Season: A Dynamic Load Balancer for Virtual Environments

Fakultät IV - Elektrotechnik und Informatik
Intelligent Networks / Intelligente Netze (INET)
Research Group of Prof. Anja Feldmann, Ph.D.

Tobias Jacobowitz
Sebastian Garn
Thomas Krenc

November 15, 2011

Prüfer: Prof. Anja Feldmann, Ph. D.
Supervisor: Ingmar Poese, Benjamin Frank
and Georgios Smaragdakis
Abstract

Currently deployed client-server architectures for Massively Multiplayer Online Games (MMOG) serve millions of concurrent clients. Scaling MMOGs to this magnitude is challenging. Several techniques have been proposed in order to distribute client’s load on multiple servers, e.g. spatial decomposition of the virtual world, or ‘sharding’, to name some. Most of the proposed solutions rely on static assignments of users to servers in the virtual world, thus, they can not cope with the volatile behavior of clients. As a consequence, some servers are overloaded, while others are underloaded. For MMOGs, computation power is a scarce resource which utilization has to be optimized in order to offer better quality of experience to the end users as well as improve the reliability of the system.

In this work, we argue that it is possible to dynamically assign users to servers in order to equalize the load of the servers allocated in a MMOG. We propose a distributed load-balancing algorithm that dynamically assign clients to servers based on the current activity in the system. Our distributed algorithm also allows an MMOG to re-assign the area of responsibility between servers in a distributed, scalable, and robust way. To evaluate the performance of our algorithm we build a prototype, called Season, that can serve as a building block for a MMOG. Our results, with real and synthetic data, show that the load among servers in an MMOG is well balanced when Season is used.
Contents

1 Introduction 1
   1.1 State of the Art ............................................... 2
   1.2 Related Work .................................................. 3

2 Basic Concepts 5
   2.1 Extended Client-Server Architecture ......................... 5
   2.2 Spatial Decomposition ........................................ 7
      2.2.1 Voronoi Diagram ........................................... 7
      2.2.2 Delaunay Triangulation ................................... 8
   2.3 Load Balancing by Movement Mechanism ....................... 9
      2.3.1 Load Based Forces ......................................... 11
      2.3.2 Spring Based Forces ....................................... 12
      2.3.3 Weighted Nodes ............................................ 14

3 Simulator 15
   3.1 Identified Issues - Chessboard Problem ..................... 16

4 Implementation Details 17
   4.1 Sequence of Processes ......................................... 17
   4.2 Client Bootstrapping .......................................... 22
   4.3 Robustness .................................................... 22

5 Experimental Setup 23

6 Evaluation 26
   6.1 Feature Demo .................................................. 26
   6.2 Quake3 Scenario ................................................. 28
   6.3 Other Experiments .............................................. 32
   6.4 Conclusion ..................................................... 35

7 Future Work 36

8 Conclusion 38
1 Introduction

In recent years networked virtual environment applications, especially massively multiplayer online games (MMOG) have become widely used [6]. For example, World of Warcraft (WoW) alone has twelve million active subscriptions. At peak times more than two million concurrent users are logged in at the same time. While other games from that genre are not as successful as WoW, they still reach numbers of subscriptions in the order of millions [1]. In order to satisfy this demand in MMO Games, scalable high performance systems are needed. Thus, the number of subscriptions only hints the capacities and scalability of the architectures the games are running on.

Today’s systems employ a client-server architecture to scale the number of concurrent clients. In traditional client-server architectures, the server is mainly accountable for enabling interaction among clients, and for handling any dynamic content of the virtual world, e.g. objects, movements, in order to keep the world consistent for the clients. Since the load increases with each client participating in a virtual environment, a single machine is not able to handle load to the extent of current MMO games. That is why multiple servers, and more important, a way to partition the load over these servers is required.

One common technique to make use of multiple servers is to decompose the virtual world into parts to which each server can be mapped [2]. Thereby, a client is assigned to a server that is responsible for the part of the virtual world the client is located in. Thus, client interaction is localized and the communication among servers is kept small. In Second Life, the world is divided into 256 x 256 meter tiles, of which each is statically bound to a machine [3] [10] [11]. Any player and its objects are solely available on the tile they are located at a point of time. WoW on the other hand follows a more sophisticated alternative: The virtual world is split into regions, according to the geographic properties of the virtual world itself and with regard to predictions of where clients may be located most often, e.g. cities. However, this technique poses a great challenge to developers, as they need to accurately predict where crowds of clients may evolve for an indefinite period of time. Any miscalculation may lead to a longer maintenance work, which of course consumes money. Furthermore, the parts are statically assigned to the servers, thus if the hotspots become dynamic, i.e. crowds may emerge spontaneously, or move over the virtual world, parts with less computational power become overloaded and the system fails. Such fails occurred also several times in the past for the MMO game Eve Online, as a result players in this game have to announce (planned) big battles to the developers first [9].

Another technique used by WoW is called sharding, in order to serve more clients. A shard is a copy of a part of the virtual world. With each shard the capacity of a part of the virtual world is increased. Also, by creating multiple copies of the server that is responsible for a quest, different clients may explore the original state of this quest to make individual experiences. However, since each shard has its own dedicated server, sharding segments the virtual world, which prevents clients from interacting with clients from another shard. This may have a negative impact on the quality of experience. Eventually, each client may choose the shard it wants to participate in. This may lead to
overloaded shards, while others are less overloaded. In the worst case a shard becomes over utilized, and further clients are not able to join the shard or they need to accept a bad game quality. Thus, shards do only scale in terms of serving more clients concurrently, however at the expense of being isolated from other shards.

In this work we propose a technique that is able to load-balance any number of servers, by dynamically changing the size of the regions the servers are responsible for. Thus not only the number of concurrent clients is scaled, but also the size of the virtual world, whereby no segmentation of the virtual world is used. Furthermore, we design, implement and analyze a server application that incorporates this technique and is able to communicate with other instances of the server application in a distributed manner. We make use of a virtualized test bed for deploying the servers in a virtual, but networked environment, which are stressed by hundreds of simulated clients providing realistic user movements, including hot-spots. Finally, we perform experiments and analyze different aspects of the application, and show how client-server architectures can benefit from load-balancing a network in a decentralized fashion.

1.1 State of the Art

Since the rapid growth of MMOGs, research is done to circumvent the scalability and load balancing issues of the traditional game server concept. This section begins with a description of the general approach most of the paper in this research area have in common. Afterwards selected work gets discussed in Section 1.2, which is especially related to the context of this report.

MMOG server architectures can be roughly divided in central and decentral. The central systems employ a central instance coordinating the game servers, which are doing the actual calculations for the virtual environment. The basic advantages and disadvantages of those systems are mostly valid for all discussions of central vs. decentral. The central systems often allow easier load balancing because of global state and coordination. At the same time the central entities may limit the scalability of the system and result in a single point of failure, as central management servers may get overloaded themselves or may suffer from system failures and thereby pose a threat to the whole architecture. Most of decentral proposals for respective architectures for MMOG servers are P2P concepts. The most extreme proposal for pure decentral P2P systems is to give up any kind of server entities and let the players become the servers as well. In centralized systems exist entities which may coordinate the client to server assignment and the server to server communication. Because of the absence of those management entities, all of these pure decentral systems need some kind of self organization and state exchange mechanisms. Thus descriptions of these organizational parts are the main focus on game server papers based on a decentral structure.

