BigWorld Technology Server Whitepaper

BigWorld Technology 2.0. Released 2010.

Software designed and built in Australia by BigWorld.

Level 2, Wentworth Park Grandstand, Wattle St
Glebe NSW 2037, Australia
www.bigworldtech.com

Table of Contents

Chapter 1. Introduction

BigWorld Technology is comprised of server, client and tools components which are used together by customers to build massively multiplayer games and virtual worlds. This document reviews the key advantages of using the BigWorld Server as an MMO backend. The document starts by reviewing the server architecture, continues with different customers' deployments and ends with a discussion of lab testing of the server scalability and fault tolerance features.

Customers and evaluators are encouraged to refer any question regarding this document to their BigWorld business manager or the BigWorld support team.

Chapter 2. Architecture Overview

2.1. Structural Overview

The BigWorld Server is made up of a set of processes which are used to manage one game shard. These processes are typically run on a cluster of networked machines. The BigWorld Server provides a scalable, reliable game server for an arbitrary number of concurrently connected users. The size of the shard and therefore the number of concurrent users (CCUs) can scale based on customers requirements. Some customers prefer to install multiple BigWorld instances, each dedicated to a different shard, while others deploy one shard for their entire game.

Each instance of the BigWorld Server has multiple processes. The bulk of the cluster machines are dedicated to running CellApps. CellApps run the majority of the actual game code and load is dynamically balanced across them in real-time. Clients do not connect directly to the CellApp machines, rather, each client connects to a BaseApp which acts as a proxy between the client and whichever cell machine its player entity happens to be on at the time. BaseApps run on machines with public IP addresses and essentially act as the firewall for the server cluster. Most of the machines that are not dedicated to running CellApps are dedicated to running BaseApps.

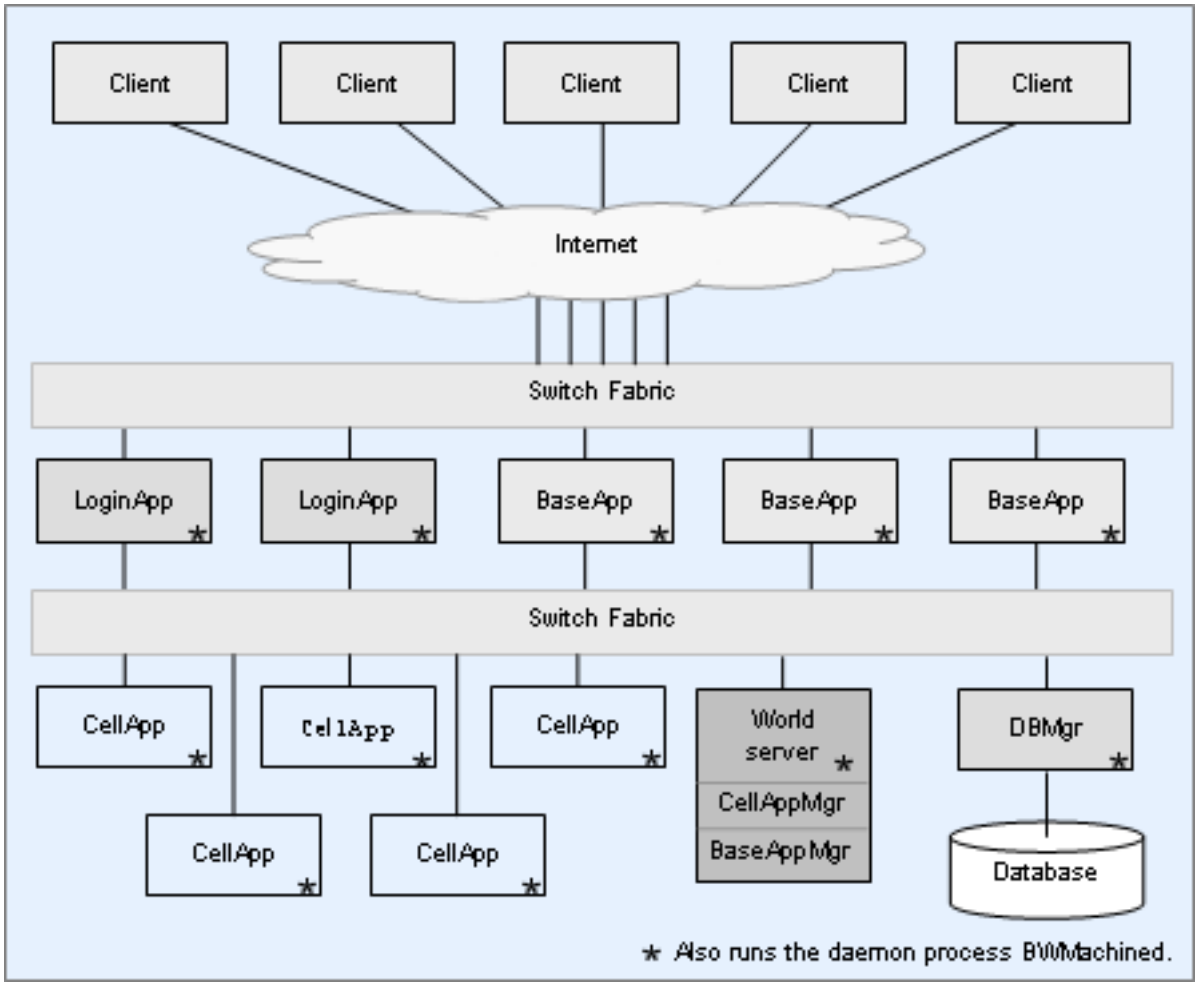
The CellAppMgr and BaseAppMgr are singleton processes that manage the load balancing for CellApps and BaseApps respectively, as well as providing other administrative functions.

The LoginApps validate client logins and are the only processes other than BaseApps which need to run on machines with public IP addresses. The login transaction and all subsequent communication from client to server is encrypted.

The Database Manager (DBMgr) stores persistent data in a MySQL database, and receives periodic back-ups of the server game state. This state is kept consistent and can be restored in the event of hardware or system failure.

To understand the relationship between CellApps, BaseApps and the client, it is important to understand one aspect of how games using BigWorld Technology are architected: game entities are implemented as distributed classes. Each entity can have any combination of Cell, Base, and client parts, which run on the CellApps, BaseApps and clients respectively.

A high level diagram of the BigWorld Server components and how they relate to each other is given in the diagram below:



High-level diagram of the BigWorld Server

2.2. Key Features

2.2.1. Load-Balancing

The BigWorld Server dynamically allocates its resources to achieve a dual objective:

1. Use as few machines as are needed to keep average load below a configured threshold, and;
2. Distribute load across active machines as evenly as possible.

This results in an economical, efficient use of resources and a server state that is well-prepared for sudden spikes in load.

The primary mechanism by which we implement this dynamic load-balancing scheme is the dynamic allocation and resizing of Cells. Each CellApp controls a number of cells, each of which is responsible for a specific part of a space. The area of each game space is divided between one or more cells as is necessary to keep the load on the controlling CellApps below a configured threshold. The areas controlled by cells do not overlap, and the cells collectively span all areas of all spaces in the game world. See SpaceViewer Screenshot on page 21 below for an example of how a number of cells share the load for a single space.

As the load on a particular cell increases, its total area is dynamically reduced by the CellAppMgr, and entities are offloaded onto adjacent cells to balance the load between available CellApps. As the average system load increases past the configured threshold for a particular space, the CellAppMgr will add a new cell and assign it to a CellApp that doesn't already have a cell in that space. As the average system load

decreases, the CellAppMgr will start to retire Cells, releasing their associated CellApps from the space, in order to maintain the average load at the configured threshold.

This aspect of the server's design is based on the heuristic that, generally, entities are interested in other entities that are spatially nearby. Since each cell controls entities in a continuous region of a space, in most cases the interactions between these entities can be resolved locally without having to communicate with another CellApp. This minimises network traffic and its associated (and significant) CPU usage, delivering maximum efficiency and server capacity.

While the load balancing algorithm for Cells is dynamic, the algorithm for BaseApps is currently static (although there is work underway to modify this behaviour), in that the base part of an entity is (usually) created on the least loaded BaseApp and is never offloaded. A reasonable turnover (i.e. rate of creation and destruction) in a system keeps the load balanced evenly over the available BaseApps.

The design for BaseApp load balancing is based on the assumption that the majority of game code is executed on the CellApp, so that the largest proportion of game code execution can be dynamically rather than statically load balanced. The Base part of a player entity spends most of its time forwarding packets between the client and its cell entity, and since the downstream bit-rate is held constant, BaseApp load tends not to fluctuate in the way that CellApp load can. To date, this static balancing scheme is more than sufficient to distribute load evenly.

Note

Ask a BigWorld business manager for a live demonstration of the server load balancing capabilities. This demonstration shows dynamic allocation and resizing of cells as a result of changing game conditions.

2.2.2. Scalability

An important feature of the BigWorld Server is that load scales roughly linearly with respect to number of entities. More specifically, for all other variables such as AI, frequency of activity, etc. held constant, increasing the number of entities (clients and NPCs) should produce a linear proportional increase in overall server load.

This is important because it means that a BigWorld Server has no "glass ceiling" for performance, where the amount of extra hardware required to progressively increase the entity capacity of the server would become prohibitive. Simply put, with a BigWorld Server, if you want to double the number of entities, you double the amount of hardware in the cluster -- The server will expand as much as the customer base requires for a predictable cost.

