Distributed database support for networked 
real-time multiplayer games 
HS-IDA-MD-02-005

Henrik Grimm

Submitted by Henrik Grimm to the University of Skövde as a dissertation towards the degree of M.Sc. by examination and dissertation in the Department of Computer Science.

September, 2002

I certify that all material in this dissertation which is not my own work has been identified and that no material is included for which a degree has already been conferred upon me.

Henrik Grimm
Abstract

The focus of this dissertation is on large-scale and long-running networked real-time multiplayer games. In this type of games, each player controls one or many entities, which interact in a shared virtual environment. Three attributes – scalability, security, and fault tolerance – are considered essential for this type of games. The normal approaches for building this type of games, using a client/server or peer-to-peer architecture, fail in achieving all three attributes. We propose a server-network architecture that supports these attributes. In this architecture, a cluster of servers collectively manage the game state and each server manages a separate region of the virtual environment. We discuss how the architecture can be extended using proxies, and we compare it to other similar architectures. Further, we investigate how a distributed database management system can support the proposed architecture. Since efficiency is very important in this type of games, some properties of traditional database systems must be relaxed. We also show how methods for increasing scalability, such as interest management and dead reckoning, can be implemented in a database system. Finally, we suggest how the proposed architecture can be validated using a simulation of a large-scale game.
# Contents

1 Introduction ............................... 1  
   1.1 Dissertation overview ..................... 3  

2 Background ................................. 4  
   2.1 Basic definitions ............................ 4  
      2.1.1 Networked multiplayer games ................. 6  
      2.1.2 Real-time games ........................... 6  
      2.1.3 Game genres ............................... 8  
   2.2 Communication paradigms ..................... 9  
      2.2.1 Event-based paradigm ....................... 9  
      2.2.2 State-based paradigm ...................... 13  
   2.3 Communication architectures .................. 17  
      2.3.1 Client/server ............................. 17  
      2.3.2 Peer-to-peer .............................. 20  
   2.4 Reducing resource requirements ............... 21  
      2.4.1 Dead reckoning ............................ 21  
      2.4.2 Interest management ....................... 24  
   2.5 Database concepts ........................... 29  
      2.5.1 Distributed and real-time databases .......... 30  
      2.5.2 Transactions .............................. 31  
      2.5.3 Consistency preservation ................... 32  
      2.5.4 Active databases .......................... 34  

3 Problem description ......................... 36  
   3.1 Focus ...................................... 36  
   3.2 Problem .................................... 38  
   3.3 Goals ...................................... 41
CONTENTS

4 A server-network architecture .................................................. 42
  4.1 Introduction ........................................................................ 42
  4.2 Regions ............................................................................. 46
  4.3 Interest management .............................................................. 49
    4.3.1 Optimizations with multicast ........................................ 51
  4.4 Proxies .............................................................................. 53
  4.5 Related work ...................................................................... 55
    4.5.1 The mirrored-server architecture ................................... 55
    4.5.2 RING ........................................................................... 57

5 Distributed database support ....................................................... 59
  5.1 Introduction ........................................................................ 60
  5.2 Consistency requirements ...................................................... 61
  5.3 Conflict resolution ............................................................... 62
  5.4 Interest management ............................................................. 64
  5.5 Dead reckoning .................................................................... 65
  5.6 Reducing transaction overhead .............................................. 67
  5.7 Fault tolerance .................................................................... 68

6 Conclusions .................................................................................. 70
  6.1 Contributions ....................................................................... 70
  6.2 Discussion .......................................................................... 72
  6.3 Future work ......................................................................... 74
    6.3.1 Validation of server-network architecture ....................... 74
    6.3.2 Database support ........................................................... 75
    6.3.3 Maintainability ............................................................... 77

Acknowledgements .......................................................................... 78

Bibliography .................................................................................... 79

List of figures ................................................................................... 83
Chapter 1

Introduction

A networked virtual environment is a simulation that allows participants from physically distant places to connect via a network and interact in a shared virtual environment. Each participant controls one or more entities with some input devices (such as mouse and keyboard) and actions taken by each participant can affect the virtual environment and other participants. Networked virtual environments can be classified by their applications. One category is military simulations (Page & Smith 1998) which is used for training military personnel and for testing military equipment. Another category is virtual meeting places, where the participants can interact and communicate in a virtual environment. A third category is networked real-time multiplayer games (Smed, Kaukoranta & Hakonen 2001) where participants are given goals, which they try to reach either by competition or by collaboration.

The problem that is investigated in this dissertation is how to build large-scale and long-running networked real-time multiplayer games where large-scale means
that many players participate in the game and long-running means that the game is played over a long period of time (days or years). We consider three attributes – scalability, security, and fault tolerance – important for this type of games. Since the basic approaches fails in achieving all these attributes, our aim is to design a solution which achieves them.

We propose the use of a server-network architecture, where multiple servers are connected in a network and each server handles a number of clients. Further, database issues are investigated and the feasibility of using a distributed database management system (DBMS) with this architecture is explored.

Our architecture achieves scalability, since each server manages a separate region of the virtual environment and only needs to know about regions that are close to its own region. Security is achieved by having secure servers manage the game state. Fault tolerance is achieved by having neighbor servers (temporarily) take control of a crashed server’s region and entities.

Advantages of using a DBMS for networked real-time multiplayer games is that it can reduce development cost and increase maintainability, since the developers do not have to implement the functionality provided by the DBMS. A disadvantage is that a DBMS may introduce considerable overhead, since most database systems prioritize correctness before performance. In this dissertation, we investigate how this overhead can be minimized, and how a DBMS can support networked real-time multiplayer games.
1.1 Dissertation overview

In Chapter 2, we present the background material related to networked real-time multiplayer games and distributed databases. Chapter 3 states the problem and the aim that are investigated in this dissertation. In Chapter 4, we propose a server-network architecture and design decisions for building networked real-time multiplayer games. Chapter 5 continues by looking at the database aspects of the server-network architecture and investigates the feasibility of using a distributed database management system when building this type of systems. Finally, Chapter 6 concludes the dissertation by summarizing and discussing the contributions and listing suggestions for future work.
Chapter 2

Background

This chapter describes networked real-time multiplayer games and issues that are relevant to the problem investigated in this dissertation. In Section 2.1, definitions for basic game concepts used in the dissertation is given. Section 2.2 presents the two basic paradigms used for sending information in networked real-time multiplayer games. In Section 2.3, the two most common communication architectures for such games are presented. In Section 2.4, methods for reducing resource requirements are described. Section 2.5 introduces basic database concepts that are necessary to consider their use in supporting networked real-time multiplayer games.

2.1 Basic definitions

Generally, game software consists of a game engine and resources. The game engine is the program that is executed to play the game. Resources are static data that
are used by the game engine and are often stored in separate files, examples are graphical textures, 3D models, sound files, and maps over the virtual environment. A game engine has a number of functions, which include presenting information to the user, handling input from the user, managing resources, storing and updating the game state, and handling events.

A user interacting with a game engine is called a player. A player can issue actions (or commands) through the game engine’s user interface. The game engine generates events, for example, when a player issues an action, as a random occurrence, or when some condition is met. Eventually each event is handled by the game engine, which may update parts of the game state, generate new events, and/or present the event to the player (e.g. playing a sound).

A game execution is an execution instance of a game engine. In a game execution there exists a number of entities. An entity is something that has a state and behavior, e.g. a character or an item (Funkhouser 1995b). Most entities represents physical objects in the game execution and have a position variable as part of their states. Positions are normally expressed as a coordinate in the virtual environment. The virtual environment may itself be represented by one or many entities. Some entities may be controllable by one or more players. An entity that is controlled by a single player is referred to as that player’s entity (also commonly called the player’s “avatar”). The state of all entities are collectively called the game state and defines the game execution at some point in time.
2.1.1 Networked multiplayer games

In a networked multiplayer game, two or more nodes are connected by a network and each participating player is controlling his/her entity through one of the nodes (it is possible that many players share one node). Each node has its own copy of the game engine. During a game execution, the nodes communicate with each other and the game state is shared between them.

Normally, resources are stored at each node and are not part of the game state. This is done to avoid that nodes have to send large amounts of data during the game execution. Entities can be associated with resources by having explicit references in the entity’s state or implicitly by the type of entity.

2.1.2 Real-time games

In real-time games, the game state is updated at regular intervals and the virtual game time follows wall-clock time. This means that even if no player actions are issued, the game state is still updated. Games that are not real-time are commonly called “turn-based games”. In turn-based games, the game state is updated only when player actions are issued, independently of wall-clock time. Often, the player have a number of possible actions to do in each turn, before the next player can continue or the turn is over. Examples of turn-based games are card games, chess, and monopoly. In this dissertation only real-time games are considered.

A real-time game is normally a soft real-time system. A soft real-time system is a system where response times are important but where it is not imperative that
responses occurs within specified deadlines (Burns & Wellings 1997).

In real-time games it is important to have a high responsiveness, which is achieved by minimizing the time from when a player issues an action until consequences of that action are visible to the players. This time is called the latency of the action. In a networked real-time game, we can differentiate between local latency and remote latency. Local latency is the time it takes until the consequences of an action are reflected in the state of the local node of the player issuing the action. Remote latency is the time it takes until the consequences of an action are reflected in the state of other (interested) players’ nodes. Consequently, local responsiveness is increased by decreasing local latency, and remote responsiveness is increased by decreasing remote latency. The maximum acceptable latencies depend on the type of game. Often, in a fast-paced game, like a driving simulator, the local latency must be small to not annoy the players. Remote latencies are often longer and more unpredictable than local latencies because of delays introduced by network communication.

In real-time games, the game state is updated at a frequency, called the update rate. During these update intervals, the state of dynamic entities are updated to reflect the passage of time. Further, the game state is presented to the player at a frequency, called the frame rate. In a single player game, the update rate and the frame rate may be equal, but they do not have to. For example, the update rate can be higher than the frame rate to increase the physical accuracy of the simulation. A higher frame rate may increase the responsiveness and visual experience for players, because the effect of an action is more likely to be presented faster and movement of entities are smoother.
2.1.3 Game genres

In this section some popular game genres are presented. Note that it may not always be possible to classify all games into a single genre, but many real-time games fit into one of the following.

In first person shooters (FPSs), the player controls a single entity while trying to achieve some goal (e.g. to defeat all or some of the other players or to capture some possession of the opponent). This type of game requires fast and accurate movement and aiming and therefore high responsiveness. The game state is often small and only a subset of it is of interest for each player (e.g. the player often sees only a small part of the map).