Besides the decision for a central or decentral architecture, every concept has to provide an algorithm for load distribution: Which host in the network is responsible for which object? A majority uses spatial decomposition of the virtual environment for this purpose. The game world is divided in multiple regions and each of those regions gets a host assigned, which is now responsible for it. The properties of those regions are
strongly differing in actual papers in terms of the geometrical shape, number and dynamics (reallocation during runtime). Spatial decomposition is mostly used to achieve a well balanced client partition among all servers. This is done based on the assumption that a server’s load is mainly based on the number of its clients. This seems to be a good approach for load abstraction as shown by Ye et al. in [15]. Although it is not always a very accurate way as described by Bezerrra et al. in [16]. For the reason of simplicity, this report always refers to load as the number of clients.

1.2 Related Work

A promising example for the centralized approach is the paper of Quax et al. [12]. The paper provides a complete architecture and classifies the servers into different types. The most important are proxy, logic, and region management server. The game world is spatially divided by the region management server according to the actual load. All regions are rectangles and may be cut in half or doubled in size if necessary. Each region gets one responsible logic server assigned. The region allocation can be dynamically changed during runtime by splitting or aggregating regions. Clients do not connect directly to the logic server but to the proxies, which are also acting as caching and broadcasting entities. Normally a server has to maintain connections to all clients, sending the same information multiple times to all clients. This creates unnecessary network based load on the game server. In opposite to this, all clients in Quax’s concept are only connected to the proxies, which may act similar as web proxies. Thus they ease the load on the logic server and handle all necessary connections, based on the client’s area of interest (AoI). Also, the proxies ensure that the region management mechanism is transparent to the clients. The main drawback in this architecture is the central region management system. This system itself generates new problems regarding load balancing and scalability. Furthermore the architecture is relatively complex with five different server roles (authentication and databases not discussed here). An advantage of the available global knowledge at the management system is that it should be theoretically possible to achieve nearly optimal load balancing for the logic servers. On the whole, the paper from Quax et al. [12] describes a more complete architecture, but in a more general manner than this report. Concepts as the proxy server may also be applied to the architecture describe here.

An opposite approach take Hu and Liao in their paper VON with proposing a pure P2P based concept [4]. The game world also gets spatially divided into small regions. The amount of regions is the number of players in the network. The shape of those regions is not a rectangle as in most concepts, but a Voronoi cell (see Section 2.2.1). Thereby each event or object is handled by the player, which is the spatially closest. As a result this concept completely gets rid of servers and is thereby fully decentral and has good scalability properties. To overcome reliability and limited knowledge problems information is saved redundantly and players are also connected with more players than their direct neighbors. Naturally every player, which is also a server as well, is moving through the game world and is thereby moving the built Voronoi diagram as a whole. The VON concept makes no special use of this property, but it is fundamental different
to most other concepts where servers have fixed or no positions in the virtual game world. A prototype implementation is also available for public use. Basically the VON concept is very promising, i.e. it provides excellent scalability and removes need for additional server infrastructure. However, in real world applications some serious problems exist, which limit the usability of the VON concept: Players in a MMOG network are very unpredictable, there are no guarantees for their uptime and they are highly untrustworthy.
2 Basic Concepts

![Figure 1: extended client-server architecture](image)

2.1 Extended Client-Server Architecture

Typical client-server architectures in massive computer gaming are simple. Servers are assigned to regions in the game world, and thus to the clients in a static manner. However, the decomposition of the game world into regions is not an easy task. If the design does not match the predicted client behavior, a server may become overloaded when too many clients are located in the same region. Since a server is not able to dynamically adapt the borders of its region to the dynamic movement of clients there is always a possibility of servers becoming overloaded, which may lead the system to fail. Furthermore, when the majority of clients move to a single place, servers covering different places are idle and thus waste resources.

In this section a concept is introduced to load-balance a global server network. Thereby the number of clients connected to an overloaded server is reduced, and reassigned to less loaded servers such that each server has roughly the same amount of load to handle. This happens by continuously decomposing the game world, and thus reassigning clients to servers, without having parts of the game world not assigned to a server. That means that every time a region’s space is reduced, other servers will assume the remaining space, keeping the game world consistent, and the number of clients per server globally balanced. Furthermore, each server is communicating to those servers, that are responsible for neighboring regions in the game world, in a P2P-like manner. Thus, the server-server communication is kept local, which significantly scales the number of servers, and, by implication, the number of concurrent clients, too.

*Figure 1* shows the extended client-server architecture. Additionally to the traditional components, namely clients and servers, which are depicted in the *Client* and *Server*...
layer, two additional components are added lying in between, called Control, and Communication layer. In the following each layer is described in more detail:

- **Client**
  The Client layer aggregates any client that participates in the game world. In order to be able to assign a client to a server, the client’s position, which represents the real coordinates in the game world, is projected to a simplified 2D Euclidean space.

- **Control**
  In this layer, an abstraction of the servers, called nodes, are arranged in the same Euclidean space as the clients from the Client layer, while the nodes are logically interconnected using the Delaunay triangulation. The resulting decomposition into regions partitions the clients among the nodes. Thereby a client is assigned to the spatially closest node. See Section 2.2 for a detailed description of the spatial decomposition. This is a continuous procedure, so every time the clients change their behavior, and thus the occupancy of the Euclidean space, the nodes adapt, keeping the servers balanced. This adaption is called node movement and is controlled by each neighborship relation between the nodes, resulting from the Delaunay triangulation, and is defined by three principles:
  - The higher the load of a neighbor is, the higher a node is attracted by this neighbor. If the load of the adjacent node is zero, there is no attraction.
  - The speed a node can move is based on its load such that fully loaded nodes do not move at all, while not loaded nodes move with full speed.
  - Neighborship relations between the nodes are represented by physical springs, which force is calculated according to the current distance between two neighboring nodes. This force determines how much a node is attracted (positive force) or repelled (negative force) by its neighbor in the 2D Euclidean space. If the force is zero, the node movement is not influenced by the distance.

In Section 2.3 the movement mechanisms are explained in more detail. Due to these movement mechanisms the decomposition of the game world is dynamic. However, in order to keep the dynamics, the nodes are able to change neighborships, so they can move flexibly enough without losing the consistency of the game world. On the global scale, the nodes will continuously try to gently reach a relaxed state, which is equivalent to a load-balanced network.

- **Communication**
  The Communication layer adds network connectivity to the nodes, by providing an abstraction, that is a P2P-like overlay, of the Server layer. Every logical connection at this layer represents a real connection at the Server layer. Connections are established and cut here, while the connectivity is calculated at the Control layer using the Delaunay triangulation. So every node positioning in the Control layer has direct influences at the connectivity of this layer, and indirectly at the connectivity
of the Server layer. Furthermore, since there is no central entity provided for this architecture, the communication between the nodes is distributed. Each node does its distinct calculations of the spatial decomposition or the movements, based on a local view. However, the nodes have no synchronization mechanisms, which may often lead two nodes to calculate different views on the node network. Thereby, a node may be determined to cut a connection to its neighbor, while the neighbor does not agree. In order not to compromise the consistency of the node network, this layer implements mechanisms that can handle inconsistent nodes, and provides repair mechanisms, in case the network becomes inconsistent. See Section 4 for a more detailed description of the implementation of the distributed communication.

- **Server**

  This layer comprises all physical servers that are physically interconnected, and are responsible for serving the game world, while the upper layers calculate which server is responsible for which client. Section 5 describes the experimental setup in detail.

2.2 Spatial Decomposition

As already described above the virtual world is spatially decomposed in the Control layer to assign clients to different nodes in a load balancing fashion. The result of this decomposition is a Voronoi diagram, where all clients are assigned to the nearest node in the 2D Euclidean space. This Voronoi diagram is actually only a model for the decomposition process. To build up this logical diagram, a Delaunay triangulation is used, which is a graph theoretical method to arrange the nodes in a triangulation which has the similar properties as Voronoi diagram, but is easier to calculate.