The BigWorld Server achieves these linear scaling characteristics by two main methods: controlling the amount of downstream bandwidth available to each client; and limiting the vision of each entity to a predetermnined range. Since a large proportion of total server CPU consumption is taken up by the formation and sending of packets, setting a maximum downstream bit-rate per client results in load that is linear to the number of clients once each client reaches this bandwidth limit.

Without these scaling mechanism in place, the server load would be quadratic in the number of clients: as the number of clients increases, the number of other entities each client can see will increase at roughly the same rate, giving an $O(n^2)$ complexity for downstream traffic and server load.

To make sure that clients are not disadvantaged by bandwidth limits, we prioritise updates for nearby entities when constructing downstream packets. Entities closest to the client avatar are given a greater share of the bandwidth and will be updated more frequently; entities further away are given less share of the bandwidth, and are updated less frequently. The client has filters to smooth the movement for entities between updates, so movement will appear smooth even for distant entities that have sparse updates. As the client approaches an entity that is far away, the update rate is increased smoothly so that interactions become more and more responsive.

Strictly speaking, server load should increase at a slightly super-linear rate once downstream bandwidth is fully utilised. Although the amount being sent per client remains constant, the algorithm that prioritises the entities and determines which ones are updated is logarithmic in the number of entities surrounding the client avatar. The coefficient of this logarithmic part of the complexity is expected to be very small relative to the linear part related to forming and sending packets.

This implies that having excessive entity density in a small area can have an impact on performance on the server. However, entity density is a problem not just for the server, but for the client as well. Having too high an entity density in one area will decrease the client performance from the additional rendering that is needed. A good game design should ensure that entity densities are kept at levels that are appropriate for ensuring good end-user experience. Our experience has been that having two thousand entities in a small area (a few hundred metres square) is handled well by the server, and if appropriate rendering is used on the client performance can be acceptable there also.

Note

The scalability capabilities of the server are demonstrated in the chapter "Feature Demonstrations" on page 23 in *Demonstration and Analysis* on page 19

2.2.3. Fault Tolerance

The BigWorld Server implements a variety of countermeasures against component or widespread machine failure. This is to ensure both data integrity and high availability of the server. This prevents the loss of game event history such as item trades, quests completed etc. and allows for an uninterrupted gaming experience.

2.2.3.1. High Availability - First Level Fault Tolerance

The aim of first level fault tolerance is to handle isolated server component or machine failures. When components or machines are found to be no longer reachable for whatever reason, the system recovers transparently to the user so that gameplay continues with as little disruption as possible.

CellApps implement fault tolerance by having each cell entity back-up its state to its Base entity at regular intervals. If the CellApp goes down for any reason, Base entities will detect that their cell parts are no longer contactable and re-create their cell entity on another available CellApp.

For BaseApps, fault tolerance is currently implemented by having the BaseApps back up amongst themselves. Each base entity is backed up to another BaseApp and is sent periodic backup updates. If a BaseApp goes down, the other BaseApps will restore the backed up base entities that were on that BaseApp.

A sentinel process called "Reviver" provides fault tolerance for the remaining server components (LoginApp, CellAppMgr, BaseAppMgr, and DBMgr). If any of these processes fail, the Reviver will detect it and immediately start a new instance of that process to ensure continuity of gameplay.

Note

Ask BigWorld business manager for a live demonstration of the server's high availability. This demonstration shows game continuity while killing a CellApp.

2.2.3.2. Preserving Game Data Integrity and Consistency - Second Level Fault Tolerance

The second level of fault tolerance deals with massive system-wide failures (e.g. power outages, disruption to network hardware etc) and is implemented by the DBMgr and the BaseApps. The aim of second level fault tolerance is to ensure data integrity rather than continued gameplay. By the time this stage of fault tolerance has been engaged it is safe to assume the server is no longer in a state where it can properly function.

The BigWorld Server continually conducts a rolling distributed backup of the game state to a distributed database. For critical state changes (such as item trades) the database write can be manually invoked to ensure the gamestate is preserved irrespective of where the rolling writeback is up to. Once the cause of the system-wide disruption has been identified and fixed, the game can be restarted from the game state stored in the database.

The second level fault tolerance mechanism is implemented using a distributed database system in the form of a secondary database for each BaseApp. This ensures scalability, and removes the DBMgr as a potential bottleneck. In addition, there is support for snapshotting the server state during online operation.

Chapter 3. Case Studies

This section discusses customers case studies. These case studies are based on real deployments. Due to commercial reasons, customer identifying details are not included in this document. We encourage you to discuss public customer information with BigWorld business managers.

3.1. Customer I

The game being deployed by this customer requires a very large single space (200 square kilometres). It also requires all gamers to be connected to the same space and be able to interact with each other.

Solution specific requirements:

1. Host a very large space.

2. Support unlimited number of gamers.

3.1.1. Selected Solution:

This customer choose to deploy a single BigWorld Server using a single network cluster. This BigWorld Server is scaled by adding additional machines to be used mainly as BaseApps and CellApps.

3.1.1.1. Solution Features

Feature Description	Feature Applicability
Server Fault Tolerance	This deployment provides a very high level of fault tolerance using the BigWorld fault tolerance capabilities. Each deployed BaseApp or CellApp is backed up by other BaseApps and the server can withstand multiple components crash without downtime.
How to Scale	The game server can scale as much as required by adding additional CellApps and BaseApps assuming reasonable entity densities are maintained.
Load balancing and efficient hardware usage	All hardware is being utilised. Load balancing is being managed by the single server.
Maintenance Effort	Maintenance of multiple large shards on multiple network clusters is relatively complicated. A MySQL DBA is recommended to manage the MySQL database.

3.1.1.2. Implementation

Resource Description	Total number of Resources
Total number of shards	1
Total machines per one shard	30
Number of CPUs per machine	4 (Quad Core machines)
Memory per machine	6 GB
Hard disk requirements	100 GB disk
Total CPUs for the deployment	120
DBMgr CPUs	4 (single machine)
BaseAppMgr CPUs	2
CellAppMgr CPUs	2
LoginApp CPUs	4
BaseApp CPUs	27
CellApp CPUs	81
Estimated machines cost (USD) (per shard)	40,000 - 50,000
Observed Max CCUs	20,000

3.1.2. Summary

This customer choose a single shard deployment. The deployed solution hosts a 200 square kilometre space with more than 20,000 CCUs and is currently available as an online game. Using the BigWorld Server solution this customer achieved a unique deployment not available from other games providers.

3.2. Customer II

The game being deployed by this customer hosts more than 100,000 CCUs. The game logic allows separating this solution into multiple shards.

Solution specific requirements:

1. Host more than 100,000 CCUs.
2. Sharding is possible for this game and interconnections between different shards are minimal
3. The customer has chosen to invest extra resources to host each shard on a separate network cluster; this provides higher availability, but lower hardware efficiency.

3.2.1. Selected Solution

This customer choose to deploy a Multiple Shard Deployment Using Multiple Network Clusters. This solution scales by adding additional machines for each shard or by adding more shards.

3.2.1.1. Solution Features

Feature Description	Feature Applicability to this deployment
Server Fault Tolerance	This deployment provides an extremely good level of fault tolerance as in addition to the standard fault tolerance capabilities of the BigWorld Server, a failure on one shard has no impact on the operation of other shards.
How to Scale	The game server can be scaled using two methods: Firstly, each shard can be scaled linearly by adding hardware to it. Secondly, additional shards can be added in dedicated network clusters.
Load balancing and efficient hardware usage	The customer choose to dedicate separate network clusters for each shard. This decision requires manually allocating servers to each shard while removing any interdependency between each shard. Customers can also choose a deployment using one network cluster hosting multiple shards, this solution will provide better scaleability than this solution while sharing similar capabilities.
Maintenance Effort	Maintenance of multiple large shards on multiple network clusters is relatively complicated. A MySQL DBA is recommended to manage the MySQL database.

3.2.1.2. Implementation

This example shows a typical shard which is part of a Multiple Large Shard Deployment Using Multiple Network Clusters.

Resource Description	Total number of Resources
Total number of shards	Unlimited
Total machines per one shard	10
Number of CPUs per machine	4 (Quad Core machines)
Memory per machine	6 GB
Hard disk requirements	100 GB disk
Total CPUs for the deployment	40
DBMgr CPUs	2
BaseAppMgr CPUs	1
CellAppMgr CPUs	1
LoginApp CPUs	2
BaseApp CPUs	8
CellApp CPUs	27
Estimated machines cost (USD) (per shard)	14,000 - 20,000
Observed Max CCUs	6,000 per shard, 120,000 for the deployment

3.2.2. Summary

This customer chose a multiple shard deployment. The deployed solution hosts around 120,000 players simultaneously. This deployment provides a world class solution and is still scaling to meet more gamers requirements.

Note

Deployment of this size usually requires either a dedicated deployment team or working with an experienced hosting company. BigWorld has multiple partners which are able to provide hosting solutions similar to the above solution.

3.3. Customer III

This customer game is in beta stages and requires up to 2000 CCUs

Solution specific requirements:

1. Simple solution to allow relatively easy deployment during beta and testing stages.
2. Easy scaling of the solution to the full scaled solution.

3.3.1. Selected Solution:

This customer choose to deploy a single medium sized BigWorld Server using a single network cluster. This solution support the required number of CCUs and can be easily scaled to either of the solutions appearing above or to any other BigWorld Server deployment.

