TKN Telecommunication Networks Group

Technische Universität Berlin
Telecommunication Networks Group

Adam – A DVE Simulator

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

Berlin, February 8, 2008

TKN Technical Report TKN-08-004

TKN Technical Reports Series

Editor: Prof. Dr.-Ing. Adam Wolisz

Abstract

During the past 25 years, a large amount of distributed virtual environment (DVE) systems has been proposed. Each was built to support a single scenario. This not only makes it impossible to rank these various DVEs, it also obscures the effect of each of the algorithms used in those systems. This makes it very hard to select the algorithms for a new scenario unless this scenario is a fairly exact copy of one of the previous sample-scenarios.

To overcome this situation, we have created a modular simulator-based DVE testbed named *Adam* with the ability to plug in different scenarios as well as different algorithms. The testbed also contains a large set of measuring tools to compare and rank the effect of each algorithm and to allow finding optimal parameters for those algorithms.

Our testbed currently supports two scenarios and several of the most common consistency algorithms found in the literature. We can compare the solutions on an objective scale and confirm that optimistic consistency typically outperforms loose consistency.

Chapter I
Introduction

Distributed virtual environment (DVE) systems have been technically feasible since the mid-1990s. Although there has always been large public and scientific interest in the technology, there are, to date, almost no large-scale distributed virtual environments in widespread use, the only exceptions being a few computer games.

The reason why DVEs have not found more distribution is unknown, but we think it is because the experience to the user is still unsatisfactory in most cases, when implemented with the still-limited network constraints. The quality of DVEs was expected to improve when network resources improve. While some network parameters (especially bandwidth) did in fact improve dramatically over the past 10 years (and will likely continue to improve), some have not improved a lot. Delay, in particular, cannot improve much further as information exchange is limited by light speed. For example, it is not possible to reach a ping value of less than 100ms from Europe to Australia and back. Therefore, we believe the shortcomings of the network will have to be compensated by better algorithms in the DVE itself. For this reason it is useful to analyze and compare existing DVE solutions to find their weak points and possibly improve on these systems.

It is difficult, however, to actually estimate the quality of any published DVE since there is no quality standard or measure to compare it to quantitatively. Most publications will simply cite that the final result was “satisfactory to the test-users.” Since the experience of a virtual environment and the effect of its playout errors are highly subjective, it is usually considered impossible to define such a comparison measure beyond the simple satisfactory-statement.

While we agree that the final result of a DVE will always have to be judged subjectively by users, we do think that a fair set of measures exist which allows a system designer to estimate what the final quality should be, given the considered algorithms. A few such measures were defined in [GSK+07].

To actually *use* these measures, a standard measuring code is required. This paper introduces such a code. The code implements some of the most common algorithms used in DVEs, a few simple scenarios and of course a large set of measurement tools.

The code will be downloadable and extensible so that other researchers can rank their own algorithms and solutions.

Chapter II discusses other similar projects and related publications. Chapter III describes the principal setup of our testbed. Chapter IV details the implemented consistency algorithms, while chapter V talks about the network models currently installed. Chapter VI shows our scenarios and chapter VII lists the measurements being performed. In chapter VIII we describe some standard experiments and how to perform them. A few early results of those experiments are shown in chapter IX. Chapter X explains what we are going to do next with the testbed. Finally, chapter XI gives a short summary of the contributions of this paper.

Chapter II
Related Work

The first major DVE was SimNet/NPSNET, which became the DIS standard in 1993 [MZP+94]. A large array of research DVEs were built similar to DIS, with some extensions. These extensions usually focused on exploiting locality, such as SPLINE [BWA96] or DIVE [FS98]. DIS was originally designed for military battlefield simulations on dedicated networks and very few of the DIS-alikes are in production use outside of this application.

On the other hand, computer games such as “X-Wing vs. TIE Fighter” (1997) [Lin99] gauged Internet connected simulations early on. “Ultima Online” (1997) connected thousands of players on a single server, “World of Warcraft” (2004) had 9 million subscribers in 2007. Surprisingly, the consistency algorithms used have not changed much between the latter two programs and are akin to DIS, except that centralized servers are used.

The research game application PaRADE (1997) was highly interactive and attempted to solve the Internet-delay problem through optimistic consistency and through prediction [RS97]. MiMaze (1999) was another very interactive research game application, but it used DIS-like mechanisms [DG99]. More recently in 2002, Cronin et al. researched algorithms to improve the consistency of an existing, highly interactive game, using optimistic consistency to achieve this [CFKJ02].

Since the current research focus is most often on scalability, i.e. the ability of a DVE to support many thousands of players at the same time, researchers have started to abandon user tests as too expensive, running simulations instead. RING already simulated 1000 users in 1995 [Fun95]. While Mercury (2002) simulated only 100 hosts [BRS02], Knutsson (2004) simulated 4000 hosts [KLXH04]. However, all of these simulations were used to find the effect of a specific algorithmic change or of an algorithmic parameter in an otherwise monolithic application and are completely nontransferable to other scenarios or other algorithms.

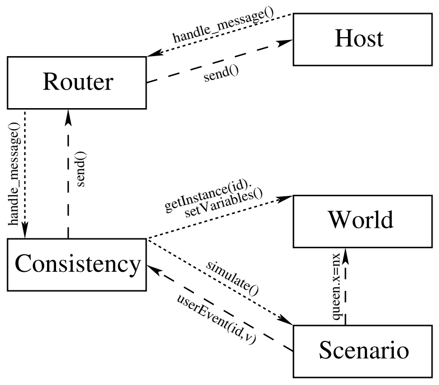

Figure 1: Modules in a Single Simulated Host

Chapter III Architecture

Running and measuring the effectiveness of a large virtual environment is a difficult problem. On the one hand, one cannot possibly find thousands of real users to participate. On the other hand, there are many types of computers and networks that may need to be tested for and one cannot provide all of those.

