TKN Telecommunication Networks Group

Technische Universität Berlin
Telecommunication Networks Group

Consistency in Distributed Systems

Sven Grottke Andreas Köpke Jan Sablatnig
Jiehua Chen Ruedi Seiler Adam Wolisz

Berlin, February 8, 2008

TKN Technical Report TKN-08-005

TKN Technical Reports Series

Editor: Prof. Dr.-Ing. Adam Wolisz

Abstract

In this paper, we deal with the problem of consistency in distributed systems where hosts have to react to a events in a time span which is less than the network latency. Distributed virtual environments and wireless sensor networks are two typical examples of such systems. In such systems, hosts routinely act while the system is in in inconsistent state in order to provide short reaction times. This inconsistency has a noticeable effect on the perceived quality of these actions and their effect on the application.

Numerous solutions have been developed which try to keep a system consistent, responsive and scalable in terms of resource consumption. Typically, there's a tradeoff between these three aspects. While reaction times and resource consumption are well understood, there exists no commonly agreed method to measure the level of inconsistency. As a consequence, it is nearly impossible to compare different solutions to the consistency problem with each other.

To overcome these difficulties, we plan to provide a framework where different consistency algorithms can be compared with each other. In this paper, we describe the typical challenges of a distributed system which requires fast reactions, and the common approaches to meet those challenges. We also define various measures for inconsistency and cost which can be used to evaluate different consistency algorithms. Finally, we give an overview of important existing solutions which we categorize by their choice of consistency algorithm and how they achieve scalability.

Contents

Chapter I
Introduction

During the last thirty years, distributed systems have been investigated and used in a variety fields such as databases, virtual environments, parallel computing, information theory, mobile communications, sensor networks and many more. Typically, development is application-driven, i.e. solutions are custom-made for one specific application at a time. This makes it hard to compare existing solutions to each other, or to apply them to a new problem. Furthermore, publications on those solutions often use different terminology, which makes it even harder to compare them even when they are from the same field of applications.

To improve this situation, we plan to create a framework where applications and solutions can be compared to each other and to existing benchmarks. To the extent of our knowledge, no such framework or common benchmark exists at the moment. As the field of distributed systems is too large and diverse to provide a universal solution, we focus on two specific fields of applications, namely distributed virtual environments (DVE) and wireless sensor networks (WSN).

We identify three critical design aspects of such systems:

1. **Consistency**: Hosts have to make decisions based on data which can be modified by other hosts. To prevent wrong decisions, the system has to make sure that the data is consistent between hosts, and modifications on one host are propagated to other hosts.

2. **Responsiveness**: Hosts have to react to events such as user input in a timely manner. In DVEs and WSN, the time available for such an reaction is typically less than the network latency. As a result, hosts acting while the system is in an inherently inconsistent state are the rule, rather than the exception.

3. **Scalability**: Distributed systems are growing in size. The typical number of concurrent hosts in a DVE is in the range of thousands, and sensor networks with a couple of hundreds of nodes are becoming more common. At the same time, network capacity and computing power are limited, which means that an increase in the number of hosts can easily overload the system.

In this paper, we describe the principle layout of a typical DVE or WSN, and the most important classes of consistency algorithms used in such systems. As we will show, each of the approaches to the consistency problem involves a tradeoff between the three design aspects of consistency, responsiveness and scalability. To make this tradeoff quantifiable, we propose several measures for the level of inconsistency, and the various cost factors such as network load, memory footprint, CPU load and in the case of WNSs, energy consumption. We give an overview of the methods used by several important standards, research platforms and commercial applications, and discuss their specific solutions to the problems consistency and scalability.

1 General Setup

The typical setup of a distributed system in our fields of interest is as follows: the system consists a number of hosts, i.e. electronic computers, which communicate over a network, e.g. the Internet or a wireless network. Each host is running the application software. There is a set of *shared* variables, i.e. variables which have a value which is replicated on each host. For the application to run correctly, it is important that all hosts agree on the value of each shared variable. When an event occurs on a host which changes the value of a variable, the change has to be propagated to the other hosts, or there will no longer be agreement on the value. In addition, the host where the event occurs has to react to the event within a specific time span. In the systems which we are interested in, the acceptable reaction time is typically *lower* than the time required to propagate the event to all other hosts. In other words, a host cannot delay its reaction to a local event until the system has reached agreement on the value of all shared variables.

Two examples will help to understand the requirements of typical applications in DVEs and WSNs, respectively:

The first application is a virtual soccer game. There, human users are participating using typical modern desktop computers. They use a mouse or the keyboard to control a virtual representation of themselves — their avatar — in a 3D visualization of a soccer stadium. Participants can take on different roles such as player, referee or spectator. The computers are connected via the Internet, which means their network characteristics such as bandwidth, latency and loss rate may vary widely. Typical system sizes may range from a few dozen players up to hundreds of thousands of participants.

In order to play the same game together, participants require consistent representations of the stadium and the objects within, such as the ball and other players. The degree of consistency a participant requires for a specific object may depend on the players role and the object in question. For example, spectators would need only a rough approximation of the spectators on the other side of the stadium, but a player would require the precise location of the ball to kick it into a certain direction. Furthermore, player actions such as kicking the ball have to have an immediate effect (usually in less than 150 MB), or the impression of immersion in the virtual world will be lost [Arm03]. This means that actions have to be processed at the originating computer before they have been received by the other participants, as the network delay will often be greater than 150 ms.

Our second example is a wireless sensor network controlling an “intelligent house”. Typical nodes are small, battery-driven embedded computers computing power and main memory of about 2–3 orders of magnitude below a desktop computer. They communicate using radio modems with low bandwidths (about 20 kbps), which are often switched off to preserve battery power. There are two kinds of nodes: sensors and actuators. Sensors gather information about their surroundings, such as room temperature and light level. Actuators decide on actions such as opening a heating vent or switching on the light based on data they receive from the sensors. Typical system sizes range from a few dozen nodes up to several thousands.

In order to make correct decisions, actuators require data similar to the values measured at the sensors. The required degree of similarity depends on the appli-

cation. In addition, actuators have to agree between them on the actions they take, to avoid situations where only half the lights in a room are switched on. Just as in the virtual soccer game, actions which are taken too late will disturb the users, such as a light switched on only several minutes after a person enters a room. The acceptable delay greatly depends on the kind of action.

2 Related Work

Consistency models and techniques have been subjects of extensive research, especially in fields of multiprocessor systems, distributed databases and interactive groupware systems. Good overviews can be found in the works of Colouris et al. [CDK01] and Galli [Gal00].