3.3.1.1. Solution Features

Feature Description	Feature Applicability
Server Fault Tolerance	This deployment provides a very high level of fault tolerance using the standard BigWorld Technology fault tolerance features.
How to Scale	The game server can be scaled by adding machines or by adding additional shards, if game logic allows a multiple shard deployment.
Load balancing and efficient hardware usage	The BigWorld Server standard load-balancing features can be used for this deployment.
Maintenance Effort	Maintenance of a medium sized BigWorld cluster is relatively low, and usually doesn't require dedicated resources.

3.3.1.2. Implementation

Resource Description	Total number of Resources
Total number of shards	One
Total machines per one shard	6
Number of CPUs per machine	2 (Dual Core machines)
Memory per machine	2 GB
Hard disk requirements	100 GB disk
Total CPUs for the deployment	12
DBMgr CPUs	1
BaseAppMgr CPUs	0.5
CellAppMgr CPUs	0.5
LoginApp CPUs	1
BaseApp CPUs	2
CellApp CPUs	7
Estimated machines cost (USD) (per shard)	6,000 - 10,000
Observed Max CCUs	1,500

3.3.2. Summary

This customer choose a medium sized BigWorld Server deployment. The deployed solution hosts up to 1,500 CCUs and was used during the beta stages of the product. This beta succeeded and the game is now being officially released using a bigger deployment.

Chapter 4. Demonstration and Analysis

4.1. Introduction

The purpose of the following demonstrations and tests is to show scientifically that the features and asymptotic trends of the BigWorld Server described in *Architecture Overview* on page 7 hold true for real world deployments. Analysing real customer deployments as well as scientific test results allows for the creation of estimates for hardware costs for different server deployments as specified in the previous chapter.

It should be noted that in the large-scale tests, game entities running significant Artificial Intelligence (AI) routines were not used. This was a deliberate choice as the variance in complexity and efficiency of AI in a real game is potentially very large, and is determined almost entirely by the design of a given game as opposed to the BigWorld Server. This is also the main reason for the wide variance in CCUs quoted in *Case Studies* on page 13 . Since these variations are difficult to account for, they were excluded from these tests. As BigWorld already has multiple customers with games in beta and release stages, we have good statistics about the expected behaviour of these factors.

4.2. BigWorld Tools

The data displayed in the following tests was collected using BigWorld Server tools. The BigWorld server is shipped with a collection of web-enabled tools which are used to monitor and administer the server. The following is a short description of these tools:

4.2.1. Stat Logger

Stat Logger, as the name suggests, is used to log runtime statistics from a live BigWorld Server. It does this by periodically querying the BigWorld server machines and processes for a predefined set of statistics and writing the results to a MySQL database. These statistics typically include (but are not limited to):

1. CPU, memory, disk and network usage;
2. Elements of server state, such as current and cumulative number of entities;
3. Profiles of specific code blocks.

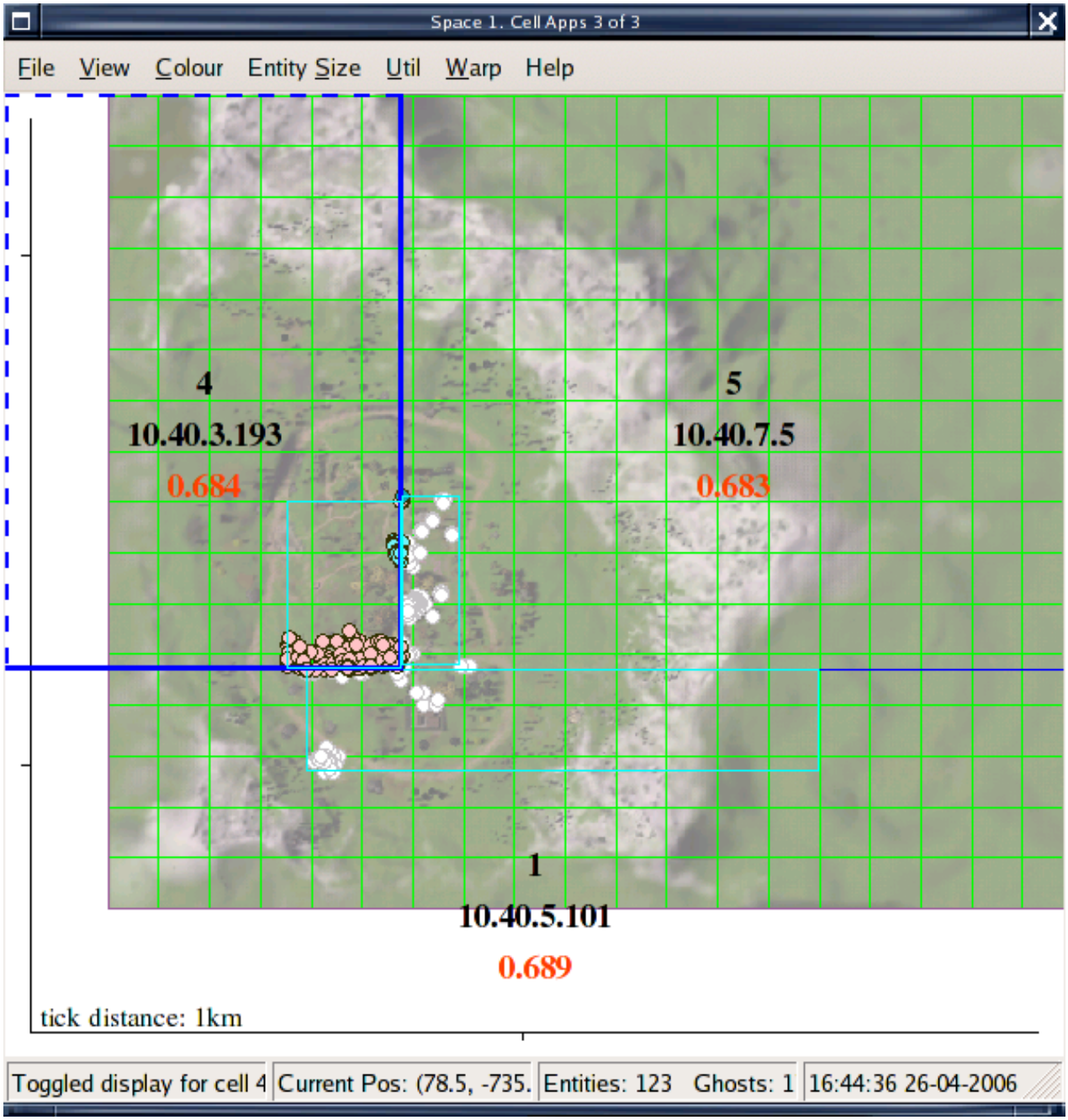
The statistics collected by Stat Logger provide the basis for the bulk of the analysis in this document.

A graphical representation of the data gathered by Stat Logger is available using the Stat Grapher module of WebConsole (see StatGrapher Screenshot on page 20, below). Note: for illustration purposes, there are additional graphs in this document generated using gnuplot. (http://www.gnuplot.info/).

Stat Grapher live monitoring interface

4.2.2. Space Viewer

Space Viewer provides a top-down graphical display of entity and cell distributions across a particular space on a BigWorld Server. It consists of a daemon and a GUI app written in Python. The daemon polls the server for cell data and writes the data to a logfile. The GUI application connects to the daemon process via TCP and requests cell data for any time within the log period.



Space Viewer showing three cells

Space Viewer displays cell boundaries as blue lines, and entities as small circles, coloured by type. The green grid lines indicate the boundaries of the chunks that comprise the 3D world data.

Each cell is labelled with its ID, IP address, and its current load (represented as a float value 0 to 1.).

Space Viewer can also display the rough entity extents for each cell (represented here by the cyan rectangles). This is useful for getting an idea of the distribution of entities in a large multi-cell system. It is not feasible to record and display the position of every individual entity in systems of this size, and as such Space Viewer only supports tracking entities on a single cell at a time. In the example above, Cell 4 is selected and we can see the individual entities within its boundaries (the coloured circles) and the ghost entities near its boundaries (the greyed out circles).

4.2.3. Bots

The technique used to generate server load in our testing procedures is via a set of *"bot"* processes. A single Bot process provides the ability to establish multiple client connections to a server with a low memory / CPU overhead for each connection, as opposed to a GUI client which is generally far more resource intensive. The client connections established via a Bot process are capable of all the same actions as a GUI client and receive the same network traffic/events.

Bots are a valuable load generation technique because they allow the simulation of huge numbers of concurrently connected clients with a relatively small amount of hardware dedicated to the load generation itself, while still providing an accurate and realistic test of the server. Since the server makes no distinction between bots and real (GUI) clients, the load on the server is an accurate reflection of the load generated by an equivalent number of real clients.

Bots should be used by developers to test their games during the development cycle and towards the beta stages. It is a useful tool to evaluate the effect of new features and optimisations by directly comparing load statistics for a given set of entities to a baseline.

4.3. Test Environment

The large scale tests were carried out at the IBM Deep Computing facility in Poughkeepsie, NY. Various logging applications were installed to log the test results. These results were then analysed.

4.4. Hardware Configuration

The results presented as part of this whitepaper were collected on three different clusters. The hardware configurations for each are given below:

4.4.1. IBM Test Cluster 1

96 dual 3.06GHz Xeon machines, each with 3GB RAM and Gigabit Ethernet, running Redhat Enterprise Linux. 16 machines in the cluster were Hyper-Threading (HT) enabled (i.e. had 4 logical CPUs).

Unless otherwise specified, all tests run on this cluster were done with the 16 HT machines allocated to running the bots processes, logging and administration, while the remaining 80 non-HT machines were allocated for running the game server.