We have therefore decided to rather simulate the entire network on a single computer. The user input to be simulated at each simulated host is generated by a scenario-specific artificial intelligence.

Adam, our testbed, is built on top of the discrete event simulator OMNet++ [Var01] in conjunction with the MRIP engine Akaroa [EPM99].

During the simulation, our testbed creates a host-module for each host to be simulated and connects them via OMNet++ network connections, allowing OMNet++ to simulate the entire connecting network. Each host-module then creates a number of modules, the most important of which are shown in figure 1:

Host: A representation of a single physical host the DVE runs on. It instantiates and controls the other modules and is also the OMNet++ interface.

Router: The network layer on a host. Implements reliable transfer algorithms as well as message aggregation if needed.

Consistency: The consistency layer on a host. This decides when to send update information to other hosts and what to do with incoming information. Often, this will hold additional, internal world views.

World: The world view on this host. All objects in this world and their states are collected here as a set of named variables.

Scenario: The scenario-specific code. This contains the rules dictating how objects in this world move about or interact. It also contains a scenario-specific artificial intelligence (*AI*) that issues user-commands.

All modules are called at regular intervals (called *ticks*) to update themselves and clean up any outstanding operations. Amongst other things, this allows the scenario to update the world view according to the scenario rules, thus progressing the virtual environment. Also, the AI decides on new actions during these calls.

When the scenario needs to change the values of any variable in the world view, it can simply do the change directly. If the scenario's AI, however, performs a user event, it will ask the consistency algorithm to perform the event instead. The consistency algorithm can then apply the event directly or delegate it for later application. The consistency may create a message to inform other hosts of this event. If so, the message is passed to the router which then decides when and how to send this message to other hosts via the host's OMNet++ interface.

When a message arrives at a host, it is passed to the consistency layer. The consistency layer analyzes the type of message. Depending on the algorithm in use, it may then drop the message, or add it to its message history, or implement the effect of the message directly by applying it to the world view. Afterwards, it may also call the scenario to update the world view.

The modules described above are implemented as C++ objects. The actual implementations (e.g. a Swarm scenario, or a loose consistency) are then inherited from these base modules, overloading their major interface functions. By using OMNet++ configuration files the user then decides which combination of objects to combine and sets their parameters. This allows easy access to any mix of available network features, algorithms and scenarios.

We have programmed several different scenarios (see chapter VI) and consistency algorithms (see chapter IV). The choice between these modules is made through a few lines of glue-code, linking and OMNet++'s .ned files. The code itself, however, is in general not specialized for a specific combination of these modules. Thus, a given scenario's code simply uses predefined consistency callbacks when it changes state. Which consistency algorithm then implements those callbacks and by what strategy is normally unknown and of no interest to the scenario. While this prevents a few optimizations that are very specific to the consistency in use, it does allow us to compare different combinations of codes fairly.

There is a set of measurement tools to evaluate the quality of the system, these measurements are explained more thoroughly in chapter VII. It should be mentioned that these tools do have the ability to look at the exact internal state of all hosts at all times. On a real network, this is never possible. The measurements are used for final analysis of the test only and are not visible inside the simulation.

On each simulated host, we perform only the core calculations necessary to find the current state of the virtual world and the AI-calculations to create human-like input from what a human *would* see if this were a real host. Rendering, waiting for user input, file-system accesses, firewalls etc. are not simulated. There is, however, a mode in which the current state of the world as seen by that host can be logged. These logs can later on be used by external, scenario-specific utilities we provide to create a visualization of the simulation. The visualization can even incorporate the views of several different hosts at the same time by drawing each view in a different color on top of each other. The visualizations have proven to be very helpful in debugging.

One of the few things implicitly shared between the hosts is an exact notion of time (i.e. all hosts use OMNET++'s simTime()). This is not possible in the real world, and in fact it is somewhat difficult to keep a large set of hosts close to the wall-clock time. It is, however, *possible* to do so, at least with respect to message causality [Lam78]. The NTP protocol usually reaches a synchronization of around 10 ms ([Mil06]). We have therefore decided to abstract (i.e. ignore) from this problem.

Our code simulates a kernel system where the virtual world is advanced at a predefined frequency, just as in real DVEs. The ticks currently occur every 50ms, corresponding to 20Hz. Network requests are answered at any time and network timeouts and batching are polled for every 1ms. In a real application, the time outside of the tick would be spent mostly to render graphics and gather user-input.

Chapter IV
Consistency Algorithms

1 Loose Consistency

Loose consistency as implemented in Adam is inspired by the algorithms used in DIS and related systems [MZP+95]. For an algorithmic description, see [GSK+07].

In our implementation of loose consistency, we support regular full updates at any given frequency (note, however, that regular updates are only sent out during a tick, so it is currently not useful to choose an update frequency that is higher than the tick frequency). Note that even for a full update, messages are only sent out for objects this host owns (see below).

We also perform player/ghost analysis on all instances in the virtual world. Actions caused on the local system (a.k.a. user-input) change only the player-instance. During each tick, the Euclidian distance between the ghost-instance and the player-instance is calculated (velocity is not used). If this distance exceeds a certain, per-object-configurable threshold, an individual update for this object is sent. Whenever an update is sent for an object for any reason, the state of the player-instance is copied onto the ghost-instance.

When a network message arrives at a host, dead-reckoning is performed on the data in that message before overwriting the local state of the instance. Dead-reckoning extrapolates the state of an object from the time a message was sent to the current time. In Adam, dead-reckoning is performed by callbacks from the scenario, which will in general implement a linear approximation, but more intelligent approaches are possible.

We currently allow passive replication only, meaning each object has a distinct host as its owner-host. This owner may send updates for the object, which all other hosts will accept and use to overwrite their instances of the object. Other hosts will neither send updates, nor will the owner accept updates for the object. In some cases, when an object does not specifically belong to any host, the ownership is passed around the participating hosts in a round-robin manner, switching every few seconds or so to insure fair playout.