Important consistency models in distributed databases and multiprocessor systems include *linearizability* (also known as atomic or strict consistency), *sequential consistency* and *causal consistency*. Linearizability guarantees that write operations are seen by all hosts in the order they were issued as defined by real time [HW87, Her90]. Sequential consistency is weaker as it only requires that all hosts see all write operations in the same order [CDK01]. Causal consistency is even weaker and requires that write operations which are causally related are seen in the same order by every host in the system [Gal00]. However, these models do not put any requirements on the time needed to execute the operations, nor on the state of the system while an operation is being performed. Thus, they are rarely used when dealing with DVEs or WSNs.

Keeping data consistent in sensor networks has not been considered as yet, despite the large number of publications that carry the keywords “consistency” and “consensus” in their title. Their topic is source coding: given a phenomenon (like temperature) how should the sensors sample it? Which interval should be chosen, are there sensors that should sample more often than others, when do sensors agree to send something, how to calibrate the sensors, and how to find out that a sensor is broken and reports bogus data? These questions must be answered before we can tackle the problem at hand. Given the large number of publications, we deem the problems solved. Once the sensors decide that a datum implies an action they send it to the actuators. Since all actuators should act in unison, some mechanism must ensure that all of them have a consistent datum. We are not aware that this problem was treated on its own right in the context of sensor networks.

Several quantitative measures are commonly used to rate the performance of distributed multimedia systems such as VoIP networks and distributed audio and video servers. They typically describe one or more quality-of-service characteristics such as latency, throughput, jitter and loss [OFB01]. These are suited to the delivery of end-to-end data streams, but do not take the inconsistency between multiple receivers into account. Application response time has been proposed by Kim et al., but it does not take the inconsistency of the system into account [KKK03].

There is a large number of publications describing specific distributed virtual environments and wireless sensor networks. Most of these describe the design of a particular system built for a specific application, e.g. a military simulator or an online game. An analysis of several of the most influential designs is given in chapter

V. However, these publications rarely provide data which can be used to compare their performance to other systems.

Chapter II
Consistency

3 Basic Definitions

According to Colouris et al., *"a distributed system one in which components located at networked computers communicate and coordinate their actions only by passing messages.*" [CDK01]. We call a computer which is part of the distributed system[1] S a *host* H[2].

An *object*, e.g. a ball, a player or a light sensor, has a set of variables or *attributes*, such as a velocity or a brightness. The state of an object is a function which maps each of the object's attributes to a specific value. e.g. (velocity, $40\,m/s$).

The *world* W is the set of all objects in a distributed system, e.g. { Ball, Player1, Player2, ... }.

The concept of *time* in distributed systems has been thoroughly reviewed by Lamport in [Lam78]. Today, technology is in place to synchronize system time to within 1 ms for all hosts [Mil06], which is sufficient for our fields of application. Thus, we treat the terms of system time (specific to each host) and real time (as seen from outside the system) as interchangeable, unless explicitly stated otherwise.

An *instance* of an object is the state of an object as seen at a given host at a specific time, e.g. $I(Ball, H_1, t)$ = { (Position, (0, 0)), (Velocity, 40m/s) }, $I(Ball, H_2, t)$ = { (Position, (0, 1)), (Velocity, 39m/s) }.

We say a host *replicates* an object iff an instance of this object exists at the host. The replication is *active* if the host is allowed to propagate changes of this object to other hosts. Otherwise the replication is called *passive*.

The *world view* $V_H(t)$ of a host at a specific time is the set of instances of the objects of a world at this host and time.

An *event* is something which happens at a specific host at a specific time, e.g. "Ball moves two meters to the left". It changes one instance. We distinguish between *deterministic* and *non-deterministic events*. Deterministic events can be replicated at each host without exchanging messages, e.g. "Ball hits a wall". Non-deterministic events can't be replicated without exchanging messages between hosts, e.g. "Player1 kicks Ball".

The *local reaction time* for an event is the time it takes until it is processed on the host where it originates. The *global reaction time* for an event is the time until it has been propagated to and processed by all hosts.

We call a distributed system *consistent at time* t when the world views of all hosts of a system S at time t are identical: $\forall H_1, H_2 \in S : V_{H_1}(t) = V_{H_2}(t)$

The problem of *consensus* is *"... for processes to agree on a value after one or more of the processes has proposed what the value should be."*[CDK01]. For ex-

[1] For the rest of this paper, we use system as a shorthand for distributed system.

[2] Alternatively, the terms *node*, *site*, *process* or *processor* can be used.

ample, hosts could be required to agree whether a goal has been scored or not, or if the heater should be switched on. We mention consensus because it is an important concept in the field of distributed systems. However, it is not discussed further in the context of this work. For a more precise definition of the consensus and a discussion of variations on the consensus problem, we refer to [LSP82] and [PSL80]. In the context of our work, it is important to note that there is a limit on the time it takes the hosts to agree on a value after it has first been proposed. The specific value of the time limit is application dependent.

4 Common Approaches for Achieving Consistency

An ideal system, where communication is unhindered by delays and errors, would be consistent at any time. However, this is impossible in any real world distributed system, because the speed of light puts a lower limit on the time required to propagate events to other hosts. Furthermore, there is an application-dependent upper limit on the local reaction time, which is typically less than or equal to the network delay. This means that hosts have to react to local events before system consistency can be restored.

Various weaker concepts of consistency have been established based on application requirements or technical capabilities. In this section, we will take a look at those concepts which we deem particularly relevant to our fields of interest, i.e. distributed virtual environments and sensor networks.

Consistency approaches primarily differ in how fast events are processed at the host where they occur, and how conflicts between events are resolved. For the moment, we assume that all objects are replicated at all hosts. This is called *total replication*. In chapter IV, we will look at *partial replication* techniques, where only a subset of the world is replicated at each host.

To help in understanding the various approaches to keep a system consistent , we use an example from our virtual soccer game. Suppose that two players are running the simulation on hosts H_1 and H_2. Both are fighting for the ball. At time t_1, player 1 tries to kick the ball. At time $t_2 > t_1$, player 2 also tries to kick the ball. At the real world, the ball would have been kicked away by player 1 already, so player 2 would either miss it, or decide not to kick it at all. But in a distributed system, if $t_2 - t_1$ is less than the network transmission delay from H_1 to H_2, player 2 has no way of knowing that player 1 has already kicked the ball away before him. Thus, a conflict has occurred which has to be resolved by the system, or both players will continue to play two different games.

4.1 Strict Consistency

A system is *strictly consistent* iff the system is consistent whenever a decision has to be made [Gal00]. Strict consistency is used when an application requires that a distributed system must not take any decisions based on an inconsistent system state. For example, in a distributed banking system, all hosts have to agree on a user's account balance before money can be withdrawn from the account. Thus, the application makes sure that the system is in one of two states: in the first states, the system is consistent and ready to process an event. In the second state, an event has

occurred and is being propagated to all hosts in order to re-establish consistency. Conflicts between events will be resolved between all hosts, usually by dropping all but one event. Strict consistency can be implemented using a two-phase-commit protocol (2PC) [ME85].