4.4.2. IBM Test Cluster 2

128 dual 3.06GHz Xeon machines, each with 3GB RAM and Gigabit Ethernet, running Redhat Enterprise Linux. 22 machines in the cluster were HT-enabled.

Unless otherwise specified, all tests run on this cluster used 20 of the HT machines for bots, one of the HT machines as the control node (where logging and administration was carried out). The remaining HT machine and 106 non-HT machines were used for running the game server.

4.4.3. BigWorld Local Test Cluster

20 machines of varying speed between 1GHz and 3GHz, running various versions of Fedora and Debian. The heterogeneous nature of the cluster, in terms of both hardware and OS, enabled server tests across multiple possible configurations while also verifying that the load-balancing and redundancy algorithms operate effectively even with cluster nodes of varying performance.

4.5. Game Configuration

Each test was run using the following server-side settings:

- Game tick rate = 10Hz (the rate at which server updates are sent to the player);
- Client Area of Interest (AoI) radius = 500m;
- Client downstream bandwidth limit = 20kbit/s;

4.6. Feature Demonstrations

4.6.1. Server Scalability

4.6.1.1. Description

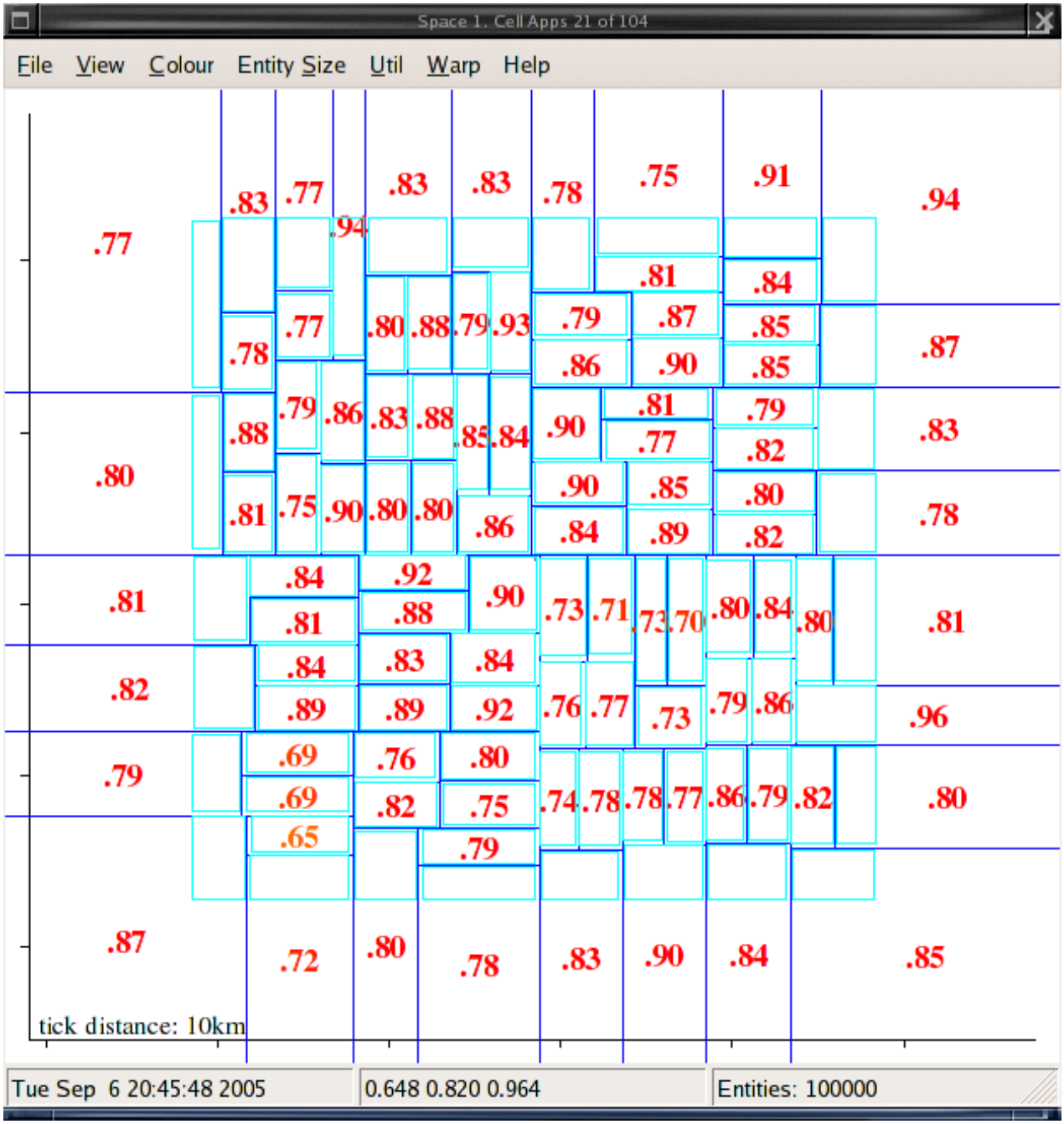
This test aimed to verify the server's ability to handle extreme volumes of concurrent players in a single shard, and to determine the maximum number of clients that could be connected on the available hardware. The test was performed on IBM Test Cluster 1 by adding bots to a 40km x 40km square region of a single space.

It is important to note that the only entities in the game world were client entities; there were no NPCs spawned. As far as server load is concerned, this test should yield overly conservative estimates of the server's ability to handle large volumes of entities. The cost of a client entity on the server is (in general) greater than an NPC entity because updates of entities around a client entity must be sent down to the client, whereas this is not true of NPCs. For example, using the FantasyDemo world that is bundled with the BigWorld server, the cost ratio of a client entity to a Guard NPC (who is constantly patrolling and watching for attacks on other Guards) is roughly 4:1. Guards are considered to be *heavy-weight* NPCs, as they are constantly pathing and run AI routines. Examples of *light-weight* NPCs would be shop-keepers, mission-givers, and other NPCs who don't move much and mostly react to player actions as opposed to acting of their own volition. We would expect these *dumb* NPCs to cost 1/20th of a client entity or even less.

In a typical game world containing both client entities and NPCs, the majority of CPU time is consumed in the formation and sending of downstream packets for client entities; therefore, as client entities are empirically more expensive than NPCs, and provided that the NPC AI isn't excessive, it is expected that there can be many NPCs for the cost of one client entity in a typical game.

4.6.1.2. Results and Analysis

The IBM Test Cluster 1, was able to achieve a total of 100,000 concurrently connected clients. The below space viewer diagram shows the distribution of the space into 80 cellapps which were used to handle the clients load.



Space Viewer showing 100,000 concurrent users

These results show that the BigWorld Server can handle 100,000 CCUs with automatic distribution of load between multiple cellapps. At the time, this was the largest number of players in a single world.

The entity bounds shown above indicate that the clients are spread over a 40km x 40km region, or 1,600km^2. Each entity has an Area of Interest (AoI) radius of 500m, which means each is aware of entities in the square kilometre surrounding it. For 100,000 entities spread over 1,600km^2, this equates to about 62 entities in the AoI of each client, which is a reasonable number of entities for each player to be aware of for a standard World of Warcraft style MMO.

4.6.2. Load Balancing

4.6.2.1. Description

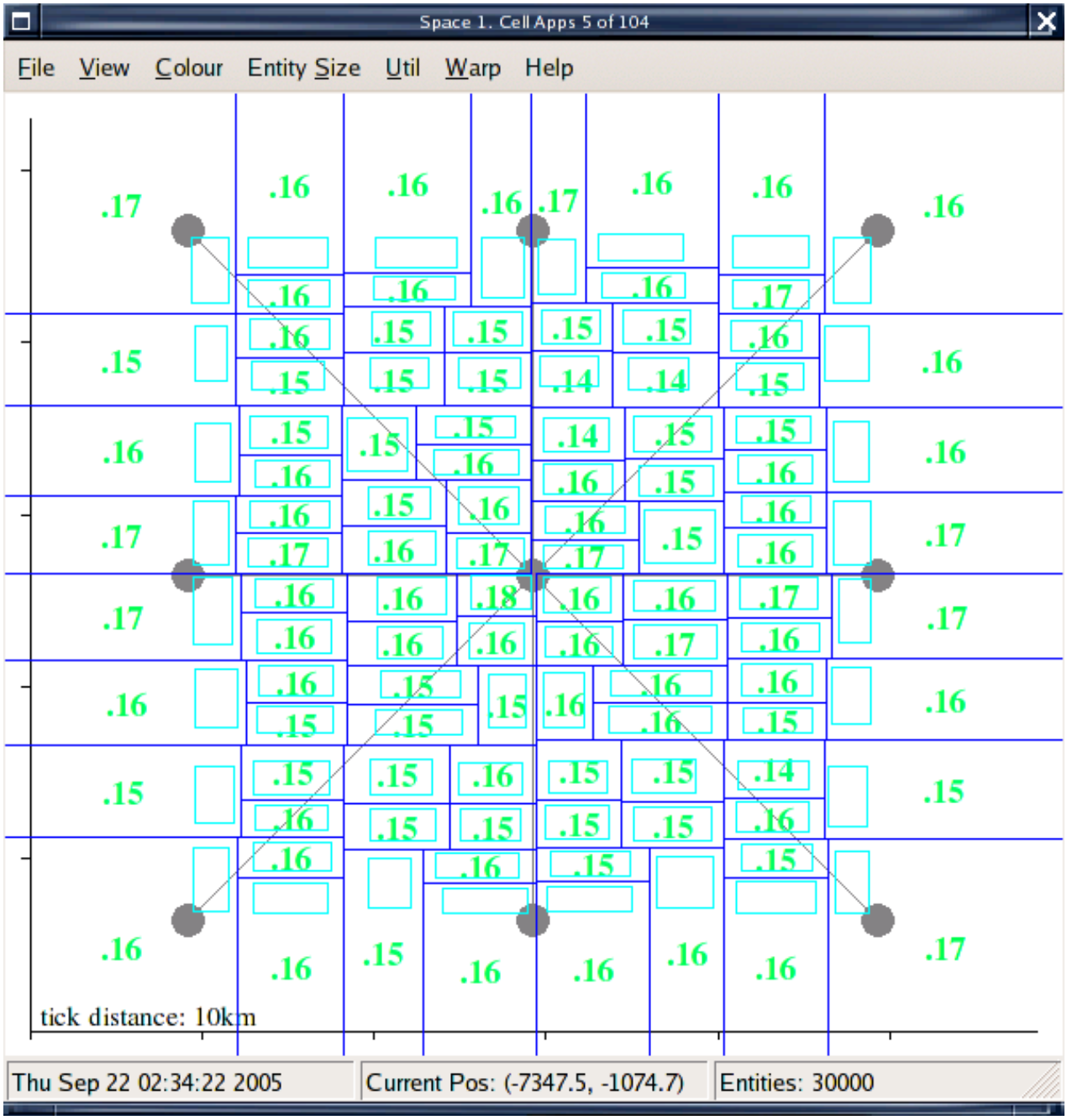
The BigWorld Server is designed to distribute load over multiple machines and to do so in a timely manner, where the desired average load level is specified by the server administrator. This also means that administrators can add or remove machines from a BigWorld cluster at will and the server will always make the best use of the resources available to it.