A type of game with similar characteristics is driving simulators. Pantel & Wolf (2002) have investigated the impact of delay in a driving simulator and conclude that the local latency should never be more than 100 ms to provide a realistic driving behavior.

In real-time strategy (RTS) games, the player is in control of sometimes up to hundreds of individual units aiming at some goal (which often is to destroy the opponent’s base while preventing the opponent from destroying the home base). Here, the player issues actions at a higher level (e.g. go to a point, attack an opponent’s unit) and the requirement on responsiveness is not as high as for FPSs. Studies done on the RTS game Age of Empires indicate that the game is still very playable up to a local latency of 500 ms (Bettner & Terrano 2001).

In role-playing games (RPGs), the player controls a single entity or sometimes a small group of entities. The goal is to explore large virtual environments, gather
items and experience, and fight enemies. The actions issued by the player are often, as in RTS games, at a higher level than for FPSs (e.g. attack a monster, cast a spell) and hence RPGs typically have lower requirements on responsiveness than FPSs. The game state is often very large (there tends to be a large number of entities in RPGs), but normally only a small fraction of it is of interest to a single player at a given time.

2.2 Communication paradigms

In networked games, two different paradigms for distributing information between nodes can be identified. We call them the event-based paradigm and the state-based paradigm, based on the type of information that is sent between nodes. These two paradigms are presented in the following subsections.

2.2.1 Event-based paradigm

When using the event-based paradigm, the game state is fully replicated at every node that controls the game state. For simplicity, the following discussion assumes that this includes all nodes that participate in the game execution. When an event is generated at a node, it must be reliably sent to all other nodes, which in turn handle the event. Thus, all events are handled by all nodes that control the game state.

The event-based paradigm is most suitable for games where there is a small number of players and the network is relatively reliable, because every event (e.g.
player action) has to reach every node. Further, it is suitable when large parts of
the game state are visible to each player (because each node holds the entire game
state) and player actions are relatively sparse and high-level. A type of games that
often has these properties and where the event-based paradigm have been used is
RTS games (e.g. Age of Empires, Bettner & Terrano 2001).

A requirement for using the event-based paradigm is that the game execution is
deterministic, i.e. given the same initial game state, if the nodes handle the same
events at the same time and order, then the resulting game state is identical on all
nodes.

Only unpredictable events (e.g. player actions) need to be sent to other nodes.
Predictable events are generated independently at each node, since each node has
the full game state. Even events that are generated as a consequence of the output
from a pseudo-random number generator (which by definition is deterministic) can
be predictably generated by each node if all nodes start with the same seed and use
the generator the same number of times.

**Synchronous execution**

The simplest approach to guarantee that all nodes handle the events at the same
time and in the same order is to execute the game in a synchronous (or lock-step)
mode. In a synchronous mode the game execution proceeds in discrete time steps
and a node cannot proceed to the next time step until it has received all events from
the other nodes for the current time step. When a node has received all events, they
are handled in some deterministic order and the node proceeds to the next time step.
Even if no events are generated at a node at some time step, it still has to send a “no events” message so that the other nodes can proceed.

A problem with executing in synchronous mode is that the virtual time of a game execution may fluctuate compared to wall-clock time. This depends on the unpredictability of the nodes and the network. Further, all nodes are degraded to the speed of the slowest node. If a node crashes, the game is stalled, until a maximum latency bound is reached when a node is considered crashed and excluded from the game by the other nodes.

To minimize the time fluctuations, events can be scheduled for some later time step instead of being handled immediately (Mauve 2000a, Bettner & Terrano 2001, Gautier & Diot 1998). A local lag is introduced so that events are delayed for a fixed number of time steps, even at the local node. Because events are handled at later time steps, the remote nodes do not have to receive them until they reach the time step when they are to be handled. Hence, if the local lag is $d$ time steps, then a node can proceed to time step $t + d$ until it needs the events that was generated at time step $t$. The cost of using this method is reduced responsiveness and hence there is a trade-off between high responsiveness (low local lag) and less time fluctuations (high local lag).

Asynchronous execution

Mauve (2000b) proposes an algorithm called Timewarp, where the nodes execute asynchronously to avoid time fluctuations and reduce message passing. Timewarp is an optimistic algorithm because it allows temporary inconsistencies, but it converges
Figure 2.1: Example of a timewarp. A new event from a remote node is received at the current time step, but it should have been executed at an earlier time step. A timewarp is made to snapshot 2 and all events between snapshot 2 and the current time step are replayed.

towards a consistent and correct state via backward conflict resolution. With Timewarp, nodes do not wait for event information from other nodes until they proceed to the next time step. Assuming that nodes do not generate any events at most time steps, this avoids having to send and wait for many “no events” messages. However, it also means that events may be received that should have been handled at a time that has already been passed. In these situations a temporary inconsistency has occurred and has to be resolved.

With Timewarp, the nodes periodically makes snapshots of the game state and keeps a list of all events they received. When an event is received which was scheduled for a time step before the current time step, a timewarp is performed. The game state is reset to the last snapshot that was made before the event is scheduled and all events
after the snapshot are replayed, including the newly arrived event (see Figure 2.1). The new event may affect the events executed after it (it may even invalidate them). Old snapshots and events may be deleted when it is sufficiently unlikely that an event arrives which precedes them. A problem with timewarp is that it is computationally expensive. The entire game state is copied at regular intervals to make snapshots, and in the case of inconsistencies, the game state has to be copied back from a snapshot and events have to be replayed. Further, rendering of inconsistent state information (e.g. the explosion of a mine that should not have happened) may distract the player who may make decisions based on an incorrect game state. To reduce the number of inconsistencies and hence the number of timewarps that have to be done, a local lag can be introduced to decrease the likelihood of events from remote nodes being too late. Hence, there is a trade-off between high responsiveness and decreased likelihood of inconsistencies.

2.2.2 State-based paradigm

When using the state-based paradigm the nodes may execute asynchronously at their own rates while having synchronized clocks. Each entity is usually owned by a single node, and only the owner is allowed to update the state of an entity. Thus, unlike the event-based paradigm, an event is only handled by the nodes that own entities that are affected by that event. The nodes hold surrogates for entities that they are interested in but do not own themselves (Funkhouser 1995b). A surrogate is an, often simplified, model of an entity that represents a node’s knowledge of that entity.

When the state of an entity is changed (as the result of one or many events), the
owning node sends state updates to other nodes to inform them about the entity’s new state. A state update holds the new state for an entity, and each node receiving a state update replaces the state of its surrogate with the new state.

An advantage of the state-based paradigm is that most state updates completely replace old state updates (e.g. position of entities). This means that state updates do not have to be sent whenever a state change occurs, but can instead be sent at some fixed interval or when the state has changed by some maximum amount (see dead reckoning in Section 2.4.1). Further, these state updates can be sent unreliably and the applications do not have to detect lost messages. If messages are reordered and/or duplicated by the network, a receiving node can simply use the most recent state update and drop the others.

The generation of some events (for example collision between entities) may depend on multiple entities that are not owned by the same node. Because different nodes may have inconsistent versions of the game state they may not individually generate the same events (as was the case with the event-based paradigm). In such situations it must be determined who should generate a possible event. For example, if an entity A fires at another entity B, then the decision if B was hit can either be done by the node that owns A, the node that owns B, or possibly another node (e.g. a server).

**Surrogate prediction**

Because network resources are limited, state updates are often received at a coarser interval than the update rate of the game execution. Surrogates can be simulated be-
Figure 2.2: Prediction by extrapolation. $P_0$ and $v_0$ is the entity’s last known position and velocity, respectively. The predicted position ($P$), is equal to $P_0 + v_0(t - t_0)$, where $t$ is the current time and $t_0$ is the time when the entity was at $P_0$.

between updates to minimize the degree of inconsistency and get a smoother transition between consecutive state updates (Funkhouser 1995b). This is done by predicting the entity’s state for each update interval. For example, by having the position and velocity for a moving entity at the time of the last state update, the current position can be extrapolated by taking the velocity times the difference between the current time and the time of the last update and adding this to the position (see Figure 2.2). The velocity of an entity can either be sent as part of the state update or be approximated from its position in earlier state updates. Other factors may also be taken in consideration to improve the prediction, like acceleration and angular velocity of entities and external forces like gravity. The simulation of a surrogate can be arbitrarily complex. Adding more complex predictions decreases the degree of inconsistency at the cost of increased computational requirements.

A node may also predict collisions of surrogates to, for example, avoid them from moving temporarily through walls. On the other hand, if the simulation results in a projectile hitting an entity, the node cannot assume that this really happened until it gets information from the node that is responsible for deciding if a collision
occurs. This can have some strange consequences. A player may, for example, see a projectile moving past an entity, when later the player’s node gets a state update saying that the entity was hit, which results in that the projectile disappears and the entity taking damage. This is a disadvantage with the state-based paradigm; a player may think that a decision is unfair because with the state shown to the player the decision might have been another. Further, players may take actions that they would not have done if they had shared a consistent view of the game state.

A problem of doing prediction by extrapolation is that it may result in jerky movement, because unpredictable events (e.g. player actions) make the state of the entity diverge from the predicted state. When a state update is received, the surrogate either has to “warp” to the correct state or some other corrective action has to be taken to repair the surrogate’s inconsistent state. The degree of inconsistency can be decreased at the cost of responsiveness by introducing a local lag so that state updates are sent before they are in effect (Gautier & Diot 1998).

In games where responsiveness cannot be compromised, introducing a local lag is not acceptable. In the FPS game Half-Life (Bernier 2001) another approach is used. Surrogates of remote players’ entities are always simulated at a fixed interval in the past in respect to the entities. Thus, if this interval is larger than the remote latency, a node may know an entity’s “future” state and can interpolate the surrogate’s state between a known past state and a known “future” state (see Figure 2.3) and hence avoid the jerkiness that results from extrapolation. The cost is that the inconsistency between nodes is increased, which is a problem when entities owned by different nodes interact.
2.3 Communication architectures

A communication architecture defines possible communication paths between networked nodes (Smed et al. 2001). In this dissertation, communication architectures are for simplicity referred to as just architectures. The following subsections describe the two basic architectures, client/server and peer-to-peer.

2.3.1 Client/server

When using a client/server architecture an extra node is added and given the role of server. The other nodes are referred to as clients. There is no communication between
 CHAPTER 2. BACKGROUND

clients, only between each client and the server (see Figure 2.4). A client/server architecture can either be centralized or decentralized, depending on whether the server is in total control of the game state or not (Baughman & Levine 2001).