2.2.1 Voronoi Diagram

A Voronoi diagram is a decomposition of a 2D plane containing a set of points (or nodes in our case) into different cells. Each Voronoi cell contains all points where its node is the nearest node in the plane. Figure 2 shows a simple Voronoi diagram with four nodes, five clients and the resulting assignments. With the use of the Voronoi diagram as the client to node assignment model, there is no need for global knowledge. Every node has only to maintain information about itself and its neighbors. This is sufficient for building local Voronoi diagrams and doing the client handover to neighboring Voronoi cells. The most valuable property of the Voronoi diagram described above, is the information about neighboring regions. As the nodes in the season concept are operating fully decentralized, there is no central instance to do the client to server assignment. So every node has to be able to decide by itself if it is responsible for a certain client and if not the node has to know which node is or might be responsible. This can be guaranteed by the Voronoi concept, when each node is maintaining at least connections to every direct neighbor in the diagram: Assuming a given and correct client assignment, a client may move out of the responsibility region of a node, but because of the locality of normal movements
the node can determine the new node, which is responsible for the client. Even if the client is not moving locally (some kind of teleportation) the node is able to determine the next node in the direction, where the client moved, so it can give a hint to the client which node actually might be responsible. As this possible responsible node can do the same, the client will eventually be redirected to the correct node. The information to be maintained for making the mechanism described above is relatively few, as every node should only have a few neighbors in a normal Voronoi diagram. The use of Voronoi diagrams in server architectures was introduced by Hu and Liao [4].

2.2.2 Delaunay Triangulation

As described above, the main property of the Voronoi diagrams, which is needed for the season concept is the neighboring information. As the calculation of the nearest node for a client for a small, local set of nodes is trivial, there is actually no need for building the Voronoi regions. Therefore the Voronoi concept can be equivalently replaced by the concept of Delaunay graphs. A Delaunay graph is a triangulated mesh of points, where no points are placed in the circumcircle of the triangles. This is shown in Figure 3, where the gray circle is the center of the triangle built by nodes 0, 1 and 3. If another node would be present within the circle, this triangulation would be invalid. The process of building such a graph is called Delaunay triangulation, which maximizes the smallest angle in each triangle. The algorithm implemented in Season is called Flip algorithm and has a complexity of $O(n^2)$. Algorithm 1 is describing this algorithm in pseudo code: After building a random triangulation, each edge is checked for the 3 and if violating its flipped. This is done for all edges until no more invalid edges are found. This triangulation was

![Figure 2: Spatial decomposition into Voronoi cells](image.png)
Figure 3: Circumcircle of triangle as Delaunay condition

originally described by Boris Delaunay in [14]. A Delaunay graph is completely equivalent to the neighborship relationships resulting from a Voronoi diagram. Thus this Delaunay triangulation is used in the season concept to determine neighborships for all nodes. Each node is maintaining a local version of the Delaunay graph, containing only itself and its neighbors. Nodes are updating their neighbors with their position and are also capable of suggesting possible new neighbors to a node (see chapter implementation 4.1). Based on the position information of a node and its neighbors each node can now determine if he is responsible for a client, which enables season to eventually distribute the clients among the nodes.

2.3 Load Balancing by Movement Mechanism

The use of Delaunay triangulation allows the spatial distribution of clients, but has itself no load balancing mechanism. If all clients are crowded in one region, this region will be just overloaded, but no further actions will be taken as the system is still static.
Algorithm 1 Flip algorithm for Delaunay triangulation

\begin{algorithm}
\begin{algorithmic}
\State \text{buildRandomTriangulation}
\State \text{triangulationValid} \leftarrow \text{true}
\While{!\text{triangulationValid}}
\ForAll{\text{edge in triangulation}}
\State \text{valid} \leftarrow \text{checkDelaunayCondition(edge)}
\If{!\text{valid}}
\State \text{flip(edge)}
\State \text{triangulationValid} \leftarrow \text{false}
\EndIf
\EndFor
\EndWhile
\end{algorithmic}
\end{algorithm}

Therefore the nodes in the season concept are mobile and can move in the overlay. The goal of this idea is that nodes with fewer load will move towards nodes with higher load, this will shrink the responsibility region (Voronoi region) of the overloaded node, thus reducing its number of clients and thereby its load, because the load is taken over by the enclosing neighboring nodes. Therefore season consists of a movement mechanism for nodes to model this load balancing effect. Each node is putting a force on all of his neighbors. This force may attract or repel a neighbor. The sum of all forces being put on a node by his neighbors defines the direction of this nodes movement. This is shown exemplary in Figure 4: Node B and C are putting an equal force on node A because of their current load. The resulting movement vector is the sum of both force vectors. Depending of the current state one force may outrun the other force. So if a neighbor is heavily overloaded, this will impose a force which is probably bigger than the force resulting from their current distance.

A simplified version of the force calculation is shown in algorithm 2. Basically there are two different parts of the forces in Season: One part is resulting from the current load of the neighboring nodes and the other part is resulting from their current position. Those mechanisms are further described in the following subsections.
Algorithm 2 Force calculation of a node

for all node in neighbors do
    posDiff ← currentDistance\_defaultDistance
    if posDiff < 1 then
        F ← \( \exp(\text{neighborLoad}) - 1 - \frac{1}{\text{posDiff}} \)
    else if posDiff > 1 then
        F ← \( \exp(\text{neighborLoad}) - 1 + \text{posDiff} \)
    else
        F ← \( \exp(\text{neighborLoad}) \)
    end if
end for

2.3.1 Load Based Forces

One base information for the calculation of the force is the current load of a neighbor. An important property is, that it is based on neighbor information only. The load of a node itself is not considered when calculating the forces. This results in nodes, which are more willing to help others instead of being selfish. The use of this load based force is somehow trivial; it makes underloaded nodes being attracted by overLoaded nodes and thereby taking some load away. The load of a neighbor is not directly used for the force calculation, but is weighted by an exponential function, which has the following formula:

\[
\exp(\text{load}) = (20 \times 2^\frac{2}{3} \times 5^\frac{1}{3})^{\text{load}} \times \frac{1}{2^{\frac{2}{3}} \times 5^{\frac{1}{3}}} - \frac{1}{2^\frac{2}{3} \times 5^{\frac{1}{3}}}
\]  

This function maps the load, which is between 0 and 1, to \([0, 20]\) and is plotted in figure 5. The concrete function with its specific numbers is a result from calculations based on several desired properties:

- The function should map a load of 0 to a result of 0.
- The function should map a load of 1 to its maximum value of 20. This maximum is the value that has to be outnumbered by posDiff (see following section 2.3.2) if this property should be stronger than the load based force. So if posDiff is smaller than 1/20 this will outnumber every possible load based force. This could be also another value here, but experiments showed its usefulness.
- The function should be exponential and satisfy the general form of \(\exp(\text{load}) = c \times a^x - c\)

This function was mainly introduced for two reasons:

- The impact of the load on the quality of service is not proportional to the growth of the load, i.e. the impact of a load of 1.0 is not two times the impact of a load of 0.5. In this example a load of 0.5 is not a problem at all, but a load of 1.0 will probably lead to a significant draw back in terms of Quality of Service. This fact can be adapted by the properties of an exponential function very well.
• The described system may block itself if there are nodes with similar load in a symmetric arrangement. To reduce the impact of this *chess board problem* it is a good strategy to differentiate more even between similar load factors.

### 2.3.2 Spring Based Forces

Besides the load based force there is also a position based force in Season. This force is a result of the current distance of two neighboring nodes and their default distance when the network was initialized. This property is acting similar to physical springs, that is why we call this the *spring mechanism* in the Season system.