Out of order messages (i.e. messages for an object where a newer message has already arrived) are simply discarded in loose consistency.

The parameters that configure loose consistency are:

Regular Update Interval: The interval after which a new full update is sent out. Given in seconds.

Player Ghost Distances: If the Euclidian distance between the state of an instance in the virtual world and where this host believes other hosts believe this object to be exceeds this number, an individual update is sent. The exact number and unit of these parameters are scenario specific, but usually the unit is screen-pixels.

2 Optimistic Consistency

Optimistic consistency as implemented in Adam is inspired by some of the newer approaches to distributed game consistency as seen in [RS97] or [MVHE04]. This sort of algorithm is already in broad use in multiprocessor applications [MT01] and distributed real time data bases. For the exact algorithm, refer to [GSK+07].

In our version of optimistic consistency, we implement reliable communication via an ACK-based ARQ protocol with selective repeat. The timeout of the repeat is configurable. Using this protocol, we are assured that messages will arrive at some point, though they may be very late.

In many cases, the relevant limiting factor for netload is the amount of *packages* sent per second, not the size of those packages. To exploit this, several messages can be aggregated into one message. Our testbed supports batching, meaning the message router can collect several distinct messages (to the same host) and simply pack them into one package. As messages are not usually sent at the same time, the first message of such a batch will have to be delayed for a specified time (the batching timeout) to gather more messages with which to pack it. The longer the batching timeout is set, the more messages will usually be collected and the smaller the package bandwidth. At the same time, an additional delay is added to message sending, which will decrease the quality of the simulated DVE.

Late messages are not discarded in optimistic consistency. Rather, they are added to the message history at their intended time-slot and then the entire world-view on this host is rolled back to that time-slot and recalculated, this time including the previously missing message. The rollback mechanism is based on a so-called trailing state system synchronization scheme as described by [CFKJ02]. Currently, eight trailing states are calculated. These are spaced in decreasing density up to 25 seconds, but both the number and the spacing can be adjusted easily.

In order to fully leverage optimistic consistency's ability to arrive at the same world-view on all hosts eventually, we limit the sent-out messages entirely to *non-deterministic* information. Thus, the current state of instances is never transmitted. Only user-input (as determined by the AI) is transmitted.

We also support local lag as suggested by [MVHE04]. Actions locally decided on by the AI are sent out to other hosts normally but are not applied immediately but rather are delayed for a short time (configured in seconds). Since a human quickly adapts to such a situation by subconsciously pre-planning his input, we also gave the AI the ability to preplan, i.e. to chose the best move for a situation as it would probably play out in the future when this action will actually be applied.

Finally, a configurable reception delay exists that acts like a jitter-buffer. If a received message is younger than the configured delay, this message is delayed until it has the required age.

The parameters that configure optimistic consistency are:

Time-Out: The interval after which a message is repeated if no ACK has been received for it yet. Given in seconds.

Local Lag: The interval by which local AI-decisions are delayed before applying them. Given in seconds.

Reception Delay: The minimum age of a received message. If a message is younger than this, it is delayed until it has reached the required age. Given in seconds.

Batching: How long to delay sending a message in order to collect other messages to combine into a single package. Given in seconds.

3 Ideal

For comparison purposes, we also provide an ideal consistency, in which all variables are kept consistent on all hosts. To achieve this, we need to set the network delay and message drop rate to zero. This is not achievable on a real network, the corresponding consistency is only used as an upper-bound comparison.

Chapter V
The Replication/Network Model

OMNet++ itself provides rich functionality to connect different simulated hosts via simulated connections. We use these connections as abstractions to the entire network between the two hosts. While this keeps the model very general and easy to use, one of the things we thereby give up is congestion modeling on the network. We *do* measure network traffic (both average and maximum) on each simulated host, though, so it is possible to catch possible congestion situations a posteriori. As long as the traffic generated by the considered DVE would not be a substantial part of the overall traffic on the Internet, this abstraction is sane.

At the time of this writing, we implemented a flat connection model, every host automatically knows and is connected to every other host. Such a model is only feasible for a small numbers of hosts. Our next step in the development of Adam is therefore a dynamic connection model where each host has to keep his own list of other known hosts and may create or destroy connections to such hosts (see chapter X).

The currently implemented delay distributions for messages are

Pareto: The delay distribution has a probability density of the form:

$$f(x) := \begin{cases} 0 & x_{\min} > x \\ \frac{k \cdot x_{\min}^k}{x - x_{\text{shift}}^{k+1}} & x_{\min} \leq x \leq x_{\text{cutoff}} \\ 0 & x > x_{\text{cutoff}} \end{cases}$$

, which is a version of the standard Pareto function except with an added cutoff.

The parameters are fully configurable, but in general, we set $x_{\text{cutoff}} = 5\text{s}$ and the shape parameter $k = 1.5$. The last free parameter, $x_{\min}$, is not usually configured directly but rather calculated from the requested $E =< f(x) >$ (the expectation value of $f(x)$).

Exponential: The delay distribution has a probability density of the form:

$$f(x) := \begin{cases} \frac{\exp \frac{-x}{E}}{E} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

, where E is the configured parameter for the expectation value of $f(x)$.

Spike: Our measurements of actual Internet pings [KW07] showed that neither of the above two distributions is a particularly good fit. Therefore, we also provide an Internet modeling distribution with the cumulative probability distribution:

$$F(x) := \begin{cases} 0 & x < T \\ 0.99 & x = T \\ 0.99 + \frac{x-T}{20\text{ms}} \cdot 0.0099 & T < x < T + 20\text{ms} \\ 0.9999 + \frac{x-T-20\text{ms}}{1.980\text{s}} \cdot 0.0001 & T + 20\text{ms} \leq x < T + 2\text{s} \\ 1 & x >= T + 2\text{s} \end{cases}$$

, where T is a threshold, which is usually calculated from the requested $E =< f(x) >$. This model represents an almost certain fixed delay with an improbable, but existing, long tail.