With a centralized client/server architecture, the server owns the complete game state and each client holds surrogates for the entities that are of interest to that client. No entity is owned by a client, not even the player’s entities. When a player issues an action, the player’s client sends that action to the server, which regularly computes the new game state resulting from the players’ actions and other events and sends state updates to the clients.

An advantage with a centralized approach is that it is simple. The server is the single authority and makes all decisions, for example whether a player hits another player, regardless of the clients’ views of the game state. Further, the server can decide what each client needs to know and filter information that is irrelevant (see server-based filtering in Section 2.4.2). Another advantage with this approach is that it can protect against cheating, assuming that the server is secure. The clients can only affect the game state by sending player actions to the server, which can accept or reject them.

While the server often has more capacity than the clients and can have a simpler user interface, it is a bottleneck both in communication and computation. It is a bottleneck in communication since all messages are either sent from or sent to the server and the number and size of messages increases with the number of clients and entities. It is a bottleneck in computation since it has to update the complete game state at regular intervals. Both of these problems result in a system with bad
scalability regarding the number of clients and entities.

The computational requirements on the server can be reduced by using a decentralized approach. With a decentralized client/server architecture, all or part of the game state is handled by the clients, reducing the requirements on the server. For example, movement and collision detection of a player’s entities can be handled by the client, which then sends the new coordinates of the entities to the server. But by giving the clients control of the game state, some of the security of the centralized approach is lost. If collision detection is performed by the clients, then a player can modify the local game engine to never detect collisions, giving the player an unfair advantage. Some form of cheat detection may be added to the server to try to detect this type of cheating, but then some of the advantages of the decentralized approach are lost, because the server has to redo some of the clients’ computations (but perhaps not with the same requirement for responsiveness).

An advantage of having a server is that it can handle functions that are not logically controlled by any client (Funkhouser 1995a). For example, a server can handle entities that are not controlled by any player, and store persistent information. A server also provides a convenient location for coordination and synchronization of global functions (e.g. generation of unique entity identifiers).

Regardless of whether a centralized or decentralized approach is used, a disadvantage of client/server architectures is that there is an added client-to-client latency, because all communication must go through the server. Furthermore, the server is a single point of failure. When the server crashes, clients cannot communicate and all or part of the game state is lost or must be recovered (unless the entire game state
2.3.2 Peer-to-peer

With a peer-to-peer architecture (or distributed architecture) every node has a communication path to all other nodes (see Figure 2.5). The game state is distributed among these nodes. If the event-based paradigm is used, then each node holds the complete game state, and when a player issues an action, it is sent to all other nodes. If the state-based paradigm is used, then each node may own some entities (normally those that are controlled by the local player), and state updates are sent to other interested nodes when the entities are updated.

Peer-to-peer architectures do not have the same disadvantages as client/server architectures. Since no server is used, node-to-node latency is smaller and there is no single point of failure. However, peer-to-peer architectures have some disadvantages. First, there are no trusted nodes, so it is harder to prevent cheating. Second, if the game execution contains complex entities not controlled by any player, then it is not obvious where to store the state of these entities. They need to be replicated at more than one node to tolerate node failures.
Examples of games using a peer-to-peer architecture are the FPS game *MiMaze* (Gautier & Diot 1998) and the RTS game *Age of Empires* (Bettner & Terrano 2001).

### 2.4 Reducing resource requirements

There are a number of methods to reduce resource requirements (computational capacity and network bandwidth) and hence increase scalability for networked real-time multiplayer games. General methods for reducing bandwidth requirements are to compress and merge messages (Smed et al. 2001). Two methods that are often used in networked real-time multiplayer games are dead reckoning and interest management, which are described in the following subsections.

#### 2.4.1 Dead reckoning

*Dead reckoning* is a method for reducing the transmission frequency of entity state updates when using the state-based paradigm (Page & Smith 1998, Mauve 2000b). As explained in Section 2.2.2, each node holds surrogates for entities that it is interested in, but do not own itself. Between updates, the surrogates’ states are predicted at each node, for example by using extrapolation of position and velocity. The idea behind dead reckoning is that as long as the divergence between an entity and its surrogates at other nodes is small enough, no state updates need to be sent, because they would not make a large difference to the surrogates’ states.

When dead reckoning is used, a node that owns an entity also keeps a surrogate for itself, which is called a *ghost* to distinguish it from the other surrogates. The
ghost is updated whenever an update message is sent to the other nodes, and the state of the ghost is predicted in the same way at the owning node as the state of the corresponding surrogates on the remote nodes. Thus, the node owning an entity knows the presumed state of the other nodes’ surrogates for that entity (assuming that the last state update was received by the remote nodes). The node can calculate the difference between the entity’s state and the state of the ghost to determine the divergence (i.e. mutual inconsistency) between the entity and its surrogates at remote nodes.

When the difference between an entity and its ghost exceeds a specified threshold, a new state update is sent to the other nodes and the state of the ghost is updated (see Figure 2.6). Thus, as long as the entity closely follows a predicted path no update messages have to be sent. The higher the threshold is, the less frequently updates are sent at the cost of larger divergences between the surrogates and the entity.

Dead reckoning is mostly used in the spatial domain for prediction of positions, but it applies for all types of states when there is some meaningful way to extrapolate state information and to measure the difference between the entity and its ghost.

With dead reckoning, less information has to be sent on the network and nodes receive and process fewer messages. The cost is that both computational and storage requirements are increased for the nodes owning entities, because they have to keep track of a ghost for each entity.
CHAPTER 2. BACKGROUND

Figure 2.6: Example of dead reckoning in a two-dimensional spatial domain. The black dots are the entity’s actual positions and the gray dots are the positions of the ghost for time steps 0 to 4. At time step 3 the distance between the entity ($e_3$) and the ghost ($g_{3a}$) exceeds the threshold ($t$). As a consequence, a state update is sent to the remote nodes and the ghost’s position is updated ($g_{3b}$).
2.4.2 Interest management

When the state-based paradigm is used, only a subset of the game state is relevant to a node. Thus, when a state update is sent, only a subset of the nodes needs to receive it. The concept of filtering irrelevant information is known as *interest management* (Morse 1996). Filtering of information can be done at the receiving node, but then network bandwidth is wasted on information that is never used and the node has to spend resources on filtering this information. As the number of entities grows, more and more resources must be spent on receiving and filtering an increasing number of messages, which results in bad scalability. The alternatives are to do the filtering at the sender or let the network do the filtering by using its multicast capabilities.

**Area of interest**

An *interest expression* is a specification of the information that is relevant to an entity or a player (Morse 1996). An interest expression may refer to several attributes of entities, e.g. position and type. In a two-dimensional spatial domain, the interest expression is often represented as an area of interest. An entity’s area of interest defines which part of the state space that is of interest to that entity. Everything outside an entity’s area of interest is not relevant to that entity. An area of interest is usually relative to the position of the entity, e.g. an entity may want to know about all other entities within a specified radius of itself. An example of this is given in Figure 2.7.

A state change affects some parts of the state space; the *area of influence*. When a state change happens, it is only of interest to an entity if the area of influence

24
intersects with the entity’s area of interest. A player’s area of interest defines the part of the state space that is relevant to the player. While the player’s area of interest depends on the state of the controlled entities, it may not be the same as the controlled entities’ areas of interest, since an entity may be affected by state changes that are not relevant to the player and vice versa. A node’s area of interest is the union of the areas of interest of all entities that are owned by the node and the player’s area of interest (if there is a player at the node). A node’s area of interest is used to decide if a state update should be sent to the node or not. Whenever an entity moves, it is checked whether the area of influence (which depends on the entity’s position and size) intersects with other nodes’ areas of interest and the state update is sent only to those where this is the case.

**Server-based filtering**

When using a centralized client/server architecture, interest management is normally handled by the server. The server knows the complete game state and can decide
what parts of the game state is of interest to each client. When a client sends an event to the server, the server computes the new state and informs only those clients that are affected by the state change. The filtering is not required to be perfect as long as at least those messages that are relevant to a client are sent to that client. A coarse and computationally inexpensive filter algorithm may be used to decrease the computational requirements at the server, at the cost of increased network requirements and added computational requirements (for filtering) at the clients.

**Group-based filtering**

With a decentralized architecture, a node does not hold the complete game state, and hence cannot always decide who the interested recipients are when sending information. A general solution to this is to use *grouping* (Aronson 1997). A *group* is something that a node can join or leave at any time and all messages sent to a group are sent to all nodes in that group.

If supported by the network, multicast can be used for grouping. Filtering is then performed by the network and the nodes are relieved from this work. However, using multicast on the Internet is not without problems. Abrams (1999) and Funkhouser (1995b) list a number of issues, including:

- The available address space is limited and the number of unique multicast groups may not be sufficient for a complex game execution.

- Delays and implicit messages generated when joining and leaving multicast
groups make it impractical to use when group memberships are very dynamic.

- Legacy hardware problems. Not all routers are configured for IP multicast and many network interface cards are limited in their multicast capability.

- Multicast is unreliable by nature and because senders cannot generally know to whom they are sending, they cannot know if the message they sent was received by all nodes that are members of the group.

Zou, Ammar & Diot (2001) discuss two strategies for grouping entities: cell-based grouping and entity-based grouping. In cell-based grouping the state space is divided into cells, where a cell is some delimited part of the state space. Further, each entity has two sets of cell references, a sending set and a receiving set. Each cell is associated with a group and the entity joins the groups of the cells in the receiving set and sends state updates to the groups of the cells in the sending set. Zou et al. (2001) define the sending set as the cells where the entity currently resides (which usually is a single cell) and the receiving set as all cells intersecting with the entity’s area of interest (see Figure 2.8). The sets are updated when an entity moves over the playing field and the entity joins and leaves the groups accordingly.

Note that the cell division will only approximate the entity’s area of interest. The fewer cells used, the more state updates are sent to the entity from sources outside the entity’s area of interest. However, while more cells increase the granularity of the filtering it also increases the overhead of managing the cells and the frequency of nodes joining and leaving groups.