All neighbor connections, determined by the Delaunay triangulation, are acting (partly) similar to physical springs: If two nodes approach each other the *spring* in between will be bulged and therefore create bigger forces to repel the nodes again. On the other hand, if two nodes will increase their distance, the *spring* gets stretched and thereby will create stronger forces to attract the nodes again. As the node network is a fully meshed graph and all node relationships are modeled by the springs, this builds a so called *mass spring system*. This means that local changes of one nodes position may affect the whole network. A simplified example is shown in figure 6 where the node in the top left corner is moving and thereby influencing all nodes in the system. The ratio of current to default distance between two nodes is called *position difference* (posDiff) in Season as named in algorithm 2. The idea of modeling springs which enable participants moving in an overlay network was first introduced by Dabek et al. in [5]. As a result all nodes try to
maintain their original distances, which is not identical to their original positions. This may feel contradictory in the first place but actually offers some desirable properties:

• If an overloaded node is in between a client hot spot and some underloaded node, he will block the other nodes from support because he is shielding the hotspot. This is shown as an example in Figure 7. Node B could not help A because he is in the way and thus load balancing would fail. This is why B needs a mechanism to somehow push A beyond the hot spot so he can take over his share of the load. This can be achieved by the spring mechanism: As B is enclosing to A because of his load, their distance will shrink under their default distance. This will bulge the spring in between and thereby repel both nodes again. But as B is strongly attracted by A’s load, this load based force will outrun the repulsion forces in case of B’s movement. So this node is still able to gain on A while pushing this node away and essentially getting to the hotspot.

• The example above also reveals another advantage of the spring mechanism: As B is enclosing A, the distance to its neighbor C increases. This stretches the spring, which will result in a force to attract both nodes again. And again the heavy load of A will outrun this spring force and B continues approaching A but will also pull C behind in the direction of the hotspot. Finally this will enable C to get involved

Figure 6: effects of the mass spring system mechanism

Figure 7: combined forces with posDiff to push a node beyond a hot spot
in the distribution of the clients in the hot spot. Even without waiting until B is overloaded, so this makes C proactive.

- The last advantage of the mass spring system is the order it imposes on the system. That is because a distance between two nodes, which is getting extremely small or big will result in very strong spring forces outrunning all possible load forces. The load is naturally limited at the top, posDiff is not. Thus the springs make the whole system contracting and repulsing, but nodes can’t overtake each other or collide. Eventually this prevents any unstable state or collapse of the system.

2.3.3 Weighted Nodes

So far only the forces have been described in the sections above. But those forces only model the vector each node is moving along, it only affects the movement speed indirectly by the size of the force. The maximum speed has to be set manually by configuration, but nodes also may move slower than their theoretical maximum speed even when they are under the influence of strong forces. This is the case for heavily loaded nodes and is called weighted nodes. A node which is already overloaded is not able to take any additional load, so there is no point in approaching other nodes for support. Therefore nodes are getting slowed down in Season based on their current load. This is done through a squared weight factor in the range of [0, 1] which is multiplied on the position delta. So a small weight factor is slowing down a node more than a factor near to 1. The exact formula for this weight is: weight = 1 − load². If a node has no load at all this result in a weight of 1 which has no impact at all on its movement speed. If the load is increasing the size of the weight is decreasing in square compared to the load and thereby the node is slowed down. This again takes into account that we want to avoid overloaded nodes and therefore it is better to slow down nodes faster than linear.
3 Simulator

Figure 8: Chess board problem in simulator in different states

Before the implementation of the distributed version of the season concept, a piece of software called simulator was implemented. This software is able to calculate an arbitrary number of clients and servers with their specific behavior related to this context. The simulator has a graphical output to visualize the processes in the simulated client/server environment. All relevant parameters as number of clients, servers, speed etc. can be configured. Client actions are based on a movement model described below in this section. Networking or communication issues were not considered during the implementation. For the implementation of the simulator two different libraries were used:

- Irrlicht, a 3D graphic engine [8]
- Vast, peer to peer and Voronoi based network engine [7]

The main goal of the simulator implementation was to verify that the basic concepts can be applied as expected, related to load balancing issues. The visualization gives a better understanding of the characteristics and problems of the theoretical concepts. Therefore those concepts were adjusted several times during testing with the simulator. I.e. the VAST library was used in the first place to calculate the client to server assignments. But the necessary calculations for building the Voronoi diagrams are costly in sense of computational load and these diagrams provided a lot of properties, which were not necessary in the season system. So it was replaced later on by our own implementation of a Delanauy triangulation, which is cheaper and only covers all information used by season. Besides the implementation of the server’s basic concepts, the client model was a hot topic in the simulator development, because this model has to fulfill several requirements:

- Movement should be randomized
Although randomized, clients must not be equally distributed among the virtual world, but should be able of building one or more hot spots

- Movement has to be highly configurable
- Client behavior has to be exactly reproducible

Eventually the client behavior was implemented as a scenario mechanism, where behavior for each client is defined by events. Those events are calculated in advance and describes an action for a specific client in a specific time slot. Clients may gather at hot spots, arbitrary in number, size and position. If a client is attracted to a specific hot spot, is defined by its position and configuration values. The scenario mechanism allows to repeat the same client behavior various times with differing settings for the servers. So it is possible to test different approaches against each other. Because this mechanism proved its handiness in the simulator it was carried over for simulating (synthesized) client behavior in the real distributed environment.

3.1 Identified Issues - Chessboard Problem

As already said above the simulator revealed several problems with our approach. Most of them were eliminated by conceptual changes, but one problem still remained as it is immanent for the season approach. We call this problem in our context the Chessboard problem: A less loaded server can be stuck between two heavily loaded servers. If the load of those servers is equal then the forces emerging from this load are also equal. Thus the forces on the less loaded server in between neutralize themselves so that the less loaded server won’t move. That is why this server does not take any load from the heavier loaded servers. In a worse case scenario this constellation may also block neighboring servers, as all servers are (transitive) inter connected and influencing each other.

Figure 9a shows exemplary a server arrangement which may cause problems. The servers are symmetrically arranged similar to a chessboard. Clients are all starting in the center of the map and are over loading the corresponding server in the first place as indicated by the red area. The circles in the corners show the configured hot spots, where the clients are likely to gather later on in the scenario.

In Figure 9b the clients already spread from the center (nearly) equal to all hot spots in the corners. This causes the related server to be overloaded. Their neighbors are less loaded but won’t help them because all forces in the spring system are neutralizing themselves. This problem can’t be solved completely in our system. One counter measure is the exponential weighting of the load (compare Section 2.3.1), where the load is not directly included in the calculations but is a result of an exponential correction function. As a property of exponential functions, this functions increases the difference between forces emerging from similar loads. Furthermore the chessboard problem is also more unlikely to happen in a bigger network as it is based on symmetric server arrangements which are more unlikely to occur in bigger systems. Additional proposals for countermeasures can be found in future work (see Section 7).
4 Implementation Details

The following chapter describes the implementation of the distributed algorithm in detail. Concerning this the first part introduces the sequence of all processes which take place for each node. Afterwards the Client Bootstrapping is explained followed by the last part which addresses the robustness of the algorithm.

4.1 Sequence of Processes

To give a better understanding all features and their characteristics are addressed in a chronological order starting with the two phases Init and Run which are depicted in Figure 9. Thereby both phases describe a state a node can be situated in.

- Init State

The init state is used to build up the topology and to initialize the network. To do so each node loads at the beginning of process a list (in the global configuration file) which contains all other nodes, their IP addresses and start positions. This enables a node to build up an initial Delaunay graph which is used to identify the nodes of importance (neighbors). If he is able to connect to all of his neighbors he will proceed to the run state. Once a connection is refused by a neighbor, the node has to change to the optional join state. This step is necessary since a refused connection indicates a current state of topology which is unequal to the expected
initial one built up by the nodelist. This usually happens if a node joins the network at a later time.