This demonstration aims to show that the server responds to changes in entity distribution in a timely manner and maintains average load as close as possible to the desired level, with as little deviation as possible, whilst using as few machines as possible.

Multiple tests were run with different entity distribution to show that the load is distributed in a balanced way between the different CellApps. Some of these tests are specified below.

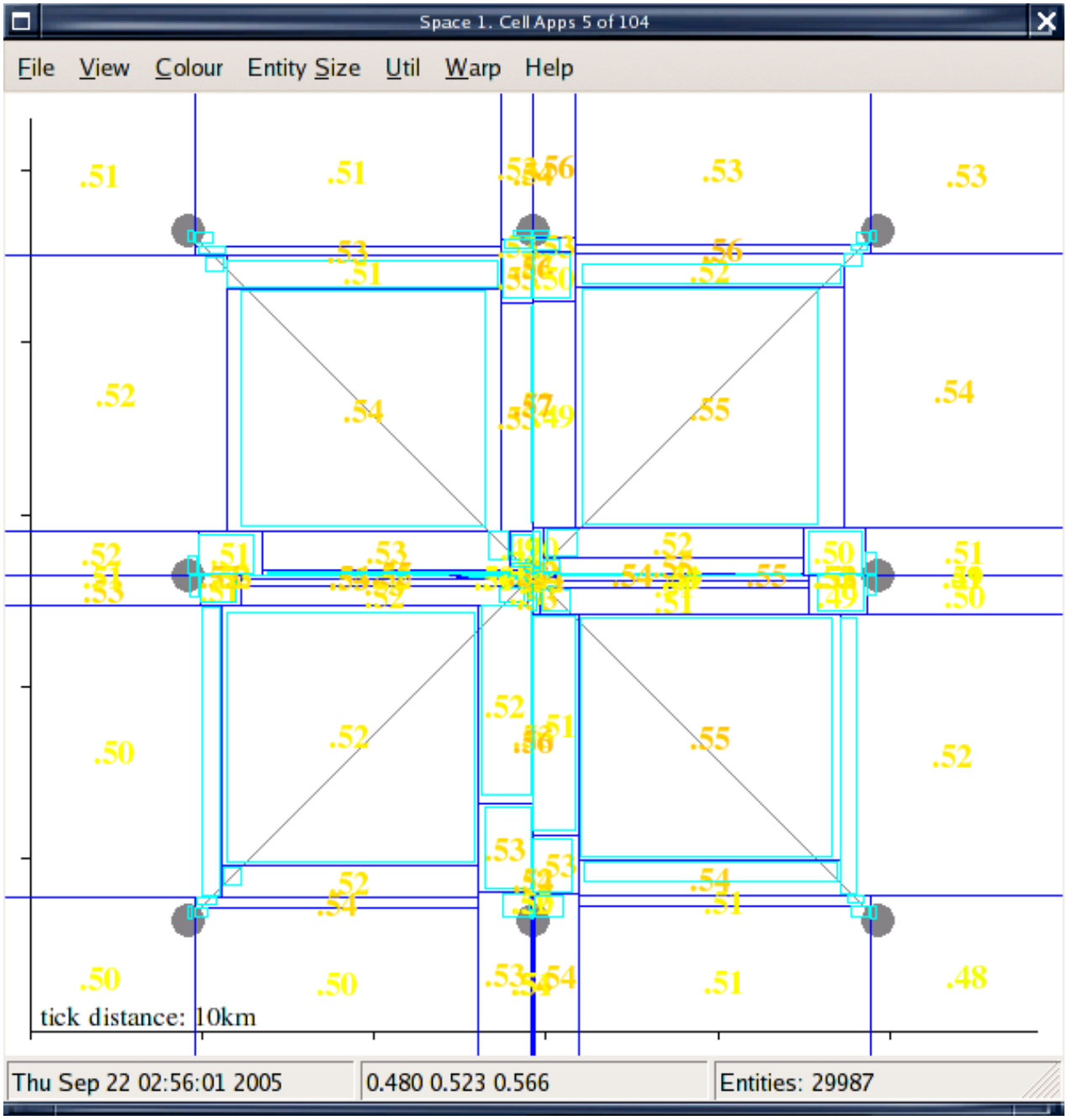
4.6.2.2. Shifting Movement Patterns

These are the results from the 8-pointed-star test. Initially, we have 30,000 clients distributed evenly across a 40km x 40km space, we can see that the space is distributed uniformly into cells:



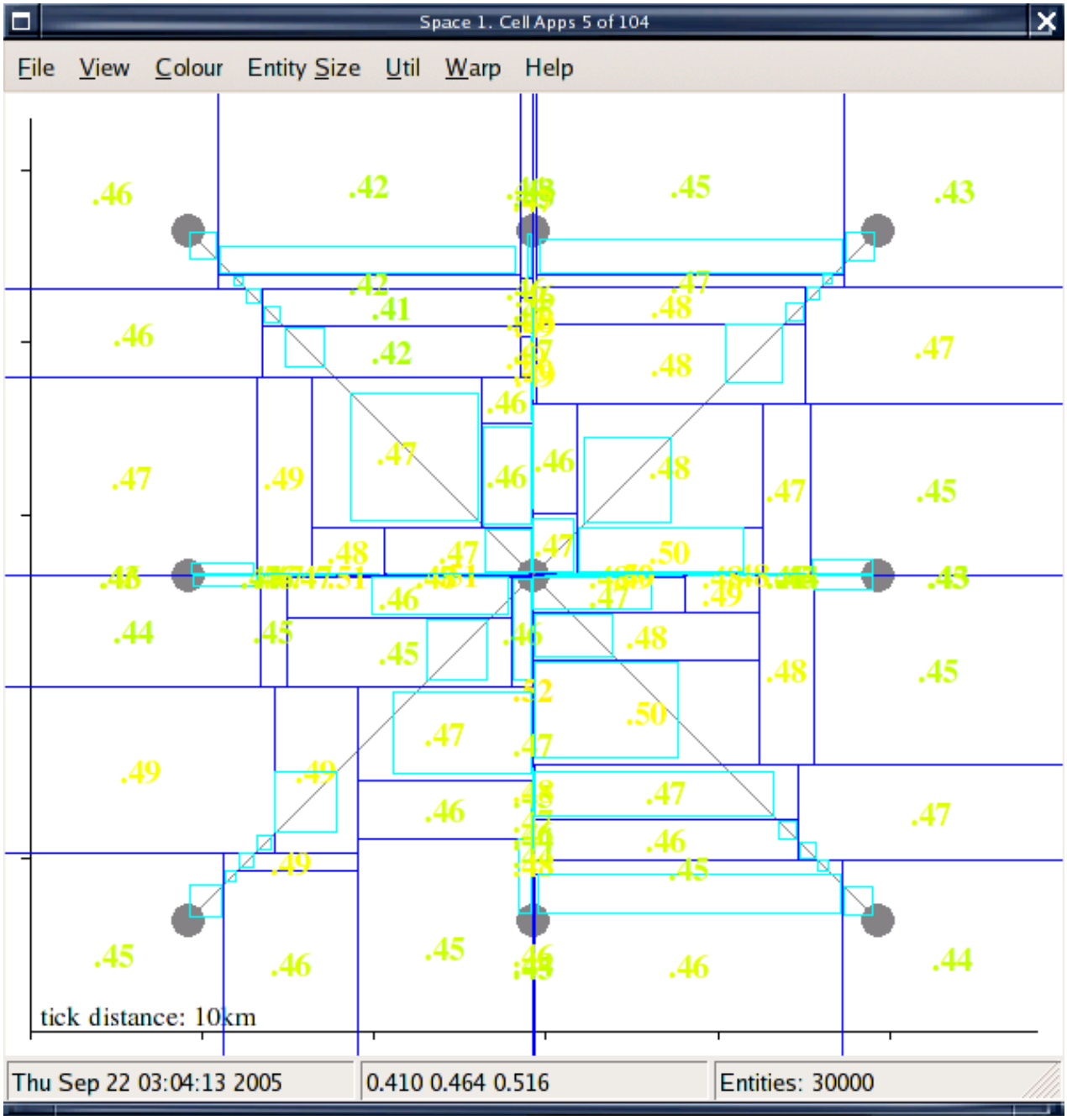
30000 entities distributed evenly across a 40km x 40km region