In our example where two players try to kick the same ball, at time t_1 host H_1 would start to propagate it's decision to kick the ball to all other hosts, including H_2. H_2 would notice that the ball has been kicked by player 1, and either not decide to kick the ball, or reverse a previous decision to kick it. In either case, all hosts will eventually agree that player 1 has kicked the ball. Only then, the ball will begin to move on all hosts, and the game will continue in a consistent state. As the conflict resolution will take at least as long as the network transmission delay from H_1 to H_2, this may well result in the simulation looking like stop-motion soccer if the network is slow. If, for some reason, a message is lost, the hosts cannot reach agreement, and the system fails.

The obvious advantage when using strict consistency is that conflicts between events will be resolved before the events can influence the system state. This can be done without the users ever noticing that such conflicts have even occurred. On the downside, events will not be processed locally before they have been propagated to all other hosts, which implies that for all events, the local reaction time is equal to the global reaction time, which in turn is at least equal to the *largest* network transmission delay in the system. This makes strict consistency approaches generally unsuitable for applications where fast local reactions are required, such as interactive virtual environments. As a further consequence, systems using strict consistency do not scale well with the number of hosts, because the maximum rate at which events can be processed is limited by the reciprocal of the average global reaction time.

4.2 The Loose Consistency Approach

Loose consistency algorithms process events locally as soon as they occur, and propagate them to other hosts later. Conflicts between events are resolved separately at each host. There are several possible methods for resolving conflicts. The most commonly used approach is for each host to send a subset of it's world view to all other hosts. This can be done at regular intervals, or when the difference in world views between hosts exceeds an application-specific threshold. This means that conflicts are resolved by simply overwriting another host's world view.

The loose approach is used when an application requires fast local reactions, but does not require the system to be actually consistent at any time t. It has also been called *best effort consistency* [RS99, CF05, Gal00]. It is the most commonly used approach in interactive distributed virtual environments such as online games or military training simulations.

In our soccer example, player 1 would kick the ball at host H_1 at t_1, and immediately see it moving in a new direction. Then, it would send a message to all other hosts indicating the balls new position and velocity. If this message is received at H_2 earlier than t_2, then player 2 would modify the decision to kick the ball, possibly even dropping the idea. But if the message is received at H_2 later than t_2, then H_2 has already kicked the ball as well, and sent an update message. There are

several possible ways in which H_1 and H_2 might react: they could both ignore or accept the other host's message, or they could decide that one of them is the correct one. In the first case, the game will become inconsistent, because both H_1 and H_2 will proceed with different world views. In the second case, a third host H_3 may have received the incorrect message before the correct one, and may have taken decisions based on the incorrect information. As decisions cannot be taken back, the game would proceed differently from the case where H_2 received the message from H_1 before t_2. Thus, the final result of the soccer game will depend on the random factor of network delay, which is clearly not desirable. If one message gets lost, the other one will obviously determine the future course of the system. If both messages get lost, the system will continue in an inconsistent state until a message on the state of the ball is received by both hosts.

The three major strongpoints of the loose approach are the low local reaction time, its robustness against transmission losses and node failures, and its moderate use of CPU time and memory. In fact, systems using loose consistency can achieve the fastest possible local reaction because they process local events immediately. Error robustness stems from the fact that subsets of world views are transmitted instead of event descriptions. This means that a receiving host can reproduce a subset of another host's world view from just one message describing this part, even when no prior messages describing it have been received. Thus, messages describing an object are sent often enough, other hosts will eventually be updated. The loose consistency approach has two major disadvantages. First, the system is rarely, if ever, consistent. Second, even with identical input data, the final result of an application may often vary each time the application is run, depending on random influences such as network transmission delay or loss rates. This is because decisions made by one host cannot be reversed, but only be invalidated by overwriting the hosts world view. As a consequence, a loose approach is not usable when application results have to be reproducible, or when a high degree of consistency is required, for example for a distributed virtual environment with precise physical modeling. For many applications, however, the world views of the participating hosts are similar enough for the applications purposes. Often, the user interface is designed such that small differences will not matter. For example, in the soccer game, the result of kicking the ball might be that it's automatically shot at the opponents goal, independent of the precise location of the ball and the point of impact of the player's foot.

4.3 The Optimistic Consistency Approach

Similiar to the the loose approach, optimistic consistency algorithms process events locally before they are propagated to other hosts. Conflicts are resolved by keeping a history of events at each host. When a host receives a message about an event which occurred in the past, it backtracks the application to the time at which the event occurs. The application is then computed again from this time, taking into account the newly arrived event.

The optimistic approach is used when an application requires fast local reaction, but can tolerate a higher global reaction time and a significantly higher use of CPU and memory resources compared to the loose approach. The concept orig-

inates from distributed databases and multiprocessor systems [BG87]. Examples for applications following an optimistic approach would be virtual environments with an accurate physical modeling, and distributed scientific simulations.

In the soccer example, player 1 would kick the ball at host H_1 at time t_1. Then, it would send a message describing this event to all other hosts. If the message arrives at H_2 before t_2, the system would behave just like one using loose consistency. But if the message arrives after t_2, there would be a difference: H_2 would backtrack its local simulation back to time t_1, and run it again including the event of player 1 kicking the ball. The attempted kick of player 2 at time t_2 would also be included, but it would most likely miss the ball. However, after both events have been processed in this way at both hosts, and assuming all messages describing events happening before t_2 have also been received by all hosts, the system would have consistent data for all times before t_2. As a consequence, the game's final outcome would be the same at all hosts.

Systems implementing optimistic consistency have local reaction times just as low as systems using loose consistency. Compared to loose consistency methods, they have the advantage of eventually achieving consistency about past system states. As we have shown in another paper, they also achieve a higher degree of similarity between host's world views in comparison to the loose approach [SGK+07]. On the negative side, an optimistic approach cannot be used when backtracking is not possible, and requires much more CPU and memory resources than loose methods. Furthermore, because of the necessity of maintaining a complete history of events from all hosts at each host, existing implementations utilizing an optimistic approach do not scale well for system sizes of more than 20 hosts.

4.4 Hybrid Approaches

Hybrid approaches use a mixture of loose and strict consistency. They are used in applications where loose consistency is sufficient for the majority of objects, but a higher degree of consistency is critical for some of the objects [AF92, Gal00].

In our soccer example, a loose approach would be used for the data about both players and the ball, while strict consistency would be used for ticket vending machines at the stadium entrance. Thus, the actual game itself would suffer exactly the same problem as when using loose consistency, but at least everybodies money would be safe.

Hybrid systems inherit the advantages and disadvantages of the both, loose and strict consistency.

Chapter III
Inconsistency Measures

Loose and optimistic consistency both accept that hosts will make decisions while the system is in an inconsistent state. The impact of inconsistency on the system is twofold:

1. Inconsistencies will cause users – or sensor nodes – to make improper de-

cisions. Even small differences in the value of a single variable may have effects which are highly sensitive in the amount of the actual error. Imagine a penalty kick in the final of the virtual soccer world cup: if the goalkeepers information about the exact position and direction of the ball is even slightly wrong, this might make the difference between winning the cup or ending up second.