In addition to the delays specified above, there is a configurable chance a packet may be lost altogether. On the modern Internet, this chance appears to be rather small, no more than 1 ‰ [KW07].

Chapter VI
The Scenarios

4 Pong

The first scenario we implemented was Pong, akin to the famous video-game from 1972. A rough computer model of tennis, two players each move a racket with the ability to deflect the ball. If a player fails to catch the ball with his racket and the

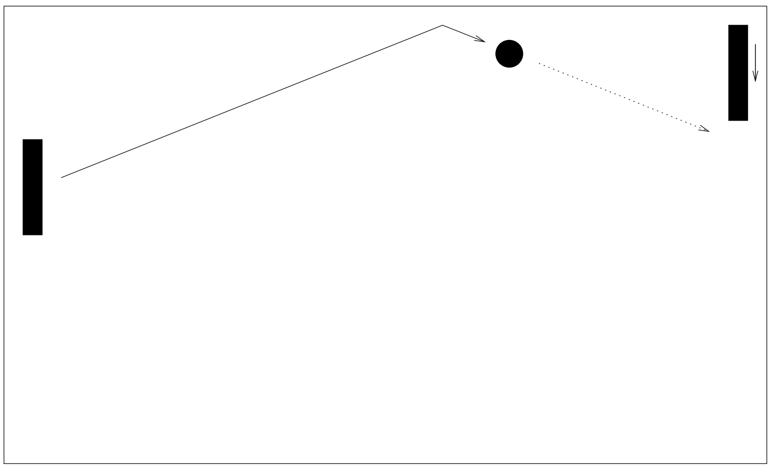

Figure 2: Visualization of Pong
The ball just bounced off the top sideline. Consequently, the right racket just started to move down to intercept it.

ball moves across his baseline, the other player scores a point and the ball is tossed back in.

The drawing in figure 2 should give you a rough idea about how the game works. Note that there are also actual visualizations (videos) of the game at work [1].

Compared to the original Pong, our version has two complications. First, the ball accelerates by 5% every time it is reflected by a racket. Second, the rackets have 3 zones. If the ball hits the middle third of a racket, it is reflected normally, but in the upper (lower) third, it is deflected more towards the top (bottom, respectively).

When passive replication is used (for loose consistency), each player's score is owned by his opponent for fairness reasons. The ball's ownership changes over time, the owner is always the host that the ball moves away from. Thus the ball is never owned by the host trying to catch it with its racket.

The scenario-specific yield-measure for Pong is points per second. The motivation is that the better the DVE is, the *less* points will be scored, as it is easier to catch the ball with the racket. If the DVE' quality gets worse, the real position of the ball is sometimes not known, and even if it becomes known, it may be too late to catch it, resulting in a score for the opponent.

The AI is kept very simple in this scenario, whenever the ball is moving towards a racket, the racket will move to the expected collision point, *not* taking into account whether the ball will bounce off the top or bottom wall. When the ball does bounce off, the racket starts to move again to try and intercept at the new expected collision point. While this simple strategem does cause a lot of unnecessary racket movement, it is a fair model of human players, who often readjust after a bounce.

[1]`http://www.math.tu-berlin.de/condel/visuals/`

5 Swarm

The Swarm models a non-cooperative game for N players, each controlling a bee and trying to stay as close to the queen-bee as possible. Every tick, honey is awarded to each bee depending on their distance to the queen, with higher distance getting less of a reward. When a bee collides with another or with the queen, the bee is penalized by being considered dead for a few seconds. While dead, a bee receives no honey and cannot control its flight.

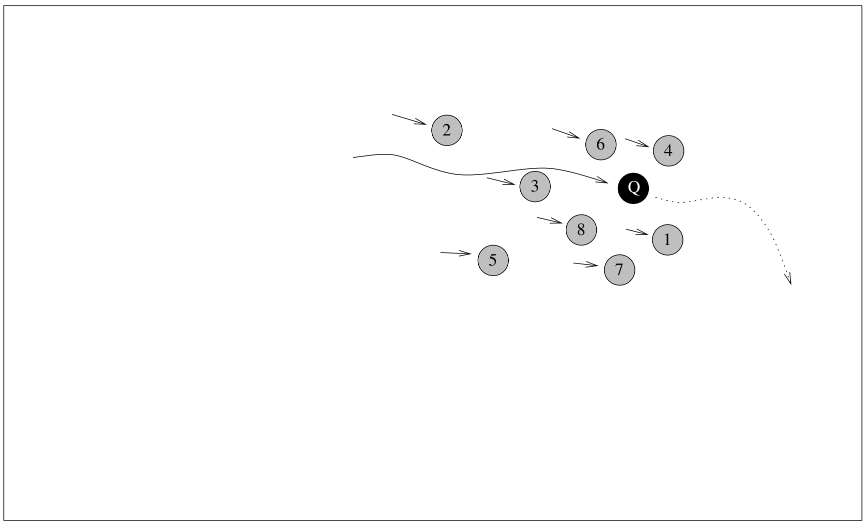

Figure 3: Visualization of Swarm
The queen (the circle with the Q in it) does a random walk, while all players steer their bees to be as close to the queen as possible without colliding with anything. The bee carrying the number 1 belongs to host 1 (and therefore, player 1) etc.

There are eight bees/players in the drawing in figure 3, plus the queen (marked with a q). Again, note that there are also actual visualizations (videos) of the game at work [2].

The game employs a physical model. Players cannot simply set position or speed of their bees, only the direction and amplitude of acceleration. Collisions between bees/queen/walls are resolved as they would for ideal, fully elastic balls of the radius shown in the screenshot. In particular a crash often leads to more crashes as the bees bounce on uncontrollably.

When passive replication is used (for loose consistency), each bee's score is owned by the next bee for fairness reasons. The queen's ownership is passed round-robin every two seconds.