- Run State
  Apparent from the init state, each node has a unique Delaunay graph and as a consequence each node has its own view on the topology. One of the major tasks of the run state is to keep that graph consistent by updating and validating it due to incoming packets sent by neighbors. This includes the construction of new neighborhood relations and the turn down of old relationships. Since these steps need some kind of synchronization and to simplify maintaining the consistency, the principle of ticks was used which guarantees that every node has the same amount of computational time. Consequently the duration of a tick is of great importance. If it is chosen too small a node could have not enough time and has to skip operations. If it is too high computation time is wasted.

The following sections describe all phases which are processed within a single tick.

- Delaunay Update
  The first phase of a tick forms a Delaunay update. Depending on the current node position and that of neighboring nodes, new neighborhood relations could be established which only become apparent through that phase. However, within this phase no further considerations will be made – it is used to prepare the subsequent phases.

- Node packet handling
  This phase is used to process all incoming packets sent by (also would-be) neighbors. Since each packet has a different meaning and is sent during different circumstances the various types are explained in the following sections corresponding to their trigger-response relation. This facilitates the differentiation of the various types of messages, thus avoiding confusion. However the following two types are generated by default by the network engine RakNet and form the basis for inter-node communication.

  - ID_NEW_INCOMING_CONNECTION
    This message identifier stands for a new incoming connection. It is usually answered with the subsequent identifier given that the node is neither overloaded nor both are already connected.

  - ID_CONNECTION_REQUEST_ACCEPTED
    Signals the receiver that his connection request was successful. Regarding the implementation this message identifier serves as a trigger for further messages (e.g. NEIGHBOR_REQUEST_UNSURE, . . .).
• **Spring Update**

Based on the current load and the load of neighboring nodes, a node gets pushed away or attracts other nodes which eventually leads to a position change in the topology. The basis for this calculation gets formed by the Default Position of a node as well as a specific load function that features exponential growth with an increasing amount of load (see *Section 2.3.1*). The default position describes the initial position of each node and is used to maintain default distances to all neighbors while load is decreasing within the neighborhood.

• **Node State Broadcast**

  - **ID_NEIGHBOR_STATUS**

    This message is sent within every tick to all neighbors of a node. It contains among other things the current position and load. All receivers use this information to calculate their position within the Spring Update and to determine neighborhood changes using the Delaunay Update.

• **New Neighbor Relations**

The new neighbor relations phase is the first one which processes the findings from the Delaunay update step and is used to recognize newly arisen neighborhood relations within the neighborhood of the node. As inherent part of this phase the node has to inform the corresponding neighbors about his new finding. This is of major importance since the neighboring nodes do not necessarily know each other and thus they would not be able to establish a relation on their own. However, as already explained at the beginning of this chapter the view of a single node is limited to his neighborhood. Connections beyond that border are hidden, which means that his perceived findings could also be false. For that reason every receiver of such a suggested neighbor has to validate its correctness.

  - **ID_NEW_NEIGHBOR_SUGGEST**

    This message contains the ip addresses, default positions and current positions of those nodes that need to know each other because of the findings of the sending node. It is sent to both of them.

  - **ID_NEIGHBOR_REQUEST_UNSURE**

    When a node receives an ID_NEW_NEIGHBOR_SUGGEST he will send this type of message to the proposed, potentially, neighbor. He also appends his own current position.

  - **ID_NEIGHBOR_REQUEST_REJECTED**

    The receiving node of an ID_NEIGHBOR_REQUEST_UNSURE rejects the request. This may have different reasons. On the one hand he has his own view on the topology and thus the requesting neighbor may not fit in it. On the other hand he could just have more up to date/obsolete position information about his current neighbors.
The node accepts the request. Both are neighbors now and add themselves to their own Delaunay graph.

- **Flipped Neighbor Relations**

![Flipped Neighbor Relations Diagram](image)

Figure 10: flipping relation

While a simple neighbor suggest is just a one way message without any response to the sender (even if the suggest was wrong) an advanced neighbor suggest which is caused by a flipped neighbor relation is a more critical message identifier which must not be ignored and where the sender expects a response. Picture 1 demonstrates such a case. Assuming Node A would send just a simple neighbor suggest to node C and D. Both of them could ignore the suggest just because of local, obsolete node positions. But node A would drop the relation to B anyway which, in the end, results in a missing edge. Within the presented implementation, node A will send an advanced neighbor suggest. This message contains, additionally to the information of a simple suggest, the node from which the edge was flipped (node B). Now, if node D rejects the neighbor request of C (or vice versa) it will send a new neighbor suggest to the initiator node A containing node B as the proposed one.

- **ID_NEW_NEIGHBOR_SUGGEST (Adv)**

The recipient of this message is able to extract the ip address and position of the node which is potentially his neighbor. Additionally the message contains the same information of the node from which the neighborhood relationship was switched.

- **Neighbor Handling**

During the neighbor handling phase physical connections are terminated which are no longer of any use. This is accomplished by comparing the current node connections with the Delaunay graph. Any connection which appears in the physical node list but which is not present as a relation in the Delaunay graph will be canceled.
- **ID_DISCONNECT**

  This message is sent to a former neighbor. Sender and receiver will remove each other out of their Delaunay graph, if not already done so.

- **Player position and migration**

  The remaining phase of a tick deals with the player packet handling. Since our load gets measured by the amount of connected players they own a key role. In terms of the implementation no real player instances were used. Instead, all incoming player events were generated by the so called pseudoclient (see Section 5 for entire description) which is capable of simulating multiple clients by playing back a previously loaded scenario. Nevertheless, meaningful messages types were used.

  - **ID_PLAYER_JOIN_REQUEST**

    This message type is sent by the pseudoclient and contains a unique player id as well as the future position.

  - **ID_PLAYER_JOIN_ACCEPTED**

    Contains the player id extracted from the request and signalizes a successful connection establishment.

  - **ID_PLAYER_JOIN_REJECTED**

    A reject is sent if a node receives a join request sent by a player for whom it is not responsible. For further information please see section bootstrapping.

  - **ID_PLAYER_MOVE**

    This message is sent by the player and consists of the id and new direction the player wants to move.

  - **ID_PLAYER_MIGRATION**

    At the end of each tick a node checks whether he is still responsible for a player based on his position. If not he will send an ID_PLAYER_MIGRATION message to the newly responsible node containing the player id and position.

  - **ID_PLAYER_MIGRATION_ACCEPTED**

    The receiver of a migration requests signalizes an accept with that message type. The remaining task of the requesting node is to send an ID_PLAYER_TAKE_OVER to the player and to remove him from his local structures.

  - **ID_PLAYER_MIGRATION_REJECTED**

    With the help of this message a node rejects a migration request. The receiver will be still responsible for that player. But as the case may be he will resend the migration request in the next tick.

  - **ID_PLAYER_TAKE_OVER**

    This message is sent by the former responsible node to the pseudoclient. It contains the player id and ip of the node which is responsible for him now.
- ID_PLAYER_QUIT
  Sent by the player when exiting.

4.2 Client Bootstrapping

Solely from the fact that the nodes are moving all the time, a player usually never knows which node is responsible for him and where it is situated for the initial connection request. For this very reason a bootstrapping algorithm was developed which enables the players to contact any of the nodes. If a node receives a PLAYER_JOIN_REQUEST it determines whether it is responsible for the transmitted player position or not by utilizing the Delaunay graph. If the node is not responsible for a given player it will send back a reject which contains an address of a neighboring node. This node could possibly be responsible for the player or at least it will be closer to the desired position. After receiving the reject the player just have to resend the join request to the appended node.