2. Usually, users won't immediately notice that the system is inconsistent. Eventually, however, inconsistencies have to be resolved, typically when a host receives a message about an event from another host and updates its own world view accordingly. This will cause non-continuous changes in the host's world view which users will typically notice as unexpected, annoying effects. As an example, let us assume that the ball is flying in a nice, ballistic curve towards a player. When a message is received that another player actually changed the direction of the ball 500 ms ago, this will cause the player to see the ball suddenly changing direction and "jumping" to a new position in mid-air, destroying the illusion of an actual soccer game.

In this section, we propose a number of measures which quantify the level of inconsistency and its effects on system behavior.

5 User Evaluation

The best way to decide which algorithm is the best for a particular application is to actually implement the application in different ways and letting the actual users decide which one is best. User opinion can be gathered in several ways, e.g. by directly asking them for an evaluation, by monitoring their behavior, or by actually selling the application and monitoring the sales numbers.

Unfortunately, it is very expensive to perform user evaluation tests for applications with 10000 or more concurrent users. Thus, such tests are commonly done for smaller systems, typically with no more than 20 or 30 users. However, the results are not always applicable to larger systems as well.

6 Divergence Measure

The divergence measure measures the average difference between the world views of all hosts.

Consider a system where A denotes the total number of attributes, and N the total number of hosts. We define $A \times N$ matrix V, where each column represents the world view of one particular host. We define a second $A \times N$ matrix R, $r_{a,h} \in [0, 1]$. $r_{a,h}$ describes the interest host h takes in attribute a. As both V and R change over time, we write $V(t)$ and $R(t)$ to describe the system state and weighting matrix at time t. We define

$$d_a(t) = \sqrt{\sum_{h=1}^{N} \frac{r_{a,h}(t)}{\sum_{h=1}^{N} r_{a,h}(t)} \left(v_{a,h}(t) - \overline{v_a(t)} \right)^2} \tag{1}$$

as the divergence of the system at time t with regard to attribute a, where

$$\overline{v_a}(t) = \sum_{h=1}^{N} \frac{r_{a,h}}{\sum_{h=1}^{N} r_{a,h}(t)} v_{a,h}(t) \tag{2}$$

is the weighted average value of attribute a over all hosts at time t[3]. In typical implementations, $r_{a,h}$ will be either 1, or 0 if a host is not interested in a particular attribute (cf. partial replication in section 10.4). We define

$$d(t) = \sum_{a=1}^{A} \frac{w_a(t)}{\sum_{a=1}^{A} w_a(t)} d_a(t) \tag{3}$$

as the divergence of the system at time t. $w(t)$, with $w_a(t) \in \mathbb{R}$, denotes the relevance vector, i.e. the relevance of each attribute with regard to the overall inconsistency. An attributes relevance is decided by humans at design time. It is typically based on the perceived influence of this attribute on the decision-finding processes of the application, or on the perception by a user. For example, the exact position of the ball would be very relevant to the decisions of all soccer players at the field, while the color of the shirt of a particular spectator would be almost irrelevant.

For a system with a session[4]length of T, we define the divergence of the system history as

$$D(T) = \frac{1}{T} \int_0^T d(t)dt \tag{4}$$

$D(T)$ is the value we use as the inconsistency measure. It is expected that $D(T)$ is invariant in T.

The divergence measure is applicable if it is possible to determine the world views of all hosts at any given time. Its validity depends on the particular choice of w. The computational complexity scales with $O(N \cdot A \cdot T)$.

7 Discontinuity Measure

The discontinuity measure quantifies the disturbance caused by sudden changes in a host's world view which are due to variables being changed by messages from other hosts.

The discontinuity measure is measured separately for each host. Suppose a message m changes the value of attribute a at host h and time t. Let V and V' denote the system state matrix with and without receiving the message. The change m causes in h's world view is expressed by

$$g_{a,h}(m) = (v'_{a,h}(t) - v_{a,h}(t))^2 \tag{5}$$

Let M_h be the the set of all messages received by h during one session. The average discontinuity at h is written as

$$G_h = \frac{1}{|M_h| \cdot \sum_{a=1}^{A} w'_a} \sqrt{\sum_{m \in M_h} \sum_{a=1}^{A} w'_a g_{a,h}(m)} \tag{6}$$

[3]We assume that $\forall t \forall a \exists h : r_{a,h} > 0$, i.e. for each time t and attribute a, there's at least one host which is interested in a at this time.

[4]A particular invocation of the application software is commonly called a session [Gal00]

where w'_a is a weighting factor denoting the relative importance of non-continuous changes in attribute a, similar to the relevance vector w in section 6. w' is defined at design time.

The discontinuity measure does not depend on any particular consistency technique. It is well suited to applications such as virtual environments, where users are very sensitive to non-continuous changes in attributes. Just as with the divergence measure, the results greatly depend on the choice of w'.

8 Yield Measures

Yield measures attempt to measure the influence of inconsistency on the decision-making process of the users of a distributed systems. As we have argued before, these influences are highly application-dependent. They are usually difficult to formulate in terms of a mathematical formula. Instead, yield measures are chosen heuristically for each application in an attempt to express the "goal" of an application in a single value.

For example, in the virtual soccer game, one could use the number of bad passes per second as the yield measure. This would be based on the hypothesis that a less effective solution would cause players to make more mistakes, e.g. because they've got incorrect information about the potential receiver of a pass. There is usually more than one possible choice of measure, which can yield different rankings for the same set of solutions.

Yield measures can be applied to any system in which at least a subset of the hosts is accessible for data gathering. Their validity depends on the choice of a particular measure, and there is no general rule for how to come up with a good one. Yield measures are typically chosen so that they are simple to implement. Their computational complexity depends on the specific measure, but typically scales with $O(N \cdot T)$.

Chapter IV
System Scalability

So far, we have mainly concerned ourselves with how to maintain data consistency in a distributed system, and how the effects of consistency on application performance can be measured. However, if we want to build the large-scale systems described in section 1, we have to consider the issue of scalability of the various consistency techniques presented in chapter II. In this chapter, we look at the cost of maintaining consistency, and at various methods to reduce this cost. By cost, we mean the amount of resources used at each host.

In general, the amount of resources required in a distributed system depends on the number of hosts (N) and the number of objects (M) in the system. In order to build an arbitrary large system, the costs per host have to be independent of the system size, i.e. $O(1)$. Obviously, this is possible only if M does not grow faster than $O(N)$. Otherwise, the average number of objects each host has to replicate would grow with the system size, eventually overloading one or more hosts. Fortunately,

this condition typically holds for the applications we are primarily concerned with, i.e. distributed virtual environments and wireless sensor networks.