With the other strategy described by Zou et al. (2001), entity-based grouping,
each entity is associated with a group. An entity sends its information to this group and joins the groups of all entities which fall inside its area of interest. To know which groups to join the same structure as used in cell-based grouping is maintained. That is, the state space is still divided into cells and each entity maintains a sending set and a receiving set. But instead of sending state updates to the groups in the sending set, the entity just sends periodic control messages indicating its identity, coordinates, and group. When an entity receives a control message it can decide whether or not the entity sending it falls within its area of interest and hence whether it should join the entity’s group or not. Thus, the filtering can be controlled at a finer level than is possible with cell-based grouping. However, there is an added overhead of sending periodic control messages, but these may be small and transmitted less frequently than information messages. Further, an extra overhead is the extra groups needed for all entities.

If the area consists of rough terrain (e.g. indoor environments) the map may
be divided into cells in a way so that the number of cells that are visible (and/or hearable, etc.) from each cell is minimized. These relationships can be pre-calculated so that each cell stores a list of visible cells. This approach is described by Barrus, Waters & Anderson (1996). For example, in a house each room can be part of a single cell. This approach has the added advantage that it can speed up the graphical rendering at each node by only considering the cells that are visible from the cell that contains the player’s viewpoint.

Statically dividing the map in cells is effective as long as the entities are evenly distributed between the cells. However, if most of the entities are gathered in a few cells, the benefits of grouping are reduced. If the cells are similar in size to the entities’ areas of interest, there is not much that can be done, because the entities should know about each other anyway. However, if the cells are large and the entities receive a lot of messages from entities outside their areas of interest, then a solution is to dynamically divide the cells with high activity into sub-cells and merge cells with little activity (see e.g. Abrams, Watsen & Zyda 1998, Aronson 1997, Morse & Zyda 2001).

2.5 Database concepts

In this section database concepts are defined and explained. In particular, distributed databases are considered since we want to examine their relation to networked real-time multiplayer games.
2.5.1 Distributed and real-time databases

A database is a collection of related data with an inherent meaning (Elmasri & Navathe 2000). A database management system (DBMS) is a collection of programs that enables users and applications to create and maintain a database. A database consists of data objects (also called “data items” or “data elements”). A data object has a type, which describes the structure of the data object.

A distributed database is a database whose data is located at several of the nodes in a distributed system (Gustavsson 2000). Each node that is part of a distributed database has a local database which consists of the data stored at the local node. A data object in a distributed database may be replicated, which means that the data object is physically stored at multiple nodes, but appears to the applications as a single (logical) data object. The physical objects that represent the same data object are called replicas of that data object. In a fully replicated database every data object is replicated on every node in the system. If some parts of the database are replicated, but it is not fully replicated, then the database is partially replicated. Two advantages with replication are that availability and fault tolerance are increased. The availability of a data object increases with the number of replicas, because the likelihood of having access to one of the replicas is increased. The fault tolerance for a data object increases with the number of replicas, because if there are $t$ replicas (at different nodes), then we can normally tolerate that $t - 1$ of these fail (assuming that the nodes are fail-silent, i.e. failing nodes do not transmit any data at all).

A real-time database is a database which is used in a real-time system and must therefore be predictable and sufficiently efficient (Gustavsson 2000). Traditionally,
CHAPTER 2. BACKGROUND

the database is kept on ordinary disks. This is not suitable for a real-time database because of the long access times and the unpredictability of many disk drivers. A solution to this is to store the database in main memory. A database system that stores some or all of its data in main memory is called a main-memory database.

2.5.2 Transactions

The transaction concept provides a mechanism for describing logical units of database processing (Elmasri & Navathe 2000). A transaction is a sequence of read and write operations that are performed as a unit. Normally, a transaction should possess four properties, commonly referred to as the ACID properties. ACID is an acronym for Atomicity, Consistency, Isolation, and Durability. They are defined as follows (Elmasri & Navathe 2000):

Atomicity A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.

Consistency A transaction is consistency preserving if its complete execution takes the database from one consistent state to another (see Section 2.5.3).

Isolation A transaction should appear as though it is being executed in isolation from other transactions. That is, the execution of a transaction should not be interfered with by concurrently executing transactions.

Durability The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.
A DBMS normally has the responsibility of ensuring the atomicity, isolation, and recovery properties. The consistency property must however also be ensured by the application, i.e. transactions should be specified in such a way that this property is ensured.

A transaction makes a series of read and write operations to the database during its execution. A transaction ends by either committing or aborting. If the transaction commits, then the DBMS guarantees that the updates made by the transaction are durable. If the transaction aborts, then all updates made by the transaction are discarded. The ACID properties guarantee that the updates made by a transaction are only made visible to other transactions when the transaction commits and then all updates are visible at once.

2.5.3 Consistency preservation

A database is in a consistent state if the database satisfies the constraints in the database specification as well as any other constraints that should hold on the database (Elmasri & Navathe 2000). As discussed in Section 2.5.2, an important property of a transaction is that it is consistency preserving, i.e. it should be guaranteed that if a database is in a consistent state before executing a transaction, it should be in a consistent state after the transaction has been executed.

In a distributed database, consistency can be divided into internal consistency and mutual consistency (Gustavsson & Andler 2002). A database is internally consistent if it does not violate any database constraints. An example of this may be that the total expenditure should never exceed the total income. The replicas of a database
CHAPTER 2. BACKGROUND

object are *mutually consistent* if they are identical.

**Replication management**

A distributed database that supports replication of data needs a *replication mechanism*, which consists of propagation and integration. *Propagation* ensures that updates of a data object on a node will be sent to all other nodes where that data object is replicated. When a node receives a propagated update, it is *integrated* in the local database.

There are two approaches for preserving mutual consistency in a distributed replicated database (Gustavsson 2000). With *immediate consistency*, updates made by a transaction must be integrated at all replicas before the transaction commits (e.g. by using the two-phase commit protocol, see Elmasri & Navathe 2000).

With *eventual consistency*, transactions are allowed to commit locally, before all replicas of a data object has been updated. Hence, temporary (mutual) inconsistencies are allowed in the database, but the database should eventually reach a consistent state if the system is made quiescent (i.e. if all running transactions are completed and no new transactions are started).

**Conflict management**

On a local node conflicts are normally avoided by using a locking mechanism, where a transaction locks all data objects it operates on until it is completed (i.e. either commits or aborts). However, with eventual consistency, conflicts may occur between transactions operating on different replicas of the same data objects. Conflicts can
be divided into read-write conflicts and write-write conflicts (Gustavsson & Andler 2002). Let \( p_t \) be the set of data objects read by transaction \( T_t \), and \( w_t \) the set of data objects written by transaction \( T_t \). A read-write conflict may occur if there exists a set of transactions \( \{T_1, \ldots, T_m\} \) such that \( w_1 \cap p_2 \neq \emptyset \wedge w_2 \cap p_3 \neq \emptyset \wedge \ldots \wedge w_m \cap p_1 \neq \emptyset \). A write-write conflict occurs when two transactions concurrently update different replicas of the same data object (concurrently in the sense that both replicas are updated before either update has been replicated to the other node).

Because conflicts may occur between transactions at different nodes we need a conflict management method, which consist of a conflict detection mechanism and conflict resolution mechanism (Gustavsson 2000). The Conflict detection mechanism is responsible for detecting a conflict and is normally handled by the DBMS (see e.g. Lundström 1997). When a conflict has been detected it has to be resolved by a conflict resolution mechanism. How a conflict should be resolved is application specific and depends on the database specification.

2.5.4 Active databases

A traditional database is passive, i.e. it just stores data and all updates are done by the applications. In an active database it is possible to specify active behavior directly in the database. This can be achieved by adding a rule mechanism to the database (Eriksson 1998). Different rule mechanisms can have different properties and expressive power. A common type is ECA rules, where the acronym ECA stands for Event, Condition, and Action, respectively. Their semantics are:

**ON event**
CHAPTER 2. BACKGROUND

**IF** condition

**DO** action

That is, the action is executed if the event is triggered and the condition is true. The event part can either be a primitive event or a composite event (an event composed of other events using logical expressions). An event may, for example, be triggered by a database operation, an application, or by the action part of a rule. The condition part can contain conditional queries on data in the database, event parameters, or other variables.
Chapter 3

Problem description

In this chapter, the problem that is investigated in this dissertation is explained. In Section 3.1, we present the type of games that is aimed at and attributes that are considered important for such games. Section 3.2 states the problem and presents the shortcomings with the usual client/server and peer-to-peer architectures. In Section 3.3, the goals of this dissertation are presented.

3.1 Focus

The focus of this dissertation is on large-scale and long-running networked real-time multiplayer games. With *large-scale* we mean that the game state is large and that a game execution involves many (perhaps thousands) players playing simultaneously. With *long-running* we mean a game execution that may run for days or years, where players regularly log on and off, and the virtual environment evolves as a result of the
CHAPTER 3. PROBLEM DESCRIPTION

players’ actions and the passage of time. Games of this type are commonly called “massively multiplayer online games”. We consider three attributes as crucial for this type of games; scalability, security, and fault tolerance.

First, a solution should be **scalable** to a large number of players and entities. The better the scalability, the easier it is to compensate for a growing number of players and entities without compromising responsiveness.

Second, a solution should be **secure**, i.e., have high protection against cheating. Many solutions in the literature do not consider security at all, while in practice this is often one of the most important issues (see e.g. Baughman & Levine 2001, Yan & Choi 2002, Pritchard 2000, Kirmse & Kirmse 1997, Smed et al. 2001). If some players cheat in a game, this can ruin the experience for other players. This is especially important for long-running games, where players may invest much time in developing their entities. In this dissertation, we limit ourself to two types of cheating. The first type of cheating is when a player incorrectly modifies the global game state (i.e. the state that is observed by other nodes). Pritchard (2000) calls this cheat “exploiting authoritative clients”. This can be done in different ways, for example by altering state updates that are sent from the player’s node to other nodes. A cheating player could use this method to, for example, increase some advantageous attributes (like health) of an entity. The second type of cheating is when a player exposes information that should be hidden to the player (Pritchard 2000). For example information about parts of the game state that is outside the player’s area of interest.

Third, a solution should be **fault tolerant** and provide **graceful degradation**. That is, the system should continue to operate in the presence of error, but it is acceptable
with a partial degradation of functionality or performance during failure and recovery (Burns & Wellings 1997). Thus, even if a node crashes, the game execution should be able to continue mostly unaffected (perhaps with lower responsiveness and/or loss of nonessential data).

## 3.2 Problem

The problem that is investigated in this dissertation is how to build large-scale and long-running networked real-time multiplayer games that are scalable, secure, and fault tolerant. The usual approaches for building networked games are with client/server or peer-to-peer architectures, which each fail on one or more of these attributes. The following text highlights these approaches and their shortcomings. First we discuss centralized approaches and then continue with decentralized approaches.