The clients begin walking to one of the points of the star, and the load increases as the clients begin bunching up around the points of the star and its centre. The important thing to notice here is that even with a shifting entity distribution (and cell layout), the minimum and maximum loads differ from the average load by less than 5%:



The 30000 entities are now concentrated at nine locations

As the clients begin walking on the paths between the points of the star and its centre, the density decreases and the distribution shifts back towards the centre, however the load still stays balanced to within ±5% of average load:



Clients begin traversing the paths from the points to the centre

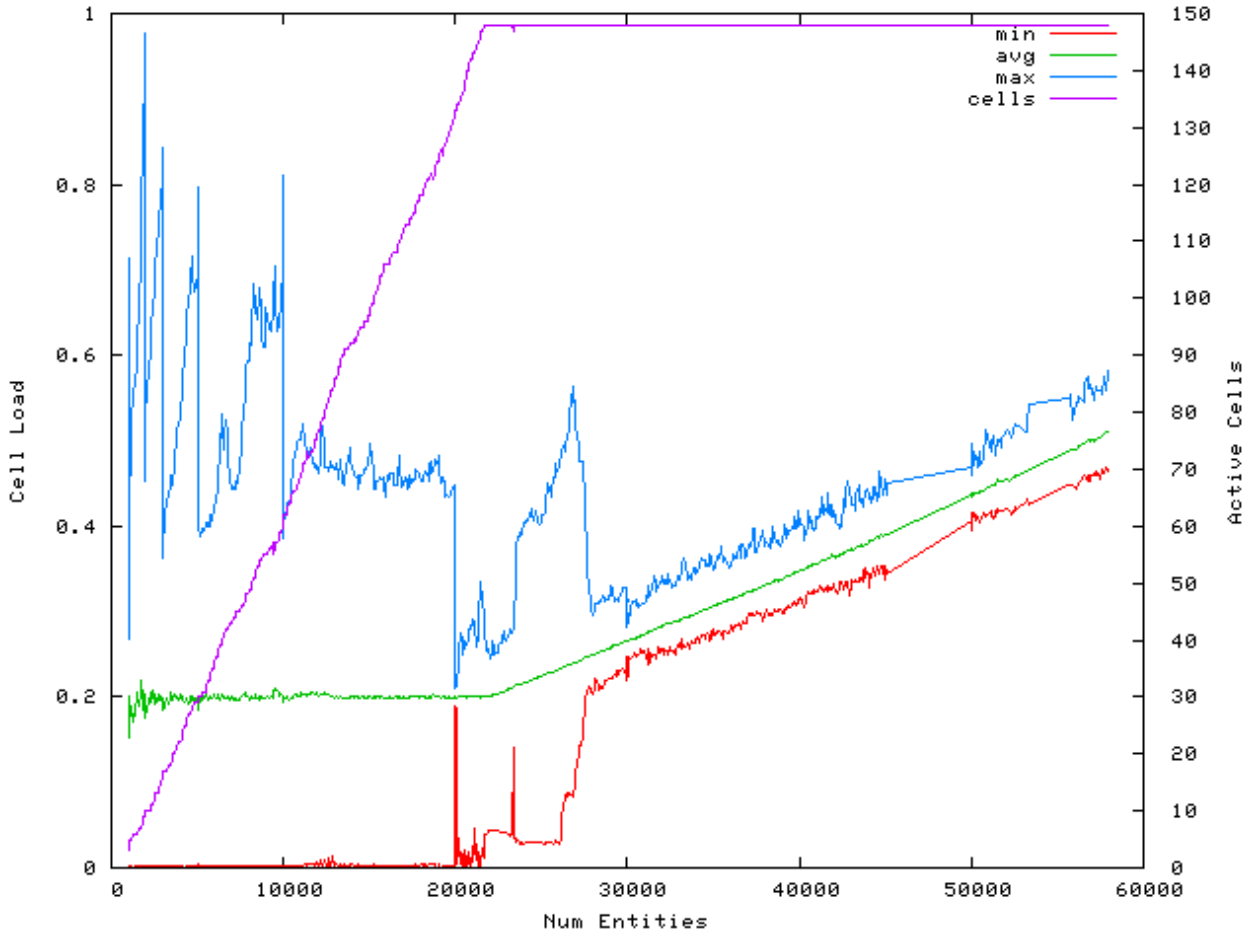
These results show the server automatically balances the load between multiple CellApps to achieve a load-balanced state that was consistent across the CellApp processes, reflecting optimal resource utilisation. Tests with other movement patterns produced similar results.

4.6.2.3. Economical Allocation of Machines

In this test we have set the server desired average load to 20% while increasing the amount of entities (see the horizontal axis). As the entities numbers increase the server automatically used more cells (purple line) and kept the average load (green line) on each cell constant. After depleting the available CellApps, the server started to use more than 20% CPU on average per cell. Please note that this demonstration uses similar raw data as the scalability demonstration below.

Customers can similarly set their desired server load and the BigWorld Server will automatically add more cells as the number of entities grows until the number of CellApps is exhausted. After this point, load increases linearly with respect to the number of entities in the system.

The graph below shows the average cell load (green), maximum cell load (blue) minimum cell load (red) and total number of cells (purple).



Graph of server load and active Cells as entities are added to a BigWorld Server

4.6.3. Scalability Demonstration

4.6.3.1. Description

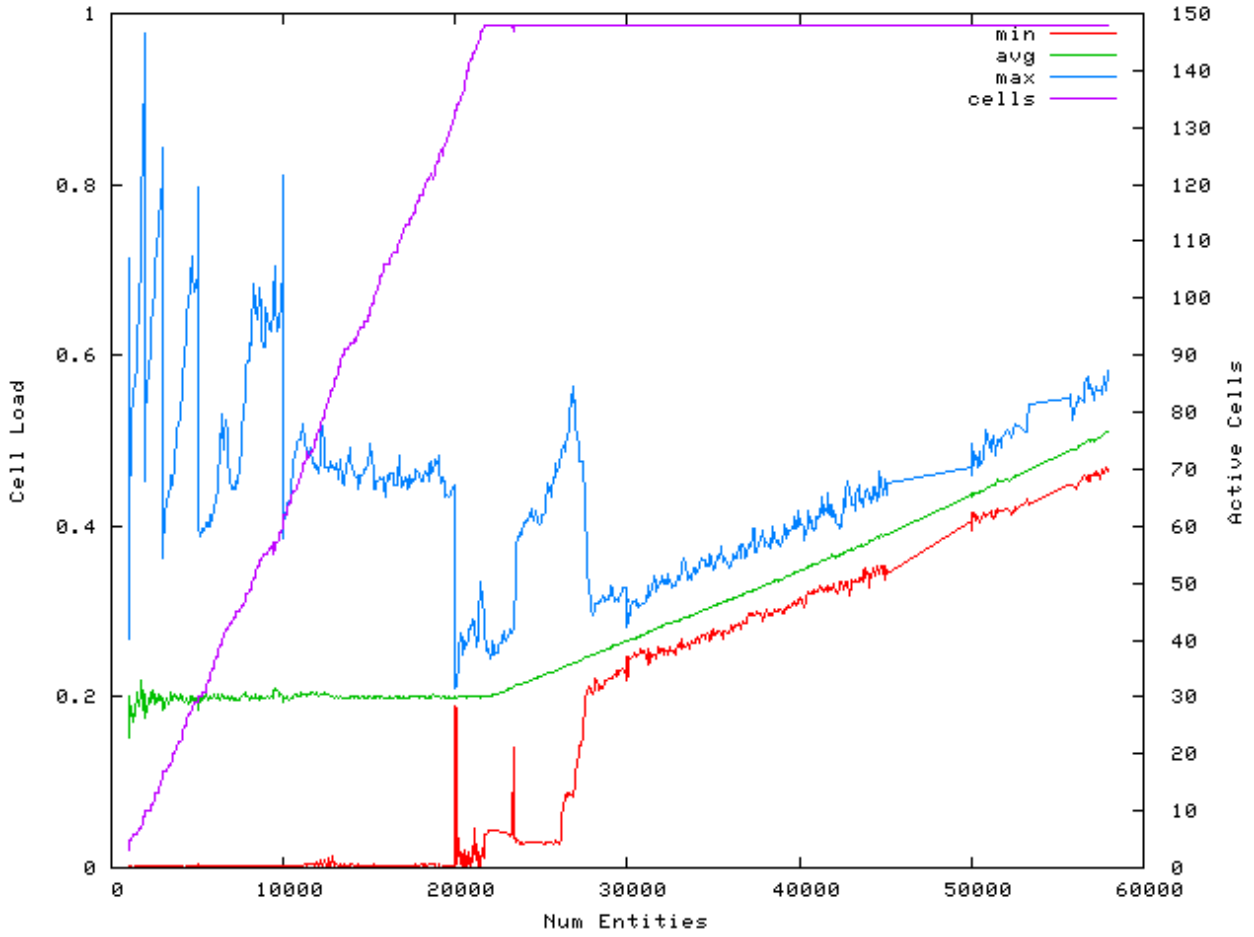
Multiple tests were run to examine the effects of altering different server parameters, but for the purposes of this demonstration they were functionally very similar. Each test consisted of adding moving bots maintaining constant density while making sure that all the downstream bandwidth to each client was being utilised. This allows us to show that the BigWorld Server does indeed scale linearly with respect to number of entities with other variables (density, AI etc.) held constant.