9 Measuring Costs

The primary cost factors at each host of a distributed system are the amount of CPU cycles, memory and network capacity consumed. In a wireless sensor network, there's the additional concern of energy consumption, which can be expressed as a function of the other three factors.

CPU load can be measured in two ways. The first method is to profile the actual binary executed at each host, and count clock cycles or measure execution times at each function. This method is easy to realize, but suffers from the disadvantage that results are not directly comparable between different hardware architectures. In heterogenous system, with many different types of hosts, this is not practical. The second method is to analyze each important function in the code and assign a cost factor to it. Then, it is enough to count how often each function is called. This has the advantage of giving results which are independent of the hardware architecture used. However, it means more work for the application developers as they have to analyze each function for their computational complexity. This is prone to errors, and has to be done again each time there's a major change to the code.

Memory consumption can be measured similarly to CPU load. However, it is more suitable to automated profiling, as it is less hardware specific.

Network load can be measured in several ways: packets sent, packets received, bytes sent, bytes received. In the applications we are looking at, messages tend to be small (about 100 bytes). In typical network layers, the cost for transmitting a packet does not depend on the size of a packet as long as it does not exceed a certain limit. For wireless sensor networks, the limit is typically about 128 bytes. For internet connections, it is typically a few kilobytes. Thus, it is feasible to ignore bandwidth considerations, and only measure the number of packets sent or received at each host.

Energy consumption in wireless sensor networks can be measured directly if the hardware supports this, or indirectly by measuring how long the battery lasts. In a simulation testbed, it can be modeled as a function of CPU load, memory consumption and network load.

10 Optimizing Network Load

The major cost factor in typical distributed systems is the network load. Modern desktop computers have ample computing power and main memory, while network connections are still limited and expensive in comparison. The same holds true for nodes in a wireless sensor network, where the radio modem dominates with regard to power consumption. Thus, in typical applications there is a tradeoff where a higher CPU load or memory consumption is accepted in return for a reduction in network load.

In the following sections, we will look at several methods to reduce the network load per host in a distributed system.

10.1 Limiting system size

A very simple way to reduce the network load at each host is to put an upper limit on the system size. Although this is a very simplistic and crude method, which doesn't involve any changes to the codebase, it is mentioned here because it is commonly done in practice. For example, in most Massively Multiplayer Online Games (MMOG), the world size is restricted to a few thousand or ten thousand players. When more players want to participate, a new world is created which is not connected to the other worlds. This approach is simple yet effective, but still does not not solve the problem of putting more concurrent users into the same system.

10.2 Message Aggregation

Sending several application messages together in one network packet reduces network load by sending one packet for several messages, instead of one packet per message. There are two ways to realize this: merging and batching. Message merging tries to combine two different messages on a semantic level, i.e. it creates a new message which contains the information from the two original messages. For example, if one message says that the ball is flying with velocity v_1 at position p_1 at time t, and another message says that the ball changes color to red at the same time t, it would be sufficient to send a single message describing the balls position, color and velocity at time t. Message batching exploits the fact that transmission costs are constant for packets up to a certain size by sending several messages together in a single packet. This may require a host to buffer messages until sufficient messages are available for sending, thus increasing the global reaction time of the system. Usually, a tradeoff has to be made between more efficient transmission by waiting longer, and faster global reaction by sending sooner.

Both, merging and batching, require the messages to have the same set of recipients, or messages will be sent unnecessarily. Message batching requires knowledge about the underlying network layer at application level.

10.3 Exploiting Determinism

A very efficient way to reduce the number of messages sent in a distributed system is to avoid sending messages which describe deterministic events. A very simple method is dead reckoning, where future positions of objects in virtual environments are predicted from known positions and velocities in the past. Dead reckoning has been shown to significantly reduce the number of packets sent without having a visible impact on application behavior [MZP+94]. More sophisticated techniques attempt to model an objects behavior, further reducing the number of non-deterministic events at the expense of more computationally complex prediction routines.

The downside of only transmitting deterministic events is a loss of robustness against transmission failures. If only one message is transmitted per non-deterministic event, and this message gets lost, receiving host cannot reproduce the event, leading to an inconsistent system. As a consequence, most applications use redundant transmissions.

10.4 Partial Replication

None of the solutions presented so far achieve our stated goal of $O(1)$ complexity of resources required at each host. In fact, it is not possible to do so with any of the consistency and resource optimization techniques we have discussed until now, unless we make a major modification to our premises. So far, we have silently assumed that every host in the system replicates *all* objects in the world (*total replication*). This means that whenever there's an event modifying an object, it has to be propagated to every host in the system. Furthermore, each host will be informed of every event, so the number of messages each host has to process will grow with $O(M)$ if we assume a constant rate of events per object. If we want to achieve $O(1)$ complexity, we have to give up the concept of total replication, and move to a design where each host replicates only a subset of the world (*partial replication*.

The fundamental idea behind partial replication is that two hosts will only exchange messages about an object if the object is *relevant* to both of them. This will reduce the complexity for each host from $O(M)$ to $O(|L(H)|)$, where $L \subset W$ is the subset of the world which host H is interested in. There are major aspects of partial replication:

1. a definition of relevance
2. a method to determine, for a given object, the set of hosts this object is relevant to, at runtime
3. a routing layer responsible for delivering to each host the messages about objects considered to be relevant to this host

Relevance can be defined as a function $R : W \times S \mapsto \{0, 1\}$, with $R(O, H) = 1 \iff O$ is relevant to H, with $O \in W$ and $H \in S$. The exact definition of R is chosen at design time.

In practice, the second and third item are usually closely connected, so we will look at them as a unit. In many applications, the application level routing layer in point 3 depends on the underlying network layer.

In the next sections, we are going to look at different concepts of the replication mechanisms in a distributed system, in a roughly chronological order of their first appearance.

10.4.1 Total replication with broadcast

Early distributed systems simply broadcasted messages to all nodes in a network, no matter whether they were actually participating in the application or not [MZP+94]. While this is a very simple method, it poses a high network load ($O(M \cdot$

N)) on each node in a network, limiting its use to small (< 1000 participants) dedicated systems, such as a custom-build LAN.

10.4.2 Relevance definition

Most distributed virtual environments define relevance based on the location of a player in the virtual world. The first systems divided the world into fixed, hexagonal cells [CMBZ00, KLXH04]. A host would be interested in objects which are located in the same cell as the hosts player, and possibly a number of adjacent cells. This is a very simple approach which doesn't take into account the setup of a particular world, and doesn't adapt to changes in the world very well. For example, a cell in an empty desert would be just as large as the one in the middle of a soccer stadium, but it would usually contain far fewer objects and players.