4.3 Robustness

An important aspect of a distributed system is the degree of robustness, i.e. to which extend the topology can deal with defects, without impairing the correct functioning of its implementation. During the development edge losses (and thereby missing connections) turned out to have a significant impact on the stability of the network, since it compromises the necessary communication between adjacent nodes. A reason for edge losses are wrong decisions of nodes when suggesting neighbors, which are triggered by asynchronous states of knowledge. Improvements, like additionally introduced message types (ID_NEIGHBOR_SUGGEST_SURE) may stem the errors, however fail in border cases. Nonetheless, the node topology is considered robust, as, due to the local view of each node, redundant information are exchanged anyway. And due to the fact that Season does not provide a centralized entity, these redundant information are unavoidable, and even improve the robustness, since they contain data for the nodes to rebuild lost edges.
5 Experimental Setup

Since the dynamics, introduced by the clients, such as movement, interaction, etc. are the actual cause for the load [15], an algorithm that partitions the clients over multiple servers is implemented, such that the entire server network is load-balanced. As the control layer mechanisms includes that each node is only interested in its neighboring nodes, the algorithm is perfectly suitable for a P2P-like distributed algorithm. Due to the Delaunay triangulation, the number of neighbors is usually low (see Section 2 – Basic Concepts). This technique can scale the number of nodes, and thus servers, that concurrently operate an entirely contiguous virtual world, significantly. Furthermore, the node network inherits typical P2P properties, such as robustness, or no single point of failure. While the Simulator (see Section 3 – Simulator) confirmed the general functioning of the control layer mechanisms, such as node pushing/pulling due to load and distances, weighted nodes, client-node mapping, etc., it is yet unclear, how the algorithm behaves in a distributed environment, exposed to latencies, asynchronous states and so on. The algorithm is extended with the communication layer, which enables the nodes to work independently in a distributed fashion in a network (see Section 4 – Implementation). So each node is controlled by one distributed server application, called Season, which communicates with other distributed server applications (other Season nodes) over a real IP network.

In order to perform several experiments using Season, a test bed was designed, that automates starting and stopping of experiments, and evaluates the resulting data. The
test bed consists of the following parts:

- **UML Machines**
  
  In order to save resources, *User Mode Linux (UML)* is used, which is a virtual Linux kernel, running as a user process on a Linux host. Each *Season node* is run on such a UML machine. Each single UML machine can consume as much CPU power as the host system can provide, since the UML machines itself are processes. Furthermore, 128 MBytes of RAM are assigned to each machine. The host system is a powerful machine, consisting of eight CPU cores, 16 GBytes of RAM as well as sufficient storage.

  The provisioning, and configuring of UML machines, and its control is supervised by bash scripts. Thus any number of UML machines can be booted up in parallel, each establishing an Ethernet connectivity using an UML switch. Afterwards the IP connectivity is established, too, so the *Season nodes* are able to communicate with each other, and with the pseudoclient. Furthermore, the UML machines have access to a NFS share on the host system, which holds the source code for *Season*, some scripts, and some configuration files, and has enough space for lots of log files.

  After the boot sequence has finished, first, the UML machines start a script that builds the current source code if necessary, and then starts the *Season* executable with a global configuration file.

- **Season**
  
  *Season* is the distributed server application that represents a node, and is run on a single UML machine. First, when *Season* starts it initializes the *Communication* layer for communication with other *Season nodes*, and for communication with the pseudoclient. Each *Season node* identifies itself and the other nodes using the nodelist (see Section 4), which contains start coordinates for the *Control* layer. At this layer the connectivity is calculated, so the nodes will know who their neighbors are, and thus where to connected to.

  Furthermore, each *Season node* has a separate connection to the pseudoclient, in order to simulate lots of clients.

- **Pseudoclient**
  
  The pseudoclient is a tool written to simulate multiple clients, while using only a single connection to each *Season node*, thus making the client handling easy. The pseudoclient uses traces containing events generated by the clients at a particular point of time, such as the join, quit, and movement, augmented with the clients position. These events are communicated between the pseudoclient and the nodes from the *Client* layer, as long as there are events. Thereby clients are identified by a unique ID.

  The experiments focus on the functioning of all implemented features, and mainly on the evolution of the load that is to be balanced. Information such as churn are
not regarded. Furthermore, there are no mechanisms that measure and analyze network specific data, such as generated traffic during client handovers between servers, nor the traffic caused by communicating control information to keep the Control layer calculations consistent. In order to simplify the experiments without losing the validity regarding the load-balancing mechanisms, the nodes communicate with the pseudoclient instead of thousands of individual clients. The events that are used to control those emulated clients are reduced to joining, leaving, and moving. Furthermore, the nodes do not calculate a real gaming environment, i.e. NPC’s, physics, etc., but a simple 2D Euclidean space with nothing but moving clients.
6 Evaluation

In this section the experiments that were performed are analyzed and discussed. There were several experiments performed, however only two are being regarded in this report. While the first experiment, called Feature Demo henceforth, focuses on the features of the implemented algorithm, the second experiment compares static and dynamic nodes on the identical set of original Quake3 client traces, and is called Quake3 Scenario henceforth. Afterwards the other experiments are briefly addressed, and finally, this section is concluded.

6.1 Feature Demo

![Graph](image)

(a) Initial state.  
(b) Tick 54, node 5 has all the peers.  
(c) Tick 334, node 5 is pulling the other nodes ...  
(d) Tick 560, ... into the crowded area.  
(e) Tick 635, nodes try to converge in an optimal state.  
(f) Tick 885, final state.

Figure 12: These figures show chronologically sequenced states of the topology in a 2D Euclidean space defined by the axes. The nodes (+ symbol) are annotated with an ID and the current load (in %), separated by a pipe.

The Feature Demo focuses on the features of the implemented algorithm, such as pushing and pulling neighboring nodes, edge-flipping, and trying to converge in a balanced state. In this experiment the clients do not move, in order to observe the pure behavior of the nodes. Figure 12 shows snapshots in chronological order, showing the states of
the node topology in a 2D Euclidean space. Five nodes are used for this experiment, ordered in a plus with a node in the center, as can be seen in Figure 12a. Furthermore, each node is able to handle 80 clients which amounts to an utilization of 100%. There is a total of 138 clients emerged between the diagonals running through the second and third quadrant. A bigger part of the clients appear in the second quadrant, also some accumulations can be noticed, see Figure 12b. Immediately after the clients have joined, node 5 takes the responsibility for all the nodes, which causes a load of 172%, thus the server is overloaded. By being overloaded, node 5 pulls nodes 1, 2, and 3 to a closer distance and closer to the clients as well, at the same time it is pushed by those nodes due to the decreasing distance. As nodes 1, 2, and 3 move left, they in turn attract node 4 with the increasing distance and with the new load taken from node 5, until all nodes finally reach the crowded area, which can be observed in Figure 12c. Meanwhile an edge-flip between nodes 1 and 4, and nodes 2 and 3 was triggered, because node 1 was pulled fast to the left which increased the distance to node 4, and nodes 2 and 3 moved together. Figure 12d shows how node 5 is pulled up towards the client accumulations by the slightly higher load of node 2. In following, node 1 has the lowest load, while node 2 has the highest load, see Figure 12e. Thus, node 1 is pulled towards node 2 and, at the same time, pushes it up, until another edge-flip is triggered, which makes nodes 4 and 5 move closer together. From now on the nodes do not move anymore significantly – they are in a converged state, which can be observed in Figure 12f. It is noticeable that the nodes have converged in the crowded area of the second quadrant.

Figure 13: Load evolution of the Feature Demo. The x-axis shows the time in ticks, while the y-axis shows the current moving average of the load in %. The parameters are five servers, while the last octet in the IP address represents the ID.
Figure 14: Moving behavior of the clients taken from real Quake3 traces. Darker spots are traversed very frequently. Clients move on the white areas as well, however with a insignificant frequency, compared to the gray areas.