The scenario specific yield-measure for Swarm is honey per second. If the DVE is good, it may be feasible for the individual bees to move quite close together to gather more honey. If the DVE is not as good, it will frequently misinform a host (and its AI) as to the other bees' and the queen's true position. This either results in more frequent crashes or the bees will have to stay further away from each

[2]http://www.math.tu-berlin.de/condel/visuals/

other and the queen in order to avoid those crashes. Either way, less honey will be collected.

The AI for this scenario is considerably more complicated than Pong's. A simple version of a potential-field steering is installed which attempts to solve the trade-off between trying to move near the queen while trying to keep a safe distance from other bees and the queen, so as not to collide with anything. A human player would quickly adapt to the game and simulation quality and adjust his safety margins accordingly. Therefore, the AI was also outfitted with a simple learning mechanism to tune its safety margins.

Chapter VII
Measurements Tools

As the purpose of the code is to rank different solutions, a good array of measurement tools is provided to compare the solutions.

6 Divergence

This measure estimates how similar the world views are on the different hosts.

For each variable a, we define as the divergence measure at time t

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

where $v_{a,h}(t)$ is the value of variable a on host h, N is the number of hosts, and the average value of variable a is

$$\overline{v_a}(t) := \frac{1}{N}\sum_h v_{a,h}(t)$$

. Thus, $d_a(t)$ is simply the standard deviation of that variable across the entire distributed system. Note that this formula is the simplification of the formula (3.1) in [GSK$^+$07] for total replication, i.e. all $r_{a,h} = 1$. In addition, we calculate the system's total divergence at time t

$$d(t) := \sum_a \frac{w_a}{\sum_a w_a} \cdot d_a(t)$$

, where w_a are configurable weighting factors that allow us to emphasize the more important variables (such as spacial differences, readily visible) versus the less important variables (such as acceleration differences, which are almost invisible to the naked eye). Finally, there is a total divergence over the entire system history given by

$$D := \overline{d(t)}$$

. Since we calculate an $d(t)$ at every tick, calculation of D is simply

$$D = \frac{1}{T}\sum_t d(t)$$

, where T is the number of ticks elapsed.

The value given by D is an objective judgment of how well the world views of the different hosts coincided during the DVE's lifetime. A smaller D indicates better consistency.

We should mention that the choice of the w_j factors has a rather large impact on the final result. There is unfortunately no well-defined scheme to find good values for these factors. We strove to set these factors analogous to how a user would rate the differences, i.e. variables that are more visible (or looked at) have higher factors than those that are not readily observable.

Also, we have normalized all values to their natural domains before using them. These normalizations, like the weighting factors w_j, also allow some adjustment of the result.

7 Discontinuity

This measurement estimates how much the DVE appears to change discontinuously. The motivation is that when a host receives a network message from another host, the host has to change his own view of the world. If the current world view is visible to the user, this change is also visible. Most humans are very sensible to such effects and find them very unpleasant.

Whenever a network message causes a change in variable a on host h's world view, the change is expressed by

$$g_{a,h}(m) := |v_{a,h}^{\text{new}}(m) - v_{a,h}^{\text{old}}(m)|$$

, where m indexes the discontinuous changes. The total discontinuous change for a single variable is then

$$G_{a,h} := \frac{1}{T} \sum_m (g_{a,h}(m))^2$$

. The total discontinuous change for an entire host is

$$G_h := \frac{1}{T} \sum_m \sum_a \frac{w'_a}{\sum_a w'_a} \cdot (g_{a,h}(m))^2$$

. w'_a are weighting factors similar to the ones in chapter 6. In both of the above, the square makes sure that larger discontinuities are emphasized, just as they are by a human observer. Finally, the system's total discontinuity is

$$\overline{G} := \frac{1}{H} \sum_h G_h$$

.

$\overline{G}$ describes the average discontinuity on the entire system. Smaller values indicate a smoother play out.

8 Yield Measures

Each scenario also has scenario-specific measures to estimate the DVE's quality. These use an in-scenario success statement to measure how well the users' in-scenario goal was reached. Typical examples would be average score, or average number of bad passes per second. These yield-measures are explained as part of their scenarios in chapter VI.

9 Cost Measures

The most important costs of a DVE is network load. Our testbed automatically measures network packages (and bytes) sent/received, both average and maximum, per host and per second. In most cases, we will report the average number of packages sent, averaged again over all hosts.

Another important factor for a DVE is computational complexity, which we also measure by counting pseudo-ops while running consistency, scenario, or network algorithms. Complexities of the AI and of the OMNet++ overhead are *not* measured. While maximum and average CPU pseudo ops are measured per host, we will usually report average CPU ops per tick, averaged again over all hosts.

Finally, memory requirements of consistency, network and scenario code are measured in bytes, both average and max. Note this is done by estimating pseudo-memory for a relatively optimized version of the algorithm, not the actual memory usage by this implementation.

10 Other Measures

In virtual environments, the local reaction time is of particular importance to retain the impression of immersion. With both loose and optimistic consistency, the local reaction time is usually zero, i.e. local decisions are performed immediately. However, the *local lag* parameter we installed in optimistic consistency improves overall system consistency at the cost of also increasing local reaction time (making the local system less responsive). This is not a real measurement, as the parameter is specified to the system. However, the local reaction time, and therefore the local lag, is a very important quality measure for the system.

Chapter VIII
Experiments

Since most of the algorithms plugged into Adam come with parameters it is essential to have some way to find optimal values for these parameters given the problem. To do this, one needs to search the parameter space, rerunning the simulation with different settings and then comparing the results.

To get a better idea of what each parameter does, we currently simply scan the parameter space along the axes of each parameter, leaving all other parameters on their defaults. At particular areas of interest, we add more search-rays with altered defaults.