To overcome the limitations of fixed cells, some applications divide the world into fixed *regions* which are chosen according to the worlds layout [Gre96]. There is no interaction nor communication across region boundaries. In our previous example, the desert might be one such region, and the soccer stadium another one. When people leave the desert to enter the stadium, they receive no more updates on changes to the desert. Regions help to handle "hotspots" in a virtual world, where lots of activity takes place. However, they still fail to adapt to changing circumstances. For example, it might be perfectly fine to combine the desert and the stadium into one region on Mondays, when no games are on, while it may be a good idea to subdivide the stadium even further on Sundays, when the championship playoffs are being held.

Locales remove the restriction that no interaction is allowed between regions [BWA96, PG00]. Hosts can announce their interest in their own and adjacent locale. They are still fixed at design time, and can't adapt to changing circumstances when the application is running.

Further refinements on locales use information about visual obstruction in a virtual environment, but still didn't allow to change the division at runtime [Fun95]. For example, people in the stadium wouldn't receive updates on objects outside the stadium, because these would be invisible to them.

An adaptive definition of relevance uses the scene graph used in virtual environments [FS98]. Hosts can express interest in arbitrary subsets of the scene graph. Scene graphs change over time, making objects relevant or irrelevant to a host. For example, a player might be interested in the parking lot of a stadium. If a car moves into the parking lot, the player would automatically be notified on event concerning objects inside the car as well, as the representation of the car would be a subtree of the scene graph of the parking lot.

In systems using *aura collision* for defining relevance, each object defines an aura, i.e. a circular area centered on its own position. An object is relevant to a host if the aura of the player using the host intersects the aura of the object [GB95, FS98, BRS02]. As auras move with their objects, this approach adapts to changes in the system very well. It also permits users to scale the load on their machine by changing the extent of their aura.

10.4.3 Relevance computation and message routing

Multicasting is commonly employed as the routing mechanism of choice in distributed systems, as it allows to efficiently combine interest management and message routing [CMBZ00, WAB+97, Gre96, PG00]. Objects send update messages to one or more multicast groups, which can be a cell or locale, for example. Host express their interest by joining the appropriate multicast groups. The actual message routing is done by the multicast network layer.

The biggest disadvantage of multicasting is that it's is still not universally supported by network routers. Thus, it only works on isolated "islands" in the Internet. Some systems try to circumvent this by using proxies to connect multicast-capable parts of the Internet [FS98].

Aura collision can be realized using a *dedicated aura server*. All objects constantly send their position to the server [GB95]. When a host wants to send a message describing an object, it asks the server for a list of hosts this object is relevant for. Such a system does not scale very well, as the aura server is a performance bottleneck, and also a single point of failure in the system.

More recent approaches attempt to distribute the workload of an aura server between all hosts by utilizing a distributed publish/subscribe mechanism to deliver messages [BRS02]. Hosts will express their interest by sending subscriptions to the network, which will be matched against publications which are update messages on objects. This approach scales very well with the number of hosts, but known implementations suffer from unacceptable routing delays due to bad design choices for the routing layer.

A very promising design described by Knutsson et al. uses *distributed hash tables* implemented by peer-to-peer overlays such as Pastry for finding the host responsible for a cell citeKLXH04, RD01. Message routing is then performed using unicast. These systems show very good scalability properties, and can profit from the built-in properties of peer-to-peer overlays such as error robustness and automated latency optimization.

Chapter V
Existing solutions

11 Classification issues

We classify existing solutions for distributed virtual environments and wireless sensor networks with regard to two main criteria: which approach is used for achieving consistency, and how is partial replication realized? A classification along quantifiable data, i.e. how many hosts or users are supported, how consistent is the system etc., is impossible, as no common benchmark exists against which existing solutions can be tested.

Partial replication techniques are classified according to the three major aspects described in section 10.4: definition of relevance, computation of relevance at runtime, and message dissemination. As in our description above, we will combine

relevance computation and message dissemination, as they are closely connected in most implementations. Where possible, we will provide the maximum number of concurrent users supported by a particular solution.

There are several other design aspects, such as the exploitation of determinism, and whether a hierarchical client-server design or a decentralized peer-to-peer architecture is used. We will mention such as aspects where information is available.

12 No Partial Replication

SimNet, the earliest large-scale distributed virtual environment, did not use partial replication. It was developed for the U.S. Department of Defense as a virtual training environment for military exercises from 1983 on. It used a dedicated LAN and offered support for more than 1000 concurrent users. Together with the *NPSNET* project, which was launched in 1986 and is technically very similar to *SimNet*, it resulted in the DIS (Distributed Interactive Simulation) IEEE standard 1278 in 1993 [MZP+94, IEE93]. Both, SimNet and NPSNET use a serverless peer-to-peer architecture. They exploit determinism by a technique called *dead reckoning*, where future positions of objects are predicted from positions in the past and known movement vectors [Gal00].

HLA (High Level Architecture) is a middleware for distributed applications with is based on CORBA. It uses a peer-to-peer architecture and does not utilize partial replication. In typical applications, HLA supports up to 100 concurrent users [CW96].

MiMaze is a distributed virtual environment developed for research purposes. It does not use partial replication, and does not exploit determinism. It supports up to 25 concurrent users [DG99].

The commercial computer game *X-Wing vs. TIE Fighter* is a space flight simulator where up to eight players can play together. It does not use partial replication [Lin99].

Trickle [LPCS04] uses a loose consistency approach to reliably deliver messages to nodes in a sensor network. It does not, however utilize partial replication.

13 Definition of Relevance

NPSNET-V, launched in 1995, expands the original NPSNET with a partial replication component which defines relevance based on fixed, hexagonal cells. Hosts are interested in their local cell and the surroundings cells. NPSNET-V supports more than $10,000$ concurrent users [MBZ+95, MZP+95, CMBZ00].

In 2004, *SimMud* utilized a simple cell grid for defining relevance in their research prototype. They've verified that their design is feasible for up to 4,000 concurrent users. Experimental results indicate that far more users could be supported were it not for (unrelated) restrictions of their testbed platform [KLXH04].

The *MASSIVE-2* project was a research project about scalability issues in distributed virtual environments. It defines relevance based on fixed regions, which could be of arbitrary shape. Each host is interested only in the region of its current location. There is no interaction across region boundaries [Gre96].

In 1996, *SPLINE* extended on SimNet by using locales instead of fixed cells to define relevance. Hosts are interested in their own and neighboring locales [BWA96, WAB+97]. A similar approach was used in *MASSIVE-3/HIVEK* (1999), where an automated cost-benefit analysis may extend a hosts range of interest beyond its immediate neighborhood [PG00].

Visual obstruction was used to define relevance in the *RING* project in 1995. RING supports up to 1000 concurrent users [Fun95].

DIVE, another descendant of SimNet, is a research platform for distributed virtual environments developed between 1992 and 1998. DIVE defines relevance based on the scene graph of the virtual world. Arbitrary subsets of the scene graph are be combined into "lightweight groups". Hosts can decide freely whether a lightweight groups is relevant to them or not. Group composition and host interest can change at runtime. DIVE uses a peer-to-peer architecture [FS98, HLS97].