*Figure 13* gives a closer look at the evolution of the load in the topology. The x-axis shows the time in ticks, while the y-axis shows the current moving average of the load in % using $\alpha = 0.01$. Furthermore, the last octet in the IP address represents the ID of the node. It is clearly noticeable how node 5 initially has all the load, while at tick 100 the load of the other nodes start to increase: Since nodes 1 to 4 try to help, they move into the crowded area and gradually try to take and balance the load from node 5. From tick 1000 on the node network is balanced – there are no more forces available that would change the position and form of this network, as long as the clients do not move.

From the algorithmic aspect, this experiment confirms the correct functioning of the springs. *Figure 12c* indicates not only the right interaction between the pulling and pushing forces, it also shows the correct use of the node weights. Nodes 2 and 3 have a higher load than node 1, which slows them down, so that node 1 moves faster towards the clients, while in the mean time an edge-flip occurred to keep the Delaunay conditions valid. Furthermore, the nodes are able to find a balanced state, starting in an uncrowded region and converging in the crowded region. Also, the nodes manage from within the crowded area to adapt to further imbalances, such as client accumulations, which means the nodes are virtually load-aware.

### 6.2 Quake3 Scenario

The *Quake3 Scenario* focuses on comparing static nodes with dynamic nodes, using an identical set of realistic Quake3 traces. While the dynamic nodes can move in the 2D Euclidean space, driven by spring forces and node weights, static nodes cannot move, thus maintaining the same position and connectivity throughout the entire experiment. This way traditional architectures with static nodes, and thus immutable regions, such as WoW employs it, can be simulated and compared against the dynamic approach.
Regarding the Feature Demo, it is already indicating what would happen if the nodes were not able to move: Node 5 is overloaded all the time (172%), while the others stay idle at 0% load. This subsection is divided into two parts, while the first analyzes the functionality of node movements, and the second regards the load-balancing.

Node Movement

Figure 15: These figures compare topologies from experiments with static (top), and dynamic (bottom) nodes, while both use an identical set of realistic Quake3 client traces. Each column represents a simultaneous snapshot of both experiments: I) Two groups of clients at two opposite corners. II) Two hot-spots at the center. III) Clients are spread evenly. The nodes (+ symbol) are annotated with an ID and the current load, separated by a pipe.

For the Quake3 Scenario realistic Quake3 traces are used. Figure 14 shows the areas that are traversed by clients. There are predefined paths the clients can take, some are less frequented, some are heavily used, such as the diagonal band. Furthermore three nodes are used, which are ordered in a triangle at the center of the virtual world, such that node 2 will have less clients to handle than nodes 1 and 3, in case of static nodes. There are exactly 100 clients in total, while each node can handle up to 80 clients, which amounts to 100% load. Thus the optimal distribution of the load would amount to around 41% for each node. Throughout the experiment the clients will move in groups and build flash crowds in regular intervals. Additionally, since Quake3 is an ego-shooter game, clients may shoot each other. If a client dies, it will respawn at the base of it’s
team which is either at the bottom-left, or top right corner of the virtual world.

Figure 15 shows a series of images, comparing static nodes (top), and dynamic nodes (bottom), whereas each column represents a simultaneous snapshots of both experiments. The characteristics of those snapshots are as followed: The first column shows two groups of clients at two opposite corners, which are the home regions for the teams fighting each other. Nodes 1 (53%) and 3 (62%) are assigned the biggest portion of clients, while node 2 (8%) has virtually nothing to do, thus the load is unbalanced. In the figure below, showing the dynamic counterpart, the triangle moves towards the bottom-left crowd, in order to reduce load on node 3, however at the same time nodes 1 and 2 are maintaining a certain declination to each other, such that the common Voronoi border cuts the upper-right crowd into half. This way both nodes equally share those clients, without being in immediate proximity of the crowd. The load distribution is well balanced, with loads ranging from 38 to 43%. The second column compares static, and dynamic nodes handling two hot-spots at the center of the map. Due to the initial node positioning, node 3 is unlucky with having two big chunks of clients right in its range of influence, thus being loaded at 78%, while nodes 1 and 2 almost only have half the load altogether (21 and 25%). The dynamic version, on the other hand, achieves loads ranging from 41 to 43%. It is noticeable that even though the arrangement of the nodes is not much different to that of the static version, the load distribution is significantly improved and almost perfectly balanced. Only a slight movement from nodes 1 and 2 towards node 3 is enough to balance the load, because the Voronoi borders are positioned such that both hot-stops are split, and thus shared among the three nodes. Finally, the last comparison bases on evenly spread clients, thus neither hot-spots nor gaps are available, and shows a similar load distribution as the first snapshot: Nodes 1 and 3 are responsible for 90% of clients which amounts to 50 and 62% load, while node 2 has a load of 12% serving 10 clients – the load is unbalanced. Again, in the dynamic version the load is well balanced,
Figure 17: Deviation from optimal state. Comparison of experiments with static and dynamic nodes, using realistic and highly dynamic Quake3 traces. The x-axis shows the time in ticks, while the y-axis shows the current moving average of the deviation in %.

ranging from 40 to 42%. The triangle places itself in the center of the diagonal band of clients.

Load-Balancing

*Figure 16* contrast the load evolution over time, for the *Quake3 Scenario* with static (16a), and with dynamic nodes (16b), respectively. For both graphics the x-axis shows the time in ticks, while the y-axis shows the current moving average of the load of the three nodes 1, 2 and 3. Since the tick duration is set to 500 ms (see Section 5), the experiments run an absolute time of 2.76 hours (10000 seconds). During that period the clients change their behavior radically in regular intervals, for instance they form hot-spots and spread across the map again, and resume their usual moving pattern. This behavior is reflected in *Figure 16a*, showing the experiment with static nodes: Although Nodes 1 and 3 are maintaining an average load of 56 and 58%, both have opposite load fluctuations beyond their average in long repeating intervals. At some points the load difference is 40%, looking at around tick 14.000 it is even 60%, while on the other hand, node 2, with having a load average of 11%, barely hitting the 20% threshold, exhibits a load difference of less than 5 to around 90%(!), regarding the other two nodes. The dynamic client movements cause a big imbalance of load, and through the lack of moving abilities the nodes cannot react in order to balance the load. As opposed to *Figure 16b*,
where the nodes are able to move, and thus adapt to the dynamic behavior of the clients. Nodes 1, 2 and 3 have virtually constant load which is a little bit more than 40%, and there is no sign of any influences by the clients. The average load ranges from 41 to 43%.

Figure 17 compares the deviation between the experiments with static and dynamic nodes, respectively. While the dynamic experiment shows a deviation between 3 and 10% (average is 5%), the static experiment exhibits a deviation between 15 and 33% (average is 21%). Furthermore, the impact of the dynamic client behaviour is noticeable in both graphs, however the dynamic nodes seem to weaken the effects.

6.3 Other Experiments

This section gives a very brief insight of other experiments, focusing on different methods that were used to evaluate and optimize the algorithms. Aside from the Feature Demo, and the Quake3 Scenario, the other experiments have been performed with many different settings in order to fully evaluate the functionality of season. The experiments can be divided into short- (2K ticks) and long-term (20K ticks) experiments, respectively. They vary in . . .