While it is possible to simply run the testbed on the current machine, this only gives the result of a single run. Statistic aberrations exist, so in most cases, one would want to run the experiment several times with different random seeds, averaging the results. Currently, we run each experiment seven times via Akaroa.

While Pong runs for 50.000 simulated seconds per experiment, Swarm's higher complexity limits us to 5.000 simulated seconds per experiment.

We are running Adam on standard desktop PCs. The computational complexity of the Swarm scenario is $O(N^3)$, where N is the number of hosts. For practical purposes, this limits the number of hosts that can be simulated to sixteen. The theoretical complexity of the simulation is $O(N^2)$, but Adam has not been optimized to this respect yet.

We currently run the following six standard combinations of scenarios and consistency plugins:

Pong Ideal The Pong scenario with ideal consistency. No parameters.

Pong Loose The Pong scenario with loose consistency. Configurable parameters [with their defaults] are:

- **loss rate** [0.001] ([KW07])
- **avg. delay** [0.1s] 100ms delay on all connections. This corresponds to a 200ms ping.
- **regular update** [1s] Send a full update every second.
- **ball update distance** [5] If the ball is 5 pixels away from its ghost, send an update for the ball.
- **racket update distance** [25] If the racket is 25 pixels away from its ghost, send an update for the racket.

Pong Opt The Pong scenario with optimistic consistency. Configurable parameters with their defaults are:

- **loss rate** [0.001]
- **avg. delay** [0.1s]
- **timeout** [0.25s] If no acknowledgment message is received within 0.3s after sending a message, the message is repeated.
- **local lag** [0s] Do not delay local actions.
- **reception delay** [0s] Do not use a jitter buffer.
- **batching delay** [0s] Do not wait to combine messages.

Swarm Ideal The Swarm scenario with ideal consistency.

- **nr. of bees** [8] Currently, simulate only a small amount of bees/hosts.

Swarm Loose The Swarm scenario with loose consistency. Configurable parameters with their defaults are:

loss rate [0.001]

avg. delay [0.1s]

nr. of bees [8]

regular update [0.35s]

bee update distance [5] If a bee is 5 pixels away from its ghost, send an update for this bee.

Swarm Opt The Swarm scenario with optimistic consistency. Configurable parameters with their defaults are:

loss rate [0.001]

avg. delay [0.1s]

nr. of bees [8]

timeout [0.25s]

local lag [0s]

reception delay [0s]

batching delay [0s]

Chapter IX
Results

11 Rate-Quality Characteristics of Loose and Optimistic Consistency

We defined the two quality measures divergence and discontinuity and it can be argued that two DVEs that show the same same values for these two quality measures would also have approximately the same quality. To use these measures quantitatively, we would like an intuition of what the qualities of two DVEs with *different* values in these measures are. In other words, we would like to understand by how much quality of the DVE really decreases if these measures, say, double. Since quality in a distributed system is extremely hard to quantify, it is impossible to answer this question this way. There is, however, another way to gain a quantitative understanding about these measures: How much do the quality measures increase if the network bandwidth used by the DVE increases? This information, the rate-quality characteristic, not only allows us to quantify the quality information given by these measures, it also allows an implementor of a distributed system to quickly calculate required network bandwidths for given target quality levels.

To find these characteristic curves, we measured parametric curves for our inconsistency measures over network load. For this, we stepped through a single parameter in the Swarm experiment, leaving the other parameters at their defaults. For loose consistency, we stepped through regular update rate, while for the optimistic case, we stepped through the message aggregation timeout. The resulting network load and inconsistency measurements from each sub-experiment are then plotted in figures 4 and 5 for a network delay of 100ms, in figures 6 and 7 for a network delay of 200ms, and 8 and 9 for the 400ms case.