MASSIVE (1994) and *Mercury* (2002) both are research platforms which define relevance using aura collision. Both use a peer-to-peer architecture, and support up to 80–100 hosts [GB95, BRS02].

In a Geographic Hash Table (GHT) [RKY+02], data is stored in a sensor network by mapping the data description to a geographic address, for instance all temperature readings between 10 °C and 25 °C are stored in some geographic region. Any node in this region can answer this question, but usually only one node (primary) answers. If it fails, a backup takes over. The Perimeter Replication Protocol (PRP) determines which nodes should act as a backup and when they take over. The primary sends its complete data periodically to the backups.

14 Computation of Relevance and Message Dissemination

In research projects, multicasting is the most popular mechanism for relevance computation and message dissemination. It is used in NPSNET-V, SPLINE, MASSIVE-2 and -3, DIVE. However, due to the lack of multicast support in many parts of the Internet, it is still not commonly used in commercial applications.

MASSIVE uses a dedicated aura server for computing relevance. Message dissemination is performed by each host using unicast.

Mercury uses a distributed publish/subscribe network for computing relevance and disseminating messages. Network load scales with $O(\log N)$ in the number o hosts, but due to its simple routing layer, the message latency, and hence global reaction time, scales with $O(N)$. For 100 hosts, latency averages at 1 s, far too high for highly interactive applications.

In SimMud, each cell is assigned a controlling host which manages a list of all objects in the cell. This host is located using a distributed hash table implemented with Pastry, a peer-to-peer overlay. If a host is interested in a cell, it registers with the cells controller. The controller multicasts all messages to the interested hosts.

15 Loose Consistency

Loose consistency has been used by nearly all the systems presented so far, and also in wireless sensors networks. The main reasons are that loose consistency is easy to implement, very robust against errors and keeps inconsistency sufficiently low for the relatively simple applications research has focused on so far.

16 Optimistic Consistency

Optimistic consistency techniques have been used by some applications which require a degree of consistency unattainable with loose techniques, such as realistic physical modeling. Optimistic consistency allows verifiable *correctness* as defined by [MVHE04] *after* the application has finished.

X-Wing vs. TIE Fighter has used optimistic consistency techniques to implement a realistic physical behavior of objects such as spacecraft and meteors. Only non-deterministic events are sent to other hosts using reliable communication.

Some commercial online games use optimistic consistency mechanisms for a subset of the game state information to prevent cheating [CFKJ02, CFJ03].

Optimistic consistency is widely used in distributed databases and multi-processor architectures [MT01, RG01].

17 Hybrid Consistency

HLA uses a hybrid consistency technique. By default, data is kept consistent using a loose approach, but the application developer can select particularly important attributes which will be kept consistent using a strict consistency technique.

18 Summary

Table 1 gives an overview of the solutions presented so far classified according to the consistency technique and partial replication techniques used.

	Loose	**Optimistic**	**Strict**	**Hybrid**
Total	SimNet, NPSNET, MiMaze, Trickle, PSFQ	X-Wing vs. TIE Fighter, Trickle, PSFQ		HLA
Zones	NPSNET-V, SimMud, MASSIVE-2, -3, RING, SPLINE, DIVE, GHT			
Aura	MASSIVE, Mercury			

Table 1: Overview of existing solutions. Rows show different definitions of relevance. Columns show different consistency techniques.

The overwhelming majority of existing solutions uses loose consistency techniques with partial replication based on zones. Noteworthy, none of the existing

implementations utilizing other consistency techniques use any sort of partial replications. As a consequence, they are not able to support large numbers of concurrent hosts.

Chapter VI
Conclusions

We have shown that in distributed virtual environments and wireless sensor networks, hosts have to react to events within a time which is less than the network latency. As a consequence, designing a consistency algorithm for such a system involves a tradeoff between reaction time, consistency and scalability. Most existing solutions favor reaction time over consistency, and accept that hosts have to act while the system is in an inconsistent state.

In this paper, we have presented the most important approaches to keeping DVEs and WSNs consistent, and analyses their respective strengths and weaknesses. As a first step towards a methodology where different solutions can be compared with each other, we have proposed several measures for the inconsistency in a distributed system, as well as various measures for the cost associated with keeping the system consistent.

Finally, we have analyzed a number of representative implementations of DVEs and WSNs and how they deal with the problem of consistency. We have found that the majority of existing solutions employs a loose consistency approach with zone-based partial replication, which is both simple and robust against network and node failures. However, the loose approach is unsuitable for applications with require a higher degree of consistency, such as simulations which employ physical modeling. Existing implementations of consistency algorithms which reduce the level of inconsistency below that provided by the loose approach do not permit the utilization of partial replication, rendering them unable to support large numbers of concurrent hosts.

Quantitative data about the quality and scalability of existing solutions is very hard to come by. Publications by the system designers rarely give numbers for the maximum concurrent hosts support by, or how they compare to other systems. To this end, we have created Adam, a modular simulation testbed for distributed systems which is described in [SGK+07]. We plan to use Adam to create a benchmark test against which existing and new solutions can be compared.

References

[AF92] Hagit Attiya and Roy Friedman. A correctness condition for high-performance multiprocessors. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 679–690, 1992.

[Arm03] Grenville Armitage. An Experimental Estimation of Latency Sensitivity in Multiplayer Quake 3. In *Proceedings of the 11th IEEE International Conference on Networks (ICON 2003))*, pages 137–141, Sydney, Australia, September 2003. IEEE Press.

[BG87] P.A. Bernstein and N. Goodman. Concurreny control in distributed systems. *Computing Surveys*, 12(2):185–221, June 1987.

[BRS02] Ashwin R. Bharambe, Sanjay Rao, and Srinivasan Seshan. Mercury: a scalable publish-subscribe system for internet games. In *Proceedings of the first workshop on Network and system support for games*, pages 3–9, 2002.

[BWA96] J. Barrus, R. Waters, and D. Anderson. Locales and beacons: Precise and efficient support for large multi-user virtual environments. In *Proceedings of VRAIS'96*, pages 204–213, 1996.

[CDK01] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: concepts and design*. Addison-Wesley, third edition, 2001.

[CF05] Angie Chandler and Joe Finney. On the Effects of Loose Causal Consistency in Mobile Multiplayer Games. In *NetGames '05: Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–11, 2005.

[CFJ03] Eric Cronin, Burton Filstrup, and Sugih Jamin. Cheat-Proofing Dead Reckoned Multiplayer Games. In *Proceedings of the 2nd International Conference on Application and Development of Computer Games – ADCOG 2003*, Hong Kong SAR, China, January 2003.

[CFKJ02] Eric Cronin, Burton Filstrup, Anthony R. Kurc, and Sugih Jamin. An Efficient Synchronization Mechanism for Mirrored Game Architectures. In *NetGames '02: Proceedings of the 1st workshop on Network and system support for games*, pages 67–73, 2002.