4.6.3.2. Results and Analysis

The graph below relates server load and number of active CellApps to the number of entities. The important thing to notice in each graph is that load grows linearly with the number of entities. The desired cell load was set to 20%. The results show that:

1. Average load (green) held constant while number of active cells (purple) increases linearly, then

2. Number of active cells held constant (i.e. all CellApps are now active) while average load increases linearly.



Graph of cell load and active cells vs. number of entities for Test 2

These results demonstrate that the BigWorld Server does indeed scale linearly and therefore any BigWorld Technology based game would be free to expand as much as its customer base demanded.

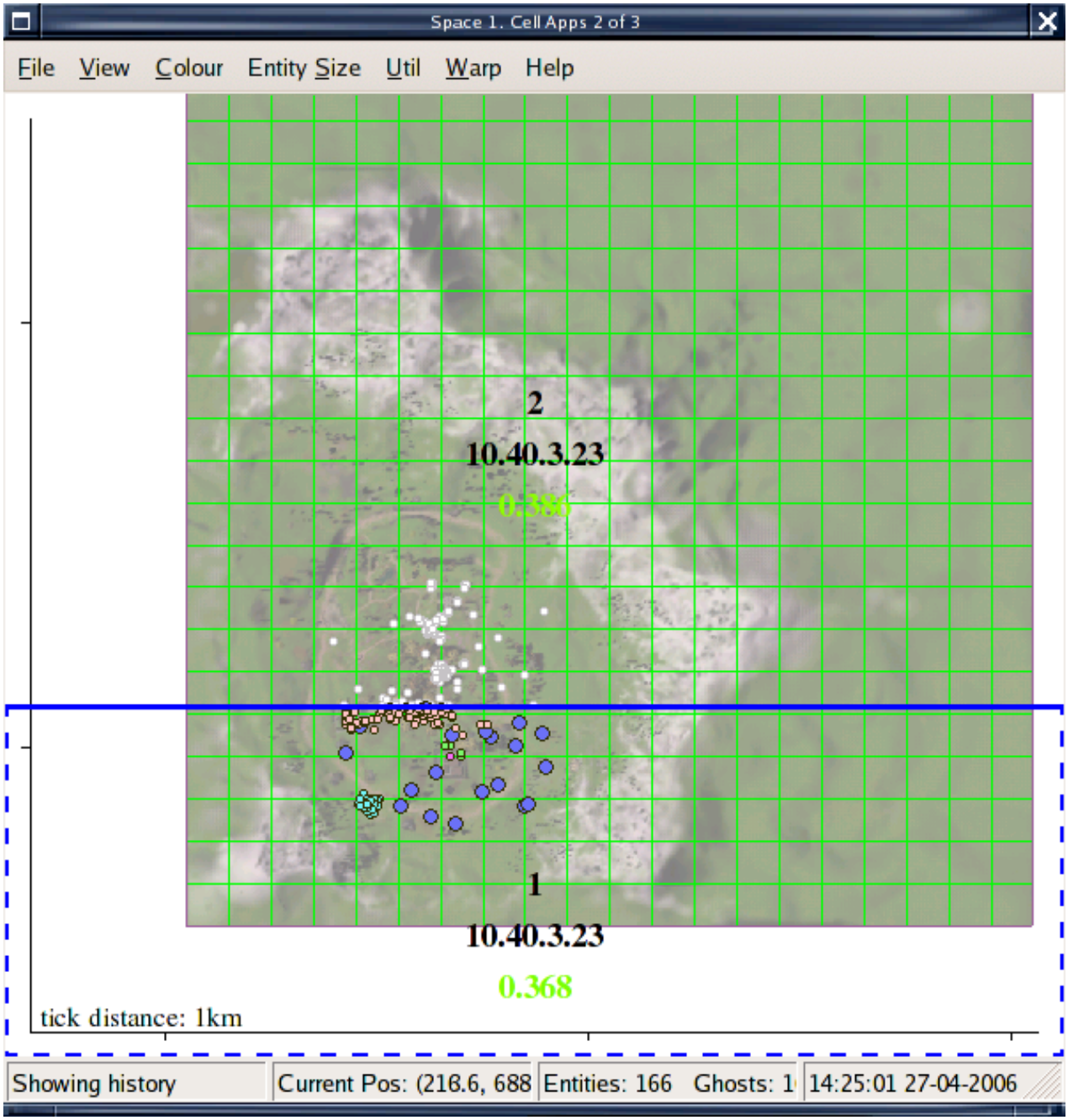
4.6.4. Fault Tolerance

4.6.4.1. Description

A demonstration of the BigWorld Server fault tolerance capabilities was performed by killing a CellApp process on a running server and observing that all the entities on the Cell owned by the killed CellApp are restored by BaseApps to another available Cell. This test does not depend on the size of the cluster as it deals with single component failure and can be also demonstrated by your account manager if required.

4.6.4.2. Results and Analysis

The server used in this test consisted of 3 CellApps. Initially only 2 CellApps are active. This test uses the standard FantasyDemo game that is shipped with BigWorld Technology, with 32 bot clients and one real game client connected.



Server state before the CellApp is killed

The following figure shows the state immediately after Cell 2 is manually hard-killed:

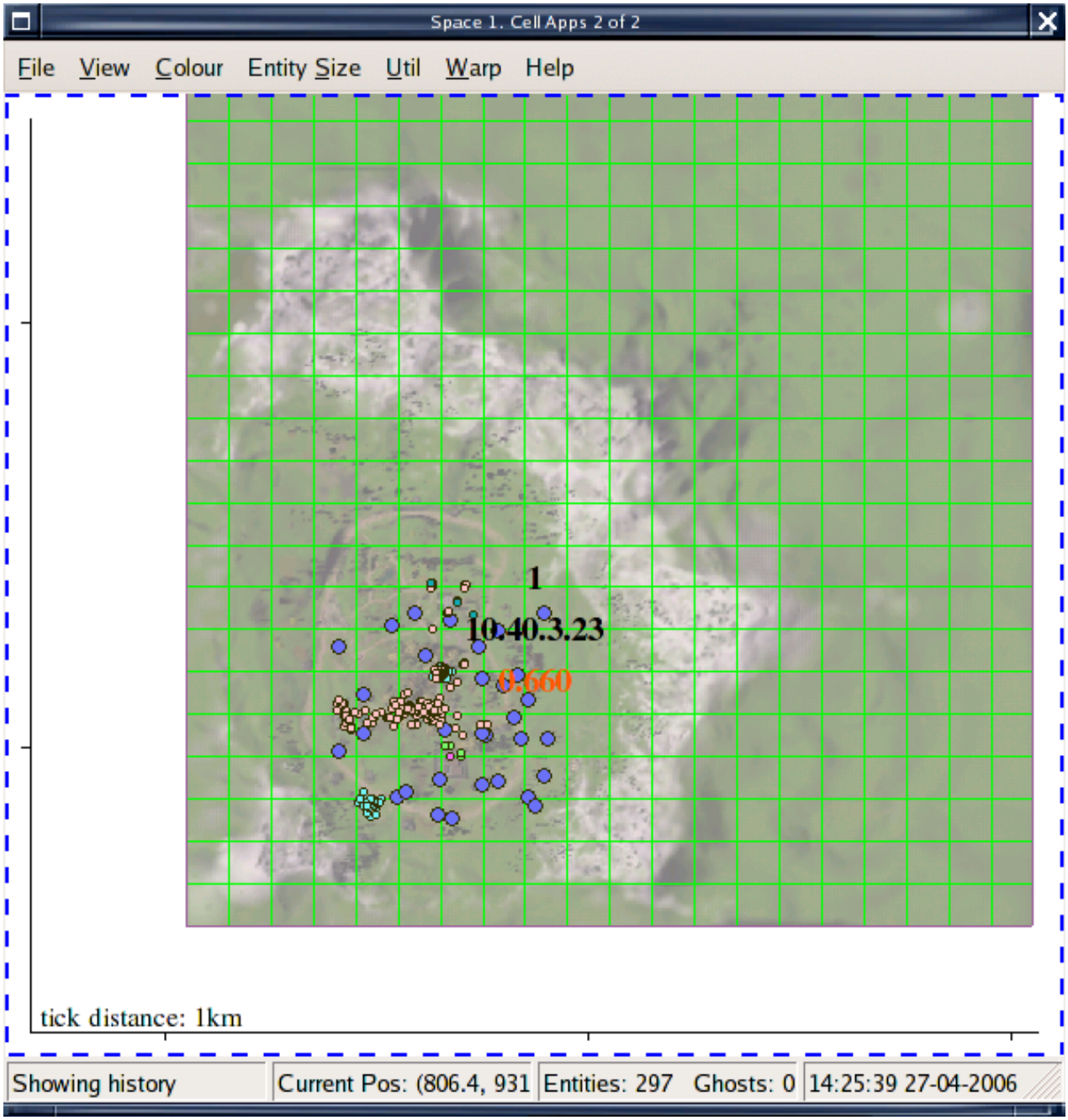


Figure 14. Server state immediately after Cell 2 is killed

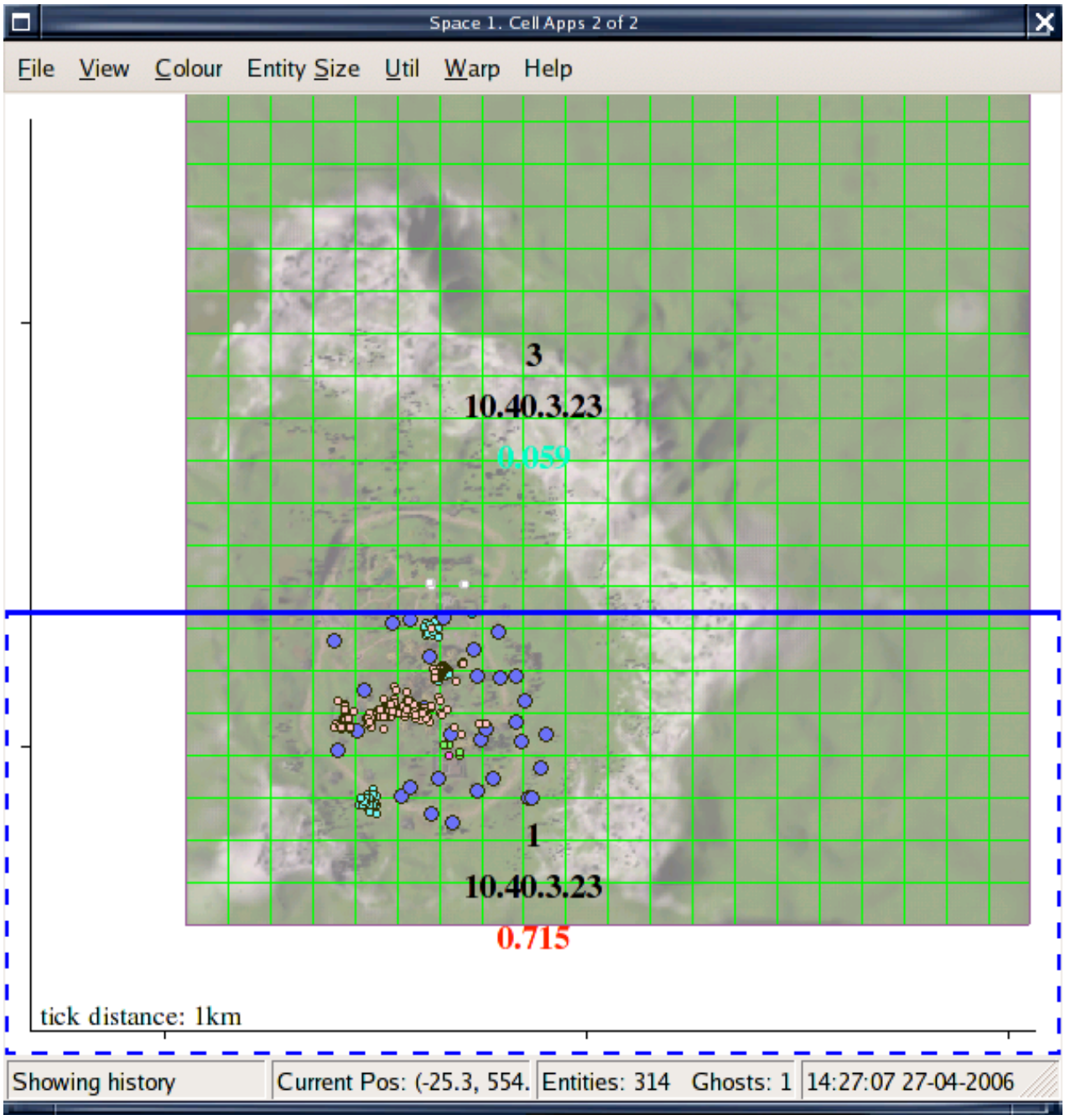
As soon as CellApp 2 is killed, the BaseApp restores that Cell's entities to the next available Cell on that space, which in this case is Cell 1. On the client, we observe a short (~2 seconds) pause in entity updates (i.e. guards and wildlife stop moving around) while the restore takes place.

Naturally, the load on the first Cell increases quickly with the extra entity load, which causes two things to happen:

1. The CellAppMgr notices that the load on Cell 1 now exceeds the 0.5 desired level and therefore adds the CellApp that was in reserve to the space.

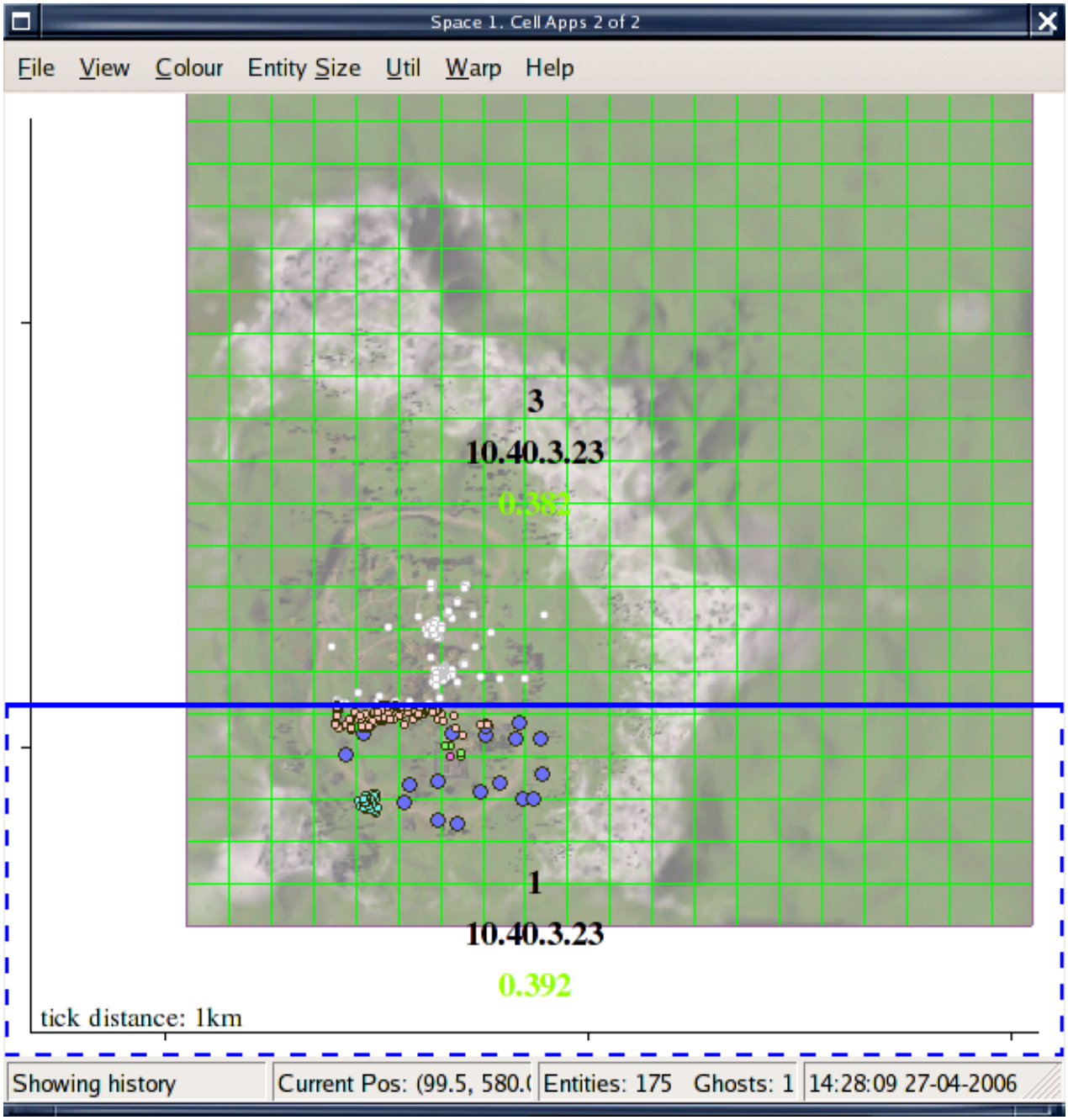
2. While temporarily overloaded, Cell 1 *scales back*, which means it temporarily throttles its downstream bit-rate to clients to reduce load while it waits for another Cell to start taking on its excess load to relieve

it. This means that clients will temporarily receive updates less frequently per-entity, but the server state is kept stable.



The unused cell is added to the space to replace the dead one

As the new Cell takes over a larger and larger portion of the space, the load on the first CellApp moves toward the lower equilibrium state where both cells have equal load:



Desired load level is restored

The above example is a typical example of how first level fault tolerance works. There is little disruption to the flow of game play on the client (it feels like a short lag spike) and the game state is preserved accurately. We have illustrated that in the event of a Cell failure, neighbouring Cells assume responsibility over the space and containing entities that previously resided on the failed Cell. After a short time, the average load of each Cell increases to a level that is inversely proportional to the number of available Cells allocated to that space. Therefore, having more available CellApps improves the fault tolerance by spreading failed Cells' load over more available Cells. This reduces the duration of scaleback observed by players.

Chapter 5. Summary

This document covers multiple aspects of the BigWorld Server. It reviews the server architecture, it discusses common customers deployments and explains the pros and cons of each deployment and it reviews scientific testing done on the BigWorld Server. Evaluators and customers can use this document to plan their future deployment and to make sure that the BigWorld Server will scale based on their needs.