With a centralized approach, a server controls the complete game state and clients hold surrogates of the entities they are interested in. Clients send player actions to the server and the server regularly updates the game state and sends state updates to clients. The server performs filtering to reduce the number and size of messages that are sent to the clients. The scalability of this approach is bad since the server is a bottleneck. All messages travel through the server and all events are handled by the server. This approach supports security and can protect against both types of cheating. First, if the server is secure, players cannot modify the game state. Second, if the server only sends information that the players should know about, then there
is no hidden information for a cheating player to expose. The fault tolerance of this approach is generally bad. Unless the game state is backed up to persistent storage, it is lost in the case of a server crash. Even if the server is able to recover, the game execution is halted during the recovery.

A decentralized approach can be realized with a peer-to-peer architecture or a decentralized client/server architecture. In the first case, nodes send messages directly between each other, while in the second case, all messages are routed through a (message) server. A decentralized approach can either be event-based or state-based (see Section 2.2).

With a decentralized event-based approach, the game state must be fully replicated and player actions are broadcast to all nodes (directly or through a server). The receiving nodes process the actions and update their local game state. All events must be processed in the same order by every node, which can be achieved by forcing nodes to execute synchronously or by using a conflict resolution mechanism like Timewarp (see Section 2.2.1). An event-based approach scales badly to large numbers of entities and nodes, because every event must be received and handled by every node and every node must hold the complete game state. Thus, it is not suitable for large-scale games or games where player actions are frequent. This approach is secure against cheating players trying to incorrectly modify the game state. Nodes can only affect the global game state by sending events and other nodes can detect if a player tries to perform a forbidden event. A player can incorrectly modify the game state at the local node, but that does not affect other players since they manage their own replicas of the game state. However, the approach is not secure
against hidden information exposure, because every node has the full game state and a cheating player can modify the local game engine to expose hidden information. The approach has good fault tolerance, because the game state is fully replicated and no information is lost if a node crashes (although the game execution may be halted until the other nodes realize that a node has crashed).

With a decentralized state-based approach, each node may own some entities (e.g. a player’s node often owns the player’s entities), and other nodes that are interested in any of those entities have surrogates for them. When the owner updates an entity, it sends state updates to the interested nodes. Interest management can be handled by a server or by using multicast and letting the network do the filtering (see Section 2.4.2). This approach may have good scalability, because each node handles a subset of the total game state and only needs to receive information from a subset of the other nodes. The scalability depends on how message distribution is handled. For example, if a server is used as a message router, it is a potential bottleneck. This approach has bad security, because entities are owned by player nodes. Thus, a cheating player can modify the state of these entities and send incorrect state updates to other nodes. Depending on how interest management is handled, it may or may not be secure against hidden information exposure. This approach supports fault tolerance, because when a node disconnects it is only the entities owned by that node that are lost or need to be recovered.
CHAPTER 3. PROBLEM DESCRIPTION

3.3 Goals

As explained in the previous section, there are disadvantages of client/server architectures as well as peer-to-peer architectures. The first goal is to propose an architecture and a framework for building networked real-time multiplayer games that meets the requirements for scalability, security, and fault tolerance. This includes identifying relevant design decisions and discussing their trade-off issues.

The second goal is to investigate how a DBMS can support the proposed architecture, and what requirements are important in a DBMS to be useful for networked real-time multiplayer games.
Chapter 4

A server-network architecture

In this chapter, we propose a server-network architecture for building large-scale and long-running networked real-time multiplayer games. A description of this architecture and motivation for using it is covered in Section 4.1. Section 4.2 describes regions, which are used to achieve scalability. In Section 4.3, interest management for the architecture is discussed. In Section 4.4, the use of proxies are discussed and proposed as a secure alternative to client-to-client communication. In Section 4.5, other projects related to our work are presented and differences are pointed out.

4.1 Introduction

A server-network architecture is a generalization of a client/server architecture, where a cluster of interconnected servers are used instead of a single server (Smed et al. 2001). At any point in time, each client is connected to one of the servers. An
example of a server-network architecture is illustrated in Figure 4.1.

To support security, we propose that the game state and all event handling is managed by the servers. A basic assumption is that servers are secure, i.e. players should not be able to directly read or modify the information stored at the servers. This may require taking a number of security precautions (e.g. using firewalls), but that is outside the scope of this dissertation. We assume that servers are secure and that players cannot directly access the information stored at the servers. Servers should also have secure communication links between them (e.g. by using a dedicated network), so that a server can trust all information sent by other servers.

The game state is partially replicated at each of the servers and the state-based paradigm is used to send information from servers to other servers and clients. Each
entity (including both player-controlled and computer-controlled entities) is owned by exactly one server, and other servers and clients that are interested in the entity hold a surrogate for it.

When a player issues an action, it is sent to the server owning the player’s entity. The receiving server eventually handles the player action and updates the entity’s state. Further, at regular intervals, each server updates the state of owned entities and predict the states of surrogates. When the game state is updated, the servers send state updates to interested clients and servers.

If a client waits to update the player’s entity until the owning server responds with a state update, the local responsiveness may be unsatisfactory. This can be avoided by using client-side prediction (Bernier 2001). When a player issues an action, this information can be used directly by the local client when predicting the state of the player’s entity. The server still owns the entity and the surrogate is updated whenever a state update is received from the server, but the local responsiveness is improved.

Each server manages a separate (geographical) region of the virtual environment (Funkhouser 1996). A server owns all entities that are inside its region, and thus the set of entities owned by a server changes dynamically when entities move between regions. This approach is scalable for two reasons. First, each server only needs to manage a subset of the total game state and if the number of servers is increased, each server has a smaller subset to manage. Second, with the reasonable assumption that most entity interactions occur between entities inside the same region, few server-to-server messages need to be sent (Funkhouser 1996). The cost for increasing the
number of entities and clients is that new servers and network resources must be added to achieve the same level of responsiveness.

Scalability is limited if all servers share a common network. As the size of the game state or the number of servers increases, the total number of messages on the network increases, eventually saturating the network. A solution to this is to have dedicated links between servers so that messages sent between server $A$ and server $B$ do not interfere with messages sent between server $C$ and server $D$. This does not mean that there has to be a link between every server, but only between those that have to communicate, i.e. those that manage regions with possible interregional interactions. A compromise is to have both; dedicated links for high-frequent communication between servers managing nearby regions and a shared network for low-frequent communication between servers managing distant regions.

The architecture supports fault tolerance since the game execution can continue in the case of a node crash. A client crash is trivial since a client does not own any part of the game state. If a server crashes, other servers managing nearby regions can take control of the crashed server’s region. The level of fault tolerance depends on how many replicas exist for each entity. If all servers that have replicas of a particular entity crash simultaneously, then the entity needs to be recovered from stable storage, and affected players have to wait until recovery is finished. Fault tolerance is discussed more in Section 5.7.

A disadvantage with server-network architectures is that, as with client/server architectures, there is an added client-to-client latency, because all information sent by a client must be received and sent by one or more servers before it eventually
reaches other interested clients. In Section 4.4, methods for sending messages that bypass the servers are discussed.

4.2 Regions

As described earlier, the state space is divided into a number of regions, and each server manages one of these regions. A region is a subset of the state space for the entities’ attributes. When an entity’s attributes are members of this subset, the entity is inside the region and hence owned by the server managing that region. It is required that every possible attribute value is inside one region so that an entity is always owned by one server. Regions are only an implementation issue and optimally the regions should not be noticeable to players. From a players point of view, entities should not be treated differently if they are in the same region as the player’s entity or in another region.

For simplicity, the following discussion only considers the state space of entities’ positions in a two-dimensional domain and regions are defined by two-dimensional geographical areas in the virtual environment.

To minimize network requirements, the virtual environment should be partitioned into regions in such a way that possible interactions between them are minimized. In open terrain it may be suitable to simply use regions that are square or hexagonal, stacked in a regular pattern (see Figure 4.2). However, if the environment contains rough terrain, it may be better to use irregular regions and put region borders where interactions between entities are uncommon.
CHAPTER 4. A SERVER-NETWORK ARCHITECTURE

Figure 4.2: Examples of region patterns using (a) squares and (b) hexagons. Hexagons have the advantage of reducing the number of close neighbors (six instead of eight), but may not always be suitable (e.g. if the virtual environment is rectangular, then some hexagons must be trimmed).

When a player moves an entity from one region to another, the state and ownership of that entity is transferred from one server to another. From there on, the client should send player actions to the other server. This can either be detected and handled by the client (if it knows about region borders) or the servers can send a reliable message to the client informing it where to send player actions from there on. Because of network latency, a server that no longer owns an entity may still receive actions for it (sent before the sender was aware of the server transfer). These messages can simply be routed by the receiving server to the owning server.

Transferring entities between servers consumes network resources and may temporarily reduce the responsiveness for players controlling the entities. This is especially apparent when a player moves an entity along the edge between two regions, frequently switching sides, which results in the entity being transferred back and
forth between servers. This effect can be reduced by letting the regions overlap to some extent, which implies that some parts of the virtual environment are covered by more than one region. Entities should stay with their current server as long as they are inside the server’s region (see Figure 4.3). This method increases the distance that an entity must move to switch back and forth between servers and hence reduces the number of server transfers. The cost is that an increased number of state updates are sent between servers having overlapping regions, since a larger number of entities at each server are of interest to the other server.
4.3 Interest management

Interest management (see Section 2.4.2) is performed at two stages. First, there is filtering between servers. A server needs to know about all entities that can be of interest to any entity it owns. Second, there is filtering from servers to clients. Because the servers own the players’ entities, they can precisely determine what parts of the updated game state are of interest to a client and only send that information. Besides reducing bandwidth requirements, this has the added advantage of preventing cheating by exposure of hidden information, because the clients do not receive any hidden information for the players to expose.

To determine if a server should inform another server about a state change we introduce the concept of a server’s area of interest. A server’s area of interest defines which parts of the state space the server is interested in and should receive information about. The server holds a surrogate for each entity that is inside its area of interest, but that is not owned by the server itself. A server’s area of interest should always be a superset of all owned entities’ areas of interest so that it always receives at least all information that is relevant to any of its owned entities. The server’s area of interest could either be static (large enough to cover the largest area of interest for any entity that can possibly be inside the region) or dynamic (depending on the entities currently inside the region). A disadvantage with a static area of interest is that a server may receive irrelevant state updates from entities that are inside the server’s area of interest, while they are not inside the area of interest of any of the entities owned by the server. With a dynamic area of interest, a disadvantage is
that the other servers must regularly be informed about changes to a server’s area of interest so that they can determine if a particular state update is of interest to the server or not.