[CMBZ00] Michael V. Capps, Don McGregor, Donald P. Brutzman, and Michael Zyda. NPSNET-V: A New Beginning for Dynamically Extensible Virtual Environments. *IEEE Computer Graphics and Applications*, 20(5):12–15, 2000.

[CW96] James O. Calvin and Richard Weatherly. An Introduction to the High Level Architecture (HLA) Runtime Infrastructure (RTI). In *14th Workshop on Standards for the Interoperability of Distributed Simulations*, March 1996.

[DG99] Christophe Diot and Laurent Gautier. A Distributed Architecture for Multiplayer Interactive Applications on the Internet. *IEEE Network magazine*, 13(4):6–15, July/August 1999.

[FS98] Emmanuel Frécon and Mårten Stenius. DIVE: A Scalable Network Architecture for Distributed Virtual Environments. *Distributed Systems Engineering Journal*, 5(3):91–100, September 1998.

[Fun95] Thomas A. Funkhouser. RING: A client-server system for multi-user virtual environments. In *Symposium on Interactive 3D Graphics*, pages 85–92, 209, April 1995.

[Gal00] Ricardo Galli. *Data Consistency Methods for Collaborative 3D Editing*. PhD thesis, Universitat de les Illes Balears, November 2000.

[GB95] Chris Greenhalgh and Steven Benford. MASSIVE: A Collaborative Virtual Environment for Teleconferencing. *ACM Trans. Comput.-Hum. Interact.*, 2(3):239–261, 1995.

[Gre96] C. Greenhalgh. Spatial scope and multicast in large virtual environments. Technical Report NOTTCS-TR-96-7, Department of Computer Science, University of Nottingham, UK, 1996.

[Her90] Maurice Herlihy. A Methodology for Implementing Highly Concurrent Data Structures. In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 197–206, 1990.

[HLS97] Olof Hagsand, Rodger Lea, and Mårten Stenius. Using spatial techniques to

decrease message passing in a distributed ve system. In *Proceedings of the second symposium on Virtual reality modeling language*, pages 7–ff. ACM Press, 1997.

[HW87] Maurice P. Herlihy and Jeanette M. Wing. Axioms for Concurrent Objects. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 13–26, 1987.

[IEE93] IEEE. IEEE 1278 standard for information technology - protocols for distributed interactive simulations applications. Entity information and interaction , 1993.

[KKK03] Sung-Jin Kim, Falko Kuester, and K. H. (Kane) Kim. Towards Enhanced Data Consistency in Distributed Virtual Environments. In *SPIE '03: Stereoscopic Displays and Virtual Reality Systems X*, volume 5006, pages 436–444, 2003.

[KLXH04] Björn Knutsson, Honghui Lu, Wei Xu, and Bryan Hopkins. Peer-to-Peer Support for Massively Multiplayer Games. In *Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, volume 1, 2004.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Lin99] Peter Lincroft. The Internet Sucks: Or, What I Learned Coding X-Wing vs. TIE Fighter. In *Proceedings of the Game Developer's Conference*, September 1999.

[LPCS04] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: A self-regulating algorithm for code maintenance and propagation in wireless sensor networks. In *First USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, 2004.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[MBZ+95] Michael R. Macedonia, Donald P. Brutzman, Michael Zyda, David R. Pratt, Paul T. Barham, John Falby, and John Locke. NPSNET: A Multi-Player 3D Virtual Environment over the Internet. In *Symposium on Interactive 3D Graphics*, pages 93–94, 210, 1995.

[ME85] B. Moss and J. Elliot. *Nested Transactions: An Approach to Reliable Distributed Computing*, pages 31–38. MIT Press, 1985.

[Mil06] David L. Mills. Network Time Protocol Version 4 Reference and Implementation Guide. Technical Report 06-06-1, University of Delaware, Electrical and Computer Engineering, June 2006.

[MT01] José F. Martínez and Josep Torrellas. Speculative locks for concurrent execution of critical sections in shared-memory multiprocessors. In *Workshop on Memory Performance Issues (WMPI), at International Symposium on Computer Architecture (ISCA)*, Gothenburg, Sweden, June 2001.

[MVHE04] Martin Mauve, Jürgen Vogel, Volker Hilt, and Wolfgang Effelsberg. Local-lag and Timewarp: Providing Consistency for Replicated Continuous Applications. *IEEE Transactions on Multimedia*, 6(1):47–57, February 2004.

[MZP+94] Michael R. Macedonia, Michael J. Zyda, David R. Pratt, Paul T. Barham, and Steven Zeswitz. NPSNET: A Network Software Architecture for Large-Scale Virtual Environments. *Presence*, 3(4):265–287, 1994.

[MZP+95] Michael R. Macedonia, Michael J. Zyda, David R. Pratt, Donald P. Brutzman, and Paul T. Barham. Exploiting Reality with Multicast Groups: A Network Architecture for Large-scale Virtual Environments. In *Proceedings of the 1995 IEEE Virtual Reality Annual Symposium*, pages 2–10, 1995.

[OFB01] João Orvalho, Pedro Ferreira, and Fernando Boavida. State Transmission Mechanisms for a Collaborative Virtual Environment Middleware Platform. In *IDMS '01: Proceedings of the 8th International Workshop on Interactive Distributed Multimedia Systems*, pages 138–153, 2001.

[PG00] J. Purbrick and C. Greenhalgh. Extending Locales: Awareness Management in MASSIVE-3. In *Proceedings of Virtual Reality, March 2000*, pages 287–287, March 2000.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–34, April 1980.

[RG01] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 294–305, December 2001.

[RKY+02] Sylvia Ratnasamy, Brad Karp, Li Yin, Fang Yu, Deborah Estrin, Ramesh Govindan, and Scott Shenker. Ght: a geographic hash table for data-centric storage. In *Proceedings of the first ACM international workshop on Wireless sensor networks and applications*, pages 78–87. ACM Press, 2002.

[RS99] M. D. Ryan and Paul M. Sharkey. The Causal Surface and its effect on Distribution Transparency in a Distributed Virtual Environment. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, pages 75 – 80. IEEE, October 1999.

[SGK+07] Jan Sablatnig, Sven Grottke, Andreas Köpke, Jiehua Chen, Ruedi Seiler, and Adam Wolisz. Adam – a simulator for distributed virtual environments. Technical Report in preparation, TU-Berlin, 2007.

[WAB+97] Richard C. Waters, David B. Anderson, John W. Barrus, David C. Brogan, Michael A. Casey, Stephan G. McKeown, Tohei Nitta, Ilene B. Sterns, and William S. Yerazunis. Diamond Park and Spline: A Social Virtual Reality System with 3D Animation, Spoken Interaction, and Runtime Modifiability. *Presence: Teleoperators and Virtual Environments*, 6(4):461–480, August 1997.