-User Virtual Environments*, Proceedings of the 1995 Symposium on Interactive 3D Graphics, Monterey CA., pp 85–92. ACM SIGGRAPH, March 1995.

[4] S. Udani, *VENUS: A Virtual Environment Network Using Satellites*, Doctoral Dissertation, Department of Computer Information Science, University of Pennsylvania, June 1999.

[5] R. Kazman, *Making WAVES: On the Design of Architectures for Low-end Distributed Virtual Environments*, IEEE Virtual Reality Annual International Symposium (VRAIS) 1993, pp 443–449.

[6] R. Kazman, *HIDRA: An Architecture for Highly Dynamic Physically Based Multi-Agent Simulations*, International Journal in Computer Simulation, pp 149–164, May 1995.

[7] V. Cahill, A. Condon, S. McGerty, G. Starovic, B. Tangney, *The VOID Shell: A Toolkit for the Development of Distributed Video Games and Virtual Worlds*, Proceedings of the First International Workshop on Simulation and Interaction in Virtual Environments (SIVE) 1995, pp 172–177.

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

[9] T. A. Funkhouser, *Network Topologies for Scalable Multi-User Virtual Environments*. In Proceedings of the Virtual Reality Annual International Symposium. VRAIS'96 (March 1996), IEEE Computer Society, IEEE, pp. 222-228.

[10] V. Estivill-Castro, M. E. Houle. "Robust Distance-Based Clustering with Applications to Spatial Data Mining." *Algorithmica (Special Issue: Algorithms for Geographical Information)*. Springer-Verlag, New York, 2001 (to appear).

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