So far, we have only considered filtering for determining whether a node should be informed about an entity’s state updates. This can be generalized to consider how important it is to have updated information about an entity, which we call the level of interest. At the lowest level of interest, there is no interest in the entity at all, while at the highest level of interest, information should be sent about every update to the entity’s state. For example, the distance between an entity and a player’s entity may decide the player’s level of interest for that entity, and hence at which rate state updates about that entity should be sent to the player’s client. As the level of interest for an entity increases, state updates are sent more often to reduce the degree of inconsistency between the entity and its surrogate.

If dead reckoning is used, the threshold that determines when state updates are sent for an entity should decrease as the level of interest for that entity increases. A server’s level of interest for an entity should be equal to or greater than the maximum level of interest for that entity by any of the server’s owned entities.

If network latencies are ignored, then servers can do exact visibility calculations and send only information that is visible to each player. However, because of network latency this may result in entities that suddenly “appear from nowhere” from a player’s point of view. To tolerate network latency and avoid such problems, the areas of interest (both for servers and individual entities) can be overestimated (Funkhouser 1995a). For example, in a FPS game a player cannot usually see what
happens behind the entity’s back. However, it can still be important to send information about what happens behind the player’s entity, because it can turn very fast and it is often important that the player gets correct information in these situations.

Another reason for overestimating areas of interest is to simplify the computations done when determining if something is inside or outside an area of interest. For example, (Funkhouser 1995a) propose stored pre-computed cell-to-cell visibility masks at the servers that are used to determine whether an entity is inside an area of interest or not.

A disadvantage with over-estimating the areas of interest is that some unnecessary state updates are sent by servers and received and filtered by clients. This can be exploited by a cheater for exposing (limited) hidden information at the client.

4.3.1 Optimizations with multicast

If multicast capabilities are present between servers, it can be used to reduce the number of messages sent between them. One approach is to associate each entity with a multicast group (similar to entity-based grouping, see Section 2.4.2). All servers that are interested in an entity joins the associated multicast group and when the state of the entity is updated, a state update is sent to the multicast group. Disadvantages with this approach is that it requires many multicast groups (one for each entity), which may be unfeasible for a large-scale game, and that multicast groups may have to be joined and left relatively frequently (when an entity crosses a server’s area of interest).

Another approach is to divide each region into a number of cells and associate
each cell with a multicast group (similar to cell-based grouping, see Section 2.4.2). When an entity is updated, a state update is sent to the multicast group of the cell where that entity resides. Servers join the groups associated with cells that they are interested in (except for cells inside their own region). A server is interested in a cell if it is interested in any part of the cell. Cells do not have to be the same size. Some parts of the regions may not even have to be covered by cells, because they are never of interest to other servers.

If static areas of interest are used for servers, then each server only has to join the corresponding multicast groups once, when the game execution starts. If dynamic areas of interest are used instead, then multicast groups are joined and left when the server’s area of interest is changed (i.e. when entities owned by the server moves). With static areas of interest we avoid the overhead of having to dynamically join and leave multicast groups, but some unnecessary information is sent.

If multicast capabilities are present from servers to clients, it can be used to reduce the number of sent messages. A server can determine which cells intersect with a client’s area of interest, and add the client to the associated multicast groups. When the client’s area of interest changes, cells are joined and left accordingly. Thus, the client receives information about the cells that intersect with its area of interest. With this approach, clients receive some information from outside its areas of interest (how much depends on the size of the cells). This may be exploited by a cheating player for exposing (limited) hidden information. Also, for security reasons, it is necessary that only servers can add and remove clients from the multicast groups. Otherwise a cheating player may join groups outside its client’s area of interest to
Levels of interest can be implemented by having multiple multicast groups for each cell, where each multicast group represents one level of interest. The server that manages a cell sends state updates to the different multicast groups with a frequency that corresponds to their levels of interest. A node joins the multicast group for each cell that represents the node’s level of interest for the cell. The number of multicast groups that is used for each cell is a trade-off between the number of messages sent by the server managing the cell and the number of messages received by servers having an interest in the cell.

### 4.4 Proxies

This sections starts with a discussion about advantages and disadvantages of having direct client-to-client communication. This is followed by a discussion on how proxies can be used as a secure alternative to client-to-client communication.

While the servers manage the game state and handle all events, some information can still be sent directly between clients. For example, chat messages or other interactions between a group of players, that do not affect the game state, can be sent directly between clients. This reduces the load on the servers and may also reduce the latency for such information. Further, when a player issues an action, it can be sent directly to other clients that are known to be interested in the player’s
entity, and they can use it for better predictions of the entity’s state (i.e. client- side prediction, see Section 4.1). Thus, assuming that the client-to-client latency is lower than the time it takes from that a player action is sent to the servers until another client receives the state update from the servers, remote responsiveness can be increased. However, if clients are able to send player actions between each other, a cheating player can modify the game engine to send false actions to other clients (while sending the correct actions to the server). The remote clients will then base their predictions on wrong information, which may result in jerky and incorrect movement of the surrogate. Further, for clients to communicate, they need the network address to each other. This can be exploited by a cheating player through flooding another player’s client with messages to reduce that player’s responsiveness and hence chances of succeeding in the game.

Summarizing, client-to-client communication can be used to reduce the load on servers and increase remote responsiveness, but introduces some new opportunities for players to cheat.

Mauve, Fischer & Widmer (2002) propose adding extra nodes, called proxies, between the clients and servers. Such proxies may be geographically distributed and a client can connect to the closest or least loaded proxy instead of a server. Each client may be statically connected to a proxy, while the proxy routes all messages between the servers and the client. Proxies can usually be trusted (because they are not under control of the players) and thus some server functionality can be delegated to the proxies.

Proxies can be used as a secure alternative to client-to-client communication. For
example, chat messages that are sent from one player to another can be routed by the proxies without involving the servers. Further, when a proxy receives a player action, it sends it to the servers and may at the same time send it to clients that are interested in the information to reduce remote responsiveness. Cheating is prevented since clients will receive the same player actions as the servers.

According to Mauve et al. (2002), proxies can help to provide congestion control, enhance robustness, minimize the impact of network latency, improve fairness, and protect against cheating. Frécon & Stenius (1998) suggest using proxies as message replicators as an alternative to using multicast. Thus, a server can send a message to a proxy, which in turn replicates the message to interested clients and other proxies.

4.5 Related work

In this section, two approaches with similarities to the approach proposed earlier in this chapter are presented. In the following discussion we refer to the approach proposed in this chapter as our approach.

4.5.1 The mirrored-server architecture

Cronin, Filstrup & Kurc (2001) propose an architecture called the mirrored-server architecture. As with our approach, the game is executed at multiple servers and clients connect to these servers to participate in the game execution. Clients send player actions to the servers, which regularly process the events and send state updates to the clients. However, unlike our approach, each server manages a complete
replica of the game state and an event-based communication paradigm (see Section 2.2.1) is used between servers. Thus, when a server receives an action from a client, it is sent to all other servers, which update their own replicas of the game state. Clients are connected to the same server during the game execution. The authors propose that the servers should be topologically distributed across the Internet and that clients should connect to the closest server. Between servers there should preferably be a fast network connection (e.g. by using a private network).

The servers in the mirrored-server architecture execute asynchronously and an optimistic conflict resolution mechanism called \textit{trailing state synchronization} is proposed, which is similar to Timewarp (see Section 2.2.1). When an event is received late, a rollback is performed and the events (including the newly arrived event) are replayed.

This approach is both secure and fault tolerant for the same reasons as our approach. It is secure, because the game state is managed by the servers and clients can only affect the game stat by sending player actions (this assumes that the servers are secure and that a players cannot gain access to the game state in any way). It is also fault tolerant since the game state is fully replicated at multiple servers. Thus, if a server crashes, the clients formerly connected to that server can simply connect to another server. For the same number of servers, this approach has higher fault tolerance than our approach since for \( n \) servers, \( n - 1 \) can crash without having to recover any part of the game state. However, the high level of fault tolerance comes at the cost of scalability. As explained in Section 2.2.1, an event-based approach is not scalable, because every event must be received and handled by every server.
 CHAPTER 4. A SERVER-NETWORK ARCHITECTURE

The computational load and the bandwidth consumption at the servers will quickly rise when the number of players and entities grows. Increasing the number of server will only make the situation worse, since then even more nodes need to receive the events. Thus, this architecture is not suitable for large-scale networked real-time multiplayer games.

4.5.2 RING

Funkhouser (1995b) describes a system called RING where multiple servers are used to manage communication between clients. The state-based communication paradigm is used to share information between nodes. Unlike our approach, entities are owned by clients. Each entity is owned by exactly one client and that client handles events affecting the entity and sends state updates to inform other nodes about the entity’s state. Clients also maintain surrogates of entities that they are interested in but do not own themselves. Clients do not send messages directly to other clients, but instead send them to servers that forward them to other interested clients and servers. The servers may filter and possibly alter the messages that are routed through them. The aim of this design is to shift some of the processing burden away from the clients and into the servers so that large networked virtual environments can be built using primarily low-cost clients.

Funkhouser (1996) proposes that servers should manage separate regions of the virtual environment and clients should send messages to the server that manages the region where the entity resides. This is similar to our approach with the exception that entities are owned by clients. Funkhouser (1996) experimentally compares this
approach with an approach where no regions are used and clients are connected
statically to the same server during the complete execution. As expected, the region-
based approach, where clients are dynamically connected to different servers, results
in far fewer messages being sent between servers since most interactions happen
between entities in the same region.

When regions are used, Funkhouser’s approach is scalable for the same reasons
as our approach. It may even need less server capacity than our approach since
more computation is done by the clients and less information probably needs to
be sent from clients to servers. This approach also supports fault tolerance. If a
client crashes the entities owned by that client may disappear, but the rest of the
game execution can continue. If a server crashes, other servers may temporarily take
control over the crashed server’s region, and no parts of the game state are lost since
it is owned by the clients. However, the problem with this approach is that it is
not secure. Because a client may own some entities, a player can modify the local
game engine to have full control of the state of those entities. A reason that security
is not considered in this work may be that RING is not considered specifically for
games, but for networked virtual environments in general, where security may not
be considered as important as for games.
Chapter 5

Distributed database support

In this chapter, we investigate how concepts in distributed databases can be applied to networked real-time multiplayer games and the feasibility of using a distributed database management system (DBMS) as a communication medium for this type of games. Further, we consider if certain traditional database properties can be relaxed in favor of performance.