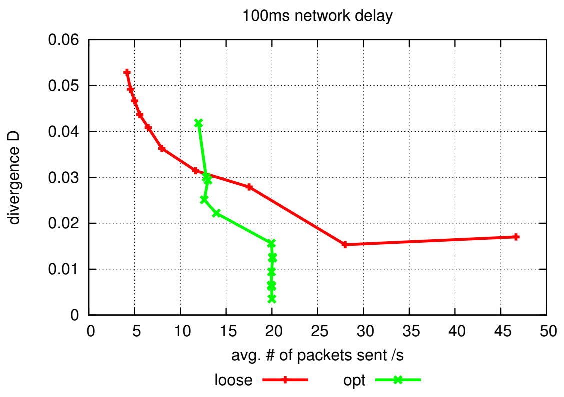

Figure 4: Divergence Characteristic for 100ms Network Delay

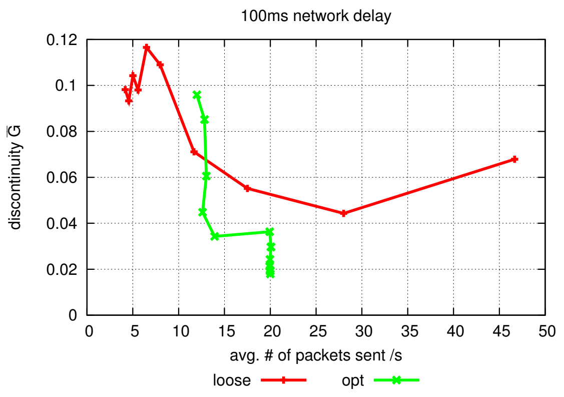

Figure 5: Discontinuity Characteristic for 100ms Network Delay

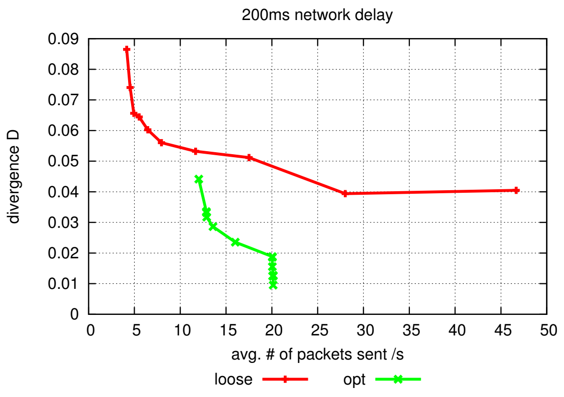

Figure 6: Divergence Characteristic for 200ms Network Delay

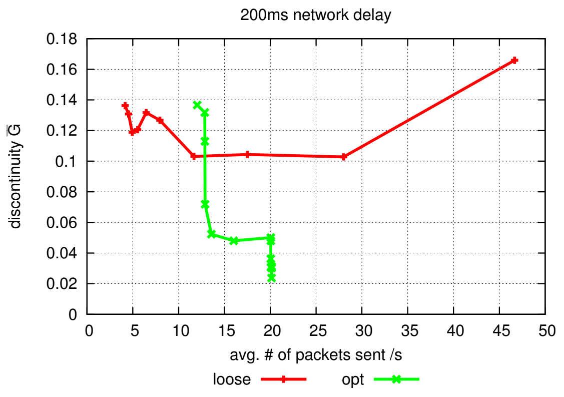

Figure 7: Discontinuity Characteristic for 200ms Network Delay

From the figures we can immediately see optimistic consistency's major downside, namely that it cannot be configured to cover a very large network load range. The network load is pretty much determined by the player action frequency, with batching only providing an effective load reduction of less than a factor of 2. Consequently, there is a certain network load below which optimistic consistency will just not work and loose consistency is therefore better by default. This point lies around 12 packets/s for swarm. This point will vary for other scenarios, user numbers, and even variant AIs.

Optimistic consistency also has a high point beyond which it cannot use utilize more bandwidth. In our setting, this point was reached at around 20 packets/s. However, the consistency at that point is already significantly better than loose's consistency at any network load.

On the other hand, optimistic consistency also yields significantly better results for the entire range of bandwidth > 13 packets/s, in all three sub-experiments.

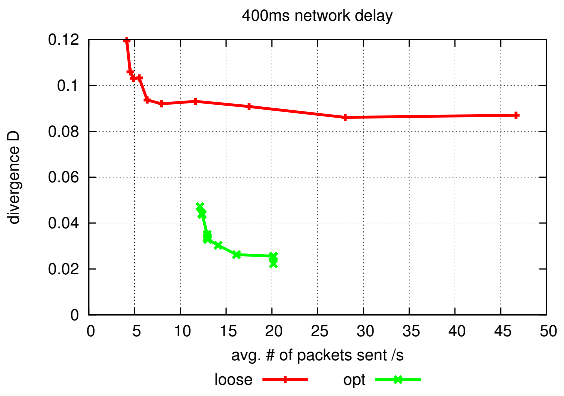

Figure 8: Divergence Characteristic for 400ms Network Delay

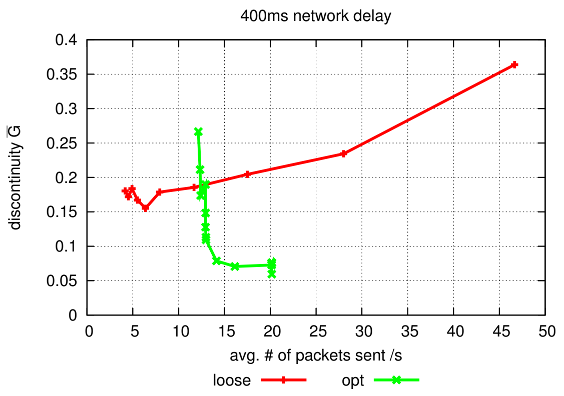

Figure 9: Discontinuity Characteristic for 400ms Network Delay

There does not appear to exist any easy fit on the curves, as the exact type of the curves depends on the consistency in use as well as the delay and the quality measure. However, divergence curves for loose consistency appear to consist of a x^{-1} part at low bandwidth connected to a saturation part where the consistency is constant.

For the discontinuity measure, loose displays an odd behavior that has long increasing branches at high bandwidth. This attests to loose consistency's inability to actually increase the quality when using more bandwidth. This is because each additional message results in an additional discontinuity, while the size of the discontinuities does not decrease similarly because of loose consistency's builtin flaws.

For optimistic consistency, the curves form a distinct spike in all cases. This hints that batching is not very beneficial in many ranges. In fact, there are usually only two points of interest. In our scenario, these lie at the no-batching point at 20 packets/s and at the 13 packets/s point. All other points either cost a lot more bandwidth for similar quality or yield worse quality for the same bandwidth.

12 Optimistic Consistency vs. Loose Consistency

Up to now, no testbed or system was ever able to directly compare the effect of loose consistency to a system employing optimistic consistency. It is easy to do this with Adam. In the following experiment, we ran two series of experiments, one with loose, the other with optimistic consistency. Both ran the *swarm* scenario with total replication.

In the experiment series, we varied the simulated network's delay time and plotted netload, divergence, discontinuity, cpu-load and honey-gathered as a direct comparison between the two methods.

Note that the parameters were chosen so that the netload of the two experiments was equal.