- Client Behaviour and Traces

In a variety of ways the nodes are stressed with diverse client behaviour. In order to simulate different scenarios, traces from the simulator (600 clients), as well as real Quake3 traces (100 clients) were used. While the simulator traces comprise of static hotspots, i.e. the hotspots have fixed position, the hotspots in the Quake3 traces are mutable. But besides that, the client movement is dynamic and arbitrary in both

Figure 18: Moving behavior of the clients taken from the simulator’s traces. Darker spots are traversed very frequently. The clients move within a big circle (light gray), while forming two static hotspots after a while (dark gray).
traces. *Figure 18* shows the moving pattern of the clients in the traces taken from the *simulator*. There is a big circular area filling the virtual space, along with two heavy hotspots. Those two hotspots emerge after a certain time at their determined position and stay there until the end of the experiments. The speed of the clients is customized according to the nodes’ speed. In some long experiments the client movement is turned off after around 15K ticks, in order to observe the convergence of the nodes. In a few shorter experiments the client movement is completely shut off in order to observe the pure node behaviour. Furthermore, few experiments were performed in which non-moving clients join the network successively at each tick. *Figure 19* shows the load evolution of the nodes of that kind of experiment. The y-axis shows the load in percent, while the x-axis shows the time in ticks. It is noticeable that after tick 600 – all clients have joined by that time – the load does not increase any further. Also, after some rearrangement the nodes converge to a stable state. So while the clients were joining the nodes kept moving such that the load distribution is balanced.

- **Number, Position, and Speed of Nodes**

  The number of nodes used ranges from 3 to 11. They were arranged specifically in order to create a more complex Delaunay connectivity, which in turn triggers frequent neighborship changes. Thus, finding and eliminating bugs at the *Commu-
Figure 20: Load evolution of an experiment with 11 nodes, comparing the static nodes against dynamic nodes. The x-axis shows the time in ticks, while the y-axis shows the current moving average of the load in %. After tick 15,000 the nodes stop moving.

Figure 21: Deviation from optimal state. Comparison of experiments with static and dynamic nodes, using traces from the simulator. The x-axis shows the time in ticks, while the y-axis shows the current moving average of the deviation in %.

...nication layer is more effective. For instance, Figure 20 shows a comparison of two experiments with static (20a) and dynamic (20b) nodes. For both graphics the x-
axis shows the time in ticks, while the y-axis shows the moving average of the load of the nodes. Due to the initial arrangement of the nodes and the subsequent emergence of both the hotspots, the load distribution in the static experiment ranges from around 20% to 90%, which is a load difference of 70%. Regarding the dynamic experiment on the other hand, the maximum difference between the loads shrink from around 40% at the very beginning to around 30% percent for the duration of the experiment. At some points the difference is even less than 20%(!). Figure 21 shows the according deviation contrasting both experiments. The results are similar to the Quake3 scenario. While in the dynamic experiment the deviation instantly decreases, and constantly is kept lower than 10%, the static nodes defer to the dynamic client behaviour, and the deviation increases. Experiments with five or less nodes have been performed as well. Furthermore, the movement speed, as well as the initial positions of the nodes have been altered, arranged in different constellations, such as a star topology, or a random configuration.

6.4 Conclusion

Season performs well. The experiments show that the load difference between multiple nodes can be significantly decreased. Even though the nodes can potentially block each other (Chessboard problem), they outperform the static nodes easily. In all experiments the load distribution was balanced to an adequate degree, which indirectly proofs the right functioning of the algorithms at the Control layer. Furthermore, the algorithms weakens the effects of even highly dynamic client behaviour, while static nodes do not even have a chance to adapt. Also the experiments show that the system is able to find a converged state, while trying to adapt to any inhomogeneities in the client distribution in the virtual world, i.e. different densities of client accumulations, hotspots and so on.
7 Future Work

In the following section some possible improvements of the Season concept are discussed, in order to further improve the system. Although they may improve the system, they were not implemented as they are out of scope of this work.

- **3D Extension**
  
  As already apparent from this work, the concepts and algorithms that were introduced and implemented, stick to a two dimensional space, for the sake of simplicity, which in case of three dimensional virtual worlds need to ignore the third dimension. Thus if a client moves vertically, it has no effect on any calculations, because its disregarded. This poses a problem regarding more complex environments such as the outer space, since potentially each dimension may be used evenly. In order to achieve a good load balancing it is advisable to perform the calculations in three dimensions instead of two, whereby the Voronoi regions would be replaced by geometric solids, so-called *Wigner-Seitz cells*, having equal mathematical properties. Thereby the construction of those cells would be similar as in 2D: while the Delaunay algorithms employs lines and circles, the 3D approach uses surfaces and spheres in order to calculate neighborhood relations, however, increases the computational complexity. Furthermore, using *Wigner-Seitz cells* increases the average number of neighbors for each node.

- **Join and Quit at Runtime**
  
  In order to perform experiments, and show the feasibility of Season it was not required to insert and remove nodes at runtime. However, with respect to a potential use case, inserting and removing nodes at runtime levers the advantages of using Season significantly. Thus for instance inserting or removing a node in case of an overall overloaded or underutilized system, respectively, would further improve the load balancing / scalability and mitigate conceptual problems, such as the Chess board problem, 2.3.1. Thereby both procedures are virtually easy to perform. Due to the capabilities of the *Control* layer, removing a node would lead the adjacent nodes to assume the resulted free space automatically, while a new inserted node would be placed to an optimal position.

- **Dynamic Voronoi Borders**
  
  The measurements have shown that the load distribution can be approached to an optimal state. However, in some cases in which the nodes mutually offset their attracting and repulsing forces, the movement is blocked (see Section 2.3.1 for Chess board problem), and so are the load balancing mechanisms. In order to improve the flexibility in case of a blocked system, the Voronoi borders may be independently enhanced with dynamics, such that the area of a node can be further shrunked in case of overload, while the adjacent areas grow. These changes in spatial decomposition would thus cause the neighboring nodes to also take the responsibility for clients that do not longer belong to the overloaded node. However,
the performance of this mechanism strongly depends on the position of the clients. Thus if the clients are located exactly in the middle of a nodes area, a reduction may induce only a marginal improvement, if at all.
8 Conclusion

In this work a distributed algorithm for balancing the load among multiple servers in virtual environments was successfully designed, implemented, and analyzed. Traditional client-server architectures in MMO games suffer from significant drawbacks, such as provisioning of servers, and a fair client-to-server assignment. The cause for this problem is a fundamental one – a static configuration of servers, and thus a static service is provided, to handle very dynamic and hardly predictable content, produced by the clients. As there is no load balancing mechanism that can adapt dynamically to the load produced by the clients, servers may become overloaded and/or underloaded. Thereby malicious or just unexpected behavior of client crowds may lead to severe problems regarding the quality of service. Also, in case of a re-design of the virtual world according to the user behavior, or underutilized servers, money and resources are wasted.

In order to balance the load among multiple servers, an algorithm was designed to continuously calculate fair client-to-server assignments, such that the load of each server is roughly the same. The algorithm is implemented to always try to gently reach an optimal geographical decomposition, by moving the nodes, and thus influencing the size of the regions. The functioning of these mechanisms, aggregated in the Control layer, were successfully validated in a piece of software called simulator.

Subsequently, the algorithm was implemented as a distributed server application with network capabilities, called Season, such that each Season instance represents a node. During the experiments each Season node is run on a virtual machine, communicating with other Season nodes over a real IP network in a distributed manner. The experiments are divided into two parts: The first set of experiments focuses on the right functioning of the Control layer mechanisms in a distributed environment, whereby the results could be positively checked with those from the simulator. The second experiments are again divided into static (nodes cannot move), and dynamic (nodes can move), in order to contrast the performance of the traditional approach against the Season approach. The results show that Season significantly weakens the effects of the dynamic client behavior and thus reaches a very constant load distribution throughout the duration of the experiments, which ranges very close around the optimal load distribution.

However, during the development of Season some conceptual problems were encountered. While some problems could be solved, other problems, such as the chess board problem (see Section 2.3.1) are addressed in the future work.
References