Section 5.1 motivates using a DBMS for this type of games and presents basic requirements of such a system. In Section 5.2, the consistency requirements of real-time games are discussed. In Section 5.3, we consider some conflicts that can occur in this type of games and how they can be resolved. Sections 5.4 and 5.5 describe how interest management and dead reckoning can be implemented when using a DBMS. In Section 5.6, it is explained why traditional transactions may not satisfy the requirements of real-time games and alternatives are presented. Section 5.7 describes how fault tolerance can be supported when using a distributed DBMS.
CHAPTER 5. DISTRIBUTED DATABASE SUPPORT

5.1 Introduction

With the architecture described in Chapter 4, the database holding the game state is distributed over the servers. While the servers collectively hold the complete game state, each individual server holds only a part of it. Each entity is owned by one server and possibly replicated at servers managing nearby regions. Hence, we have a distributed, partially replicated, database.

Using a DBMS for networked real-time multiplayer games can have a number of advantages. It may reduce the development cost and increase maintainability, because the developers can rely on the functionality provided by the DBMS (such as transaction processing and recovery mechanisms) instead of having to implement the same functionality themselves. A distributed DBMS can be viewed as a communication abstraction for the application as it automatically handles the propagation and integration of data. Applications at different nodes do not have to explicitly send messages between each other, but can instead use replicated data objects to distribute information among them. A disadvantage with a DBMS is that it may add extra overhead, both in space and time, because it may have features and give guarantees that are not needed by a game, but still consume resources.

For a DBMS to be feasible to use with networked real-time multiplayer games it is important that it does not incur a large overhead in space and time. To meet the high level of efficiency required by this type of games, a main-memory database is suitable to avoid the large overhead involved with ordinary disks (see Section 2.5.1). While it may not be necessary to store all data in main memory, it is important for
high-frequency data, like positions of entities.

For the same reason, it may not be suitable to require immediate consistency of distributed data, because it introduces overhead in the form of synchronization protocols which may result in long delays (Gustavsson 2000). The alternative is to use an eventual consistency approach, where transactions are allowed to commit locally before all replicas have been updated (see Section 2.5.3). Eventual consistency does not have the same synchronization overhead as immediate consistency, but it introduces the possibility of temporary inconsistencies.

5.2 Consistency requirements

Real-time games are different from most traditional database applications. Responsiveness is often more important than correctness. With a state-based communication paradigm, it is not necessary that every player has the same perfect view of the game state (see Section 2.2.2). Requiring this would imply low responsiveness and bad scalability. Some information may have a low level of interest to a player and should hence have a low priority of being updated.

As a result, some decisions made by a player are based on information that is out-of-date. Hence, a decision that is correct from one player’s point of view may not be correct from another player’s point of view. How this should be handled depends on the type of game and other design decisions. For example, consider a FPS game with two players, A and B, having instant-hit weapons (i.e. the time it takes from the weapon is fired until the bullet strikes the target is considered to be zero). If
player A aims correctly at player B and pulls the trigger, player A would expect a hit. However, based on the information shown to player B, this may not be the case. For example, at the time player B observes that player A fires the weapon, player B’s position may already be safely behind a wall. While the servers hold the correct game state, it may not be the best solution, from the players’ points of view, to use this information for all decisions. It may, for example, be better that the servers make decisions based on what they assume the players observe. In the example given above it may be best to make the decision based on what A sees, since it is probably more annoying for a player to not hit a target when aiming correctly, than for a player to get hit without being in the shooting player’s direct aim (which is probably hard to recognize for the player being shot at).

5.3 Conflict resolution

As described in Section 2.5.3, conflict resolution is application specific. In this section we consider some conflicts that can occur in networked real-time multiplayer games when using eventual consistency and how they can be resolved.

As explained before, write-write conflicts occurs when two (or more) nodes update the same (logical) data object. These conflicts can be prevented by using master replication (Gray, Helland, O’Neil & Shasha 1996, Breitbart & Korth 1997). With master replication each data object has a master node. The master node holds the primary copy of the data object, which means that it is the only node that is allowed to update the data object. When the master node updates the primary copy, the
update is replicated to the other nodes holding replicas of the data object. The other
nodes have read-only access to the data object and if they need to update the data
object, they have to send a request to the master node to do the update on their
behalf. Since there is only one node that can update the data object, write-write
conflicts between nodes cannot occur. Thus, the overhead of detecting and resolving
write-write conflicts are avoided.

Master replication is well suited for our architecture, because every entity is
owned by a single server. Thus, a server is assigned the role of master node for the
entities it owns. It is reasonable to assume that the majority of updates to an entity
is initiated by the server that owns it, since it is the owning server that receives
actions from the player and updates the entity accordingly. For those (rare) cases
when a server needs to update an entity owned by another server, a request is made
to that server. This adds some overhead, but it should be a small part of the total
number of updates.

If master replication is used, then it must be possible to change an entity’s master
node dynamically as the entity’s ownership is transferred between servers. When
an entity leaves a server’s region, that server informs the new owner about it and
transfers the entity’s data. When the new owner receives this information, it informs
the client controlling the transferred entity, and possibly other servers, about it.
Because of network latency, the other nodes will not be informed about this change
of ownership directly, and some events and update requests affecting the entity’s
state may still be sent to the previous owner. The previous owner should therefore
be prepared to receive such messages, buffer them until the transfer is finished, and
then forward them to the new owner.

Read-write conflicts can still occur, even if master replication is used. An example is when two entities, $A$ and $B$, that are owned by different servers may collide. The server that owns $A$ has to read the state of both $A$ and $B$ to determine if a collision occurs, and if it occurs, $A$’s state is updated. For $B$’s owner the situation is the reverse. Hence, there is a dependency between the transactions and a read-write conflict may occur. For this particular case, most games may be able to ignore read-write conflicts since collision handling does not normally need to be totally correct (since surrogates are used in the calculations, the result will not be correct anyway). However, it may not be possible to prevent or ignore all conflicts for all types of data objects and transactions in a game. For some types, it may still be necessary to detect and resolve read-write conflicts, and for other types, master replication may not be suitable (e.g. if we must update entities owned by different servers in a single transaction). Hence a DBMS suitable for games and our architecture should allow the developers to specify the policy for resolving read-write conflicts for different transactions and data objects, and if a data object should be updatable by just one master node or by multiple nodes.

5.4 Interest management

As described in Section 2.4.2, interest management is used to avoid sending information that is irrelevant to the receiver. An entity’s state updates are only of interest to a subset of the nodes, and in a distributed database, the entity needs to be replicated
only to that subset of nodes. This subset of nodes changes dynamically as the game progress. The replication of entities could either be handled individually for each entity or for groups of entities. If each entity is handled individually, the servers have to regularly check each entity’s area of interest to decide which nodes should have a replica of the entity. This may introduce considerable overhead depending on how expensive it is for the DBMS to add and remove replicas.

Another possibility is to divide each region into a number of cells and replicate all entities inside a cell collectively (similarly to the approach described in Section 4.3.1). In a DBMS, this may be supported by segments. A segment is a container of objects, where properties of the container are shared between all contained objects (Mathiason 2002). This allows the specification of the subset of nodes to which a segment (and hence its contained objects) is replicated. Each cell should be associated with a segment and this segment should be replicated to all servers that are interested in that cell. It is required that segments allow dynamic insertion and deletion of objects, since entities dynamically move between cells. However, the set of nodes that a segment is replicated on can be static if the servers’ areas of interest are static (see Section 4.3). Depending on how segment replication is implemented in the DBMS, this may reduce the overhead compared to individual replication.

5.5 Dead reckoning

In this section, it is described how dead reckoning (see Section 2.4.1) can be implemented in a database. There should be two data objects for each entity. The first
data object represents the entity and is only stored at the owning node (i.e. it is not replicated). The second data object represents the surrogate (and ghost) and is replicated at all nodes that are interested in the entity (including the owner). The owner updates the entity for each update interval. When the entity and the surrogate prediction diverges by more than some specified threshold, the surrogate is updated and the DBMS replicates this update to all other interested nodes automatically. The nodes use the surrogate to predict the entity’s state for each update interval.

If the DBMS supports active behavior (see Section 2.5.4), then it may be possible to specify the update of the surrogate to be done automatically by the database. In ECA rules this has the following form:

\[
\text{ON update to entity}
\]

\[
\text{IF difference between entity and surrogate prediction exceeds threshold}
\]

\[
\text{DO update surrogate}
\]

The surrogate prediction is a derived data object. It is derived by extrapolation from the surrogate using the current wall-clock time. This can be implemented directly in the DBMS if it supports derived data objects that are recalculated every time they are used, based on the current wall-clock time.

Different levels of interest (see Section 4.3) for an entity can be supported by having multiple surrogates for that entity, one for each level of interest. Each surrogate has a different threshold that regulates when it is updated, depending on its level of interest. A node that owns an entity has the primary copies of all these surrogates. Each other node that is interested in this entity has a replica of one of the surrogates, i.e. the one that corresponds to that particular node’s level of interest for the entity.
An example of this is given in Figure 5.1.

5.6 Reducing transaction overhead

Real-time games often require frequent access to the game state. For each update interval the states of entities are updated and predictions are done for surrogates. Interactions (such as collisions) between entities and surrogates are handled and the state is updated accordingly. During these update intervals a large part of a server’s local database is queried and updated. If this is done within a normal transaction that fully adheres to the isolation property, considerable overhead may be introduced from concurrency control. One way to avoid this is to only use non-
preemptive transactions, which can run from start to completion without interference from other transactions. This means that no concurrency control is needed and thus the transaction overhead is reduced. Generally, such approaches may reduce the concurrency of the system, but in this case it is probably not a problem since the transactions must execute in a short time interval. Another possibility is to use transactions that relax the isolation property. According to Gray & Reuter (1993), many systems allow transactions with different degrees of isolation, which allows a compromise between correctness and performance. Any of these alternatives may be suitable for networked real-time multiplayer games as long as the overhead from concurrency control is kept to a minimum.

5.7 Fault tolerance

A fault tolerant DBMS should be able to recover from a crash, without losing data. One of the advantages of having a distributed database is the high fault tolerance made possible by having replicated data. If data is replicated, it does not become unavailable when a node in the system crashes, since it can be retrieved from any of the other nodes containing replicated copies of that data (Leifsson 1999).

With our architecture, a crash can occur at a client or a server. A client crash (or disconnect) is not problematic because the complete game state is held by the servers. The server owning the client’s entities can simply store these entities until the client reconnects. However, a server crash may be problematic, because a server owns part of the game state. If dynamic regions are allowed, a server crash can
be handled by having servers managing nearby regions take control of the crashed 
server’s region and entities until the crashed server has recovered (assuming that 
servers can detect when another server has crashed). It requires that the neighbor 
servers hold surrogates of all entities owned by the crashed server and that the 
surrogates contain enough information for recreating the entities. By splitting the 
region and recreating the entities from their surrogates, the game can continue mostly 
unaffected. By using the surrogates for recovery, the affected entities will be slightly 
out-of-date, but this should be acceptable for most entities in the (rare) event of a 
server crash.

Using replication for recovery may not always be possible. For example, the 
entities in some parts of a server’s region may not be replicated or servers controlling 
neighboring regions may crash at the same time. To be able to recover from a crash 
under these circumstances some form of logging to stable storage must be used (see 
e.g. Leifsson 1999). However, it may not be worth the overhead of logging every 
update. As discussed earlier, it is often acceptable to recover from slightly out-of-
date data. Whether or not this is sufficient depends on the type of data. Transactions 
updating entities’ positions do not normally have to be logged every time, but can 
instead be logged periodically. Other transactions, like players trading objects, may 
require guaranteed persistence and should always be logged.
Chapter 6

Conclusions

This chapter summarizes contributions of the dissertation and presents future work. In Section 6.1, a brief summary of the contributions are presented. Section 6.2 continues by discussing the results. In Section 6.3, suggestions for future work are presented.

6.1 Contributions

The following is a summary of the contributions.

- We survey the field of networked real-time multiplayer games.
  - Basic concepts in this relatively new and unestablished field of research are defined.
  - Two different communication paradigms (event-based and state-based) for sharing information between nodes are identified.
– The two most common architectures (client/server and peer-to-peer) and their advantages and disadvantages are discussed.
– Methods for reducing resource requirements are discussed.
– Database concepts are defined and explained.

• We propose use of a server-network architecture for large-scale and long-running networked real-time multiplayer games.

– Scalability, security, and fault tolerance are shown to be supported by the architecture.
– Regions are explained and overlapping regions are proposed as a method to reduce server transfers.
– Interest management for the architecture is discussed and possible optimizations using multicast are presented.
– Proxies are presented as a secure alternative to client-to-client communication.

• We explore how a distributed database can support the server-network architecture.

– Advantages and disadvantages of using a distributed DBMS are discussed.
– Methods for implementing dead reckoning and interest management are presented.
– Master replication is discussed as a method to avoid write-write conflicts.
– Methods for reducing transaction overhead are presented.

6.2 Discussion

In Chapter 4, the server-network architecture is proposed as an architecture for building large-scale and long-running networked real-time multiplayer games. It supports scalability, security, and fault tolerance, which we consider to be important attributes for this type of games.

When discussing the scalability issues of the server-network architecture, we have assumed that entities are uniformly distributed in the virtual environment. If some areas have a higher average number of entities than other areas, this can be balanced by having differently sized regions. Generally, if all servers have the same capabilities, then they should control regions that on average contain the same number of entities (assuming that all entities are equally expensive to own). However, sometimes a large number of player entities may temporarily gather in an area of the virtual environment. This may result in the server or servers controlling that area becoming heavily loaded with decreased responsiveness as a consequence. A possible solution to this problem is to allow region borders to change dynamically as a form of load balancing. That is, a server controlling a region with a small number of entities can have its region subsume parts of another region having a large number of entities and transfer ownership of the entities in the affected area. However, if a large number of entities are gathered in an area that is as small as the entities’ areas of interest, this may not solve the problem. If such a heavily occupied area is managed by multiple
servers, then a large number of server-to-server messages are needed to handle all interactions between entities. A drastic way to solve this is to simply not allow too many entities to gather in a small enough area. However, then another problem is how this restriction should manifest itself to the players.

We argue that the server-network architecture is secure, because the game state is controlled by secure servers. How to make the servers secure (e.g. using firewalls) is outside the scope of this dissertation. It is important that clients cannot directly modify the game state, because this can be exploited by cheating players. Generally, the only information that servers should accept from clients are player actions (and possibly some player information when the client connects and disconnects). Thus, the clients can only affect the game state indirectly through player actions that are processed by the servers. If a forbidden command (e.g. shooting when no ammunition is left) is issued, the servers can simply ignore it. However, there are other forms of cheating that we have not considered. For example, a cheater may use a program that automatically calculates the optimal actions and sends them to the server. This is common in FPS games where programs that help the players to aim are commonly used (Pritchard 2000). One can argue that this is not cheating, because the actions that are sent are perfectly legal and do not break any rules of the game. However, it may ruin the experience for other players, and some games try to detect when it is used (e.g. when a players has an aim that is just too good to be true, although this is of course not a reliable solution).

Chapter 5 presents some requirements that are necessary for a DBMS to be suitable for games. It should support partial replication so that a surrogate can be
replicated at nodes that are interested in the entity. It should also support eventual consistency to avoid the cost of distributed concurrency control. Further, the DBMS should support main-memory resident data to avoid the overhead of using ordinary disks. Master replication is proposed as a method to prevent write-write conflicts and the overhead associated with resolving such conflicts. Also, since each server accesses a large part of the local database during the regular update intervals, we argue that conventional transactions introduce too much overhead from concurrency control. To reduce this overhead we suggest the use of either non-preemptive transactions or transactions with a lower degree of isolation. Chapter 5 also contains suggestions for how dead reckoning and interest management can be handled in a database. Many of the database methods discussed in that chapter are suitable not only for the server-network architecture, but can be applied in other architectures as well.

6.3 Future work

In the following subsections, suggestions for future work are presented.

6.3.1 Validation of server-network architecture

In order to validate that the server-network architecture proposed in this dissertation achieves the desired results, an implementation of it could be made. For this purpose, it would be sufficient to use a very simple game simulation, e.g., having a two-dimensional virtual environment where each entity’s state consists of just a position attribute. An entity’s areas of interest could be defined as a circular area, centered
at the entity’s position with a fixed radius. The set of actions that a player can issue could also be very limited, e.g., increase/decrease velocity and turn right/left. Players can be simulated by having client nodes produce random actions at random points in time.

Instead of using an actual node for each client and server, multiple (virtual) nodes can be simulated on a single physical node. To simulate the network traffic between nodes, a network simulator can be used, and specified limitations on bandwidth and latency can be put on the links between the virtual nodes. A network simulator may also make it easier to analyze the network traffic for different scenarios and network configurations. For example, it would be easy to compare differences between having a shared network between servers, and having a dedicated link between each pair of neighboring servers (see Section 4.1). Further, it would be possible to add virtual proxy nodes (see Section 4.4) to the simulation, and see how they affect the network traffic. Experiments can also be made to find suitable sizes of overlap between regions and cells (if grouping is used), and analyze the different trade-off issues that are discussed in this dissertation.

6.3.2 Database support

To validate the results in Chapter 5, an existing DBMS could be used in an implementation. As described earlier, a suitable DBMS should support eventual consistency and main-memory resident data. An example of a DBMS that supports these properties is DeeDS (Andler, Hansson, Eriksson, Mellin, Berndtsson & Eftring 1996). It would be interesting to use an existing DBMS to analyze how it affects the perfor-
mance, and to find potential performance bottlenecks.

One aspect that needs to be analyzed further is how conflicts between transactions at different nodes should be handled. We propose the use of master replication (see Section 5.3) to prevent write-write conflicts. However, this may not always be the best approach and may not always be possible. It would help to have a classification for which types of entities and transactions this is the case. Read-write conflicts may still occur even if master replication is used. How these conflicts should be resolved may differ depending on the type of entities and transactions, which also needs to be analyzed further.

Active database support is only briefly touched upon in this dissertation. It may be possible to use active database support (e.g. ECA-rules, see Section 2.5.4) to simplify some game mechanisms that otherwise must be implemented in the applications (e.g. interest management).

When database support was discussed in Chapter 5, we only considered the database to be shared among the servers. However, as most of the data is sent to the clients, a natural extension is to let the database be shared with the clients as well. That is, the DBMS could replicate data objects directly to the clients so that message passing can be hidden from the applications. Thus, the entities that are of interest to a client are replicated by the DBMS to that client. This requires that the DBMS supports high security restrictions and that replicas residing at the clients should have (restricted) read-only access.
6.3.3 Maintainability

An important attribute that has not been considered in this dissertation is maintainability, i.e. how easy is it to do maintenance of the game. This is especially complicated for long-running networked real-time multiplayer games, since players generally assume continuous game execution. Temporarily stopping the game for maintenance should be avoided if possible. Instead, on-line maintenance, i.e. maintenance during the game execution, is preferred. Much care must be taken in designing the game to allow this type of maintenance. For example, it should allow insertion, removal, and modification of entities and possibly some game rules during the game execution. This may be a strong argument for using a DBMS, since support for transaction processing simplifies this kind of tasks. Another maintainability issue is how to handle software updates. The software may have to be updated at both clients and servers. One possibility for updating the server software is to update one server at a time, and let neighboring servers temporarily take control of the maintained server’s region (as if the server had crashed).
Acknowledgements

I would like to thank my supervisor Sanny Gustavsson and my examiner Prof. Sten F. Andler for their helpful comments and inspiring support during this project. I would also like to thank the rest of the Distributed Real-Time Systems research group for their valuable feedback. Thanks also to Kim Laurio for reviewing this dissertation and giving a lot of useful suggestions for improvements.
Bibliography


BIBLIOGRAPHY


80


List of figures

2.1  Timewarp .................................................. 12
2.2  Prediction by extrapolation .............................. 15
2.3  Prediction by interpolation .............................. 17
2.4  Client/server architecture ............................... 17
2.5  Peer-to-peer architecture ............................... 20
2.6  Dead reckoning ........................................... 23
2.7  Areas of interest .......................................... 25
2.8  Cell-based grouping ....................................... 28
4.1  Server-network architecture ............................. 43
4.2  Region patterns ........................................... 47
4.3  Overlapping regions ....................................... 48
5.1  Dead reckoning with two levels of interest .............. 67