Again, it's obvious from the data in figures 10 and 11 that for the swarm scenario at least optimistic consistency works a *lot* better than loose consistency, The

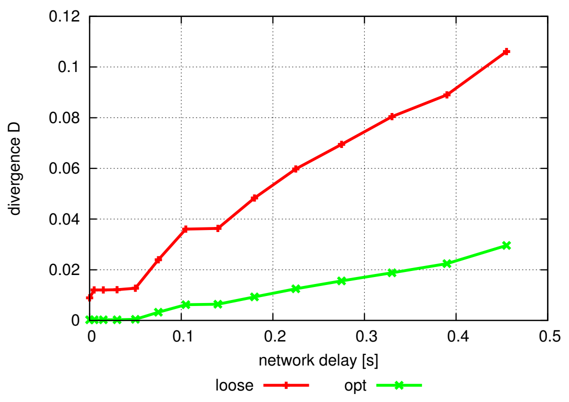

Figure 10: Divergence Comparison
at 20 packets/s

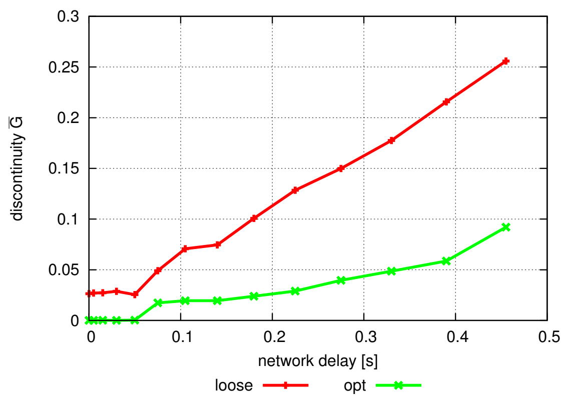

Figure 11: Discontinuity Comparison
at 20 packets/s

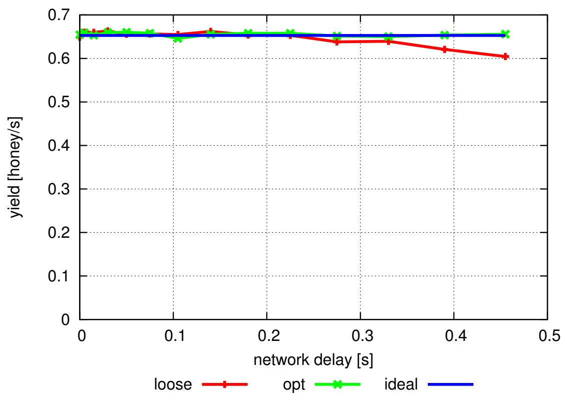

Figure 12: Score Comparison
at 20 packets/s

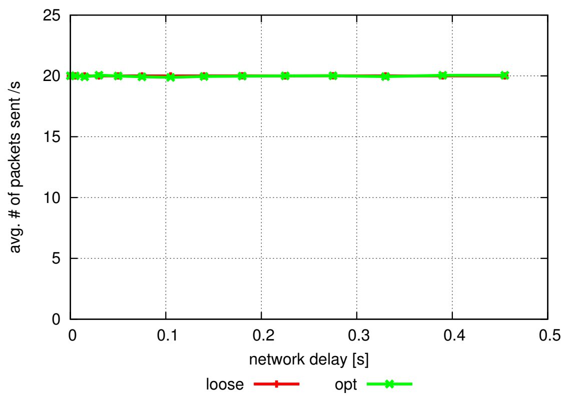

Figure 13: Netload Comparison
at 20 packets/s

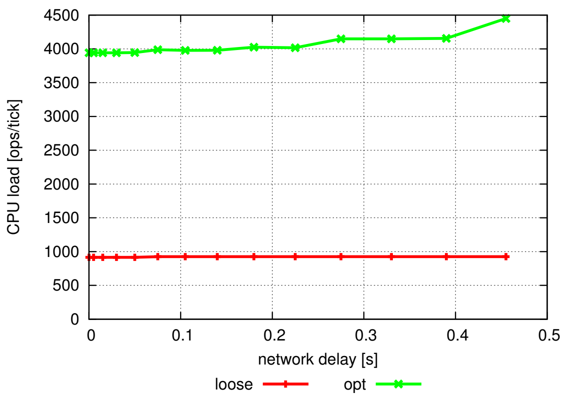

Figure 14: CPU Load Comparison
at 20 packets/s

quality is better when looking both at divergence and at discontinuity, so there does not seem to be a user-visible drawback. Optimistic consistency rates about five times as good as loose consistency. Both divergence and discontinuity behave roughly linear in delay, therefore an improvement in these values by a factor five has a similar effect as if the delay was reduced by a factor five, which signifies an extremely strong improvement. A more telling comparison would be to set the parameters so that loose and optimistic consistency perform equally well in our quality measures and then compare their bandwidth. As one can see from the figures in chapter 11, there simply is no amount of bandwidth that will allow loose consistency to perform as well as optimistic consistency without batching because of the saturation branch.

The parameters were adjusted beforehand so that both experiments would utilize the same amount of netload (i.e. send the same amount of packages, see figure 13). The CPU-load for calculating the world-state is, however, noticeably higher (about four times as high, see figure 14) for the optimistic case.

From the graphs, one can also see that the yield-measure (i.e. average honey gathered) is unfortunately not a very good meter for quality, as the bees' good AI makes up for bad consistency in the system and lets the bees gather more or less the same amount of honey regardless of the DVE's quality.

13 Local Lag

Delaying local actions is an effective method of increasing consistency for distributed systems in general but for optimistic systems especially. The delay gives other hosts a chance to receive the message describing this action before the action is due, which will often result in the action being performed simultaneously on all hosts. This method, termed *local lag* [MVHE04] is often used in one form or another (e.g. [DG99]) and is reported to be quite good.

We have run the swarm scenario with optimistic consistency and total replication. The experimental series was done with the delay distribution spike (see chapter V), an average delay of 100ms, a loss rate of 1 ‰, and a timeout for the reliable ARQ algorithm with a 250ms timeout. Eight bees were simulated, no batching or reception delay was used.

As one can see from figures 15 and 16, local lag really is as effective as cited previously by other groups. Use of local lag of about the average network delay time cuts both discontinuity and divergence by a factor of seven. Since both divergence and discontinuity behave roughly linear in delay (see figures 10 and 11), this means that a locally-lagged DVE behaves similar as if the network delay was cut by seven. Comparison with the graphs in chapter 11 shows that a similar effect could also be achieved via (roughly) doubling network bandwidth. As noted in chapter 11, however, there is a maximum bandwidth for optimistic consistency beyond which additional bandwidth cannot be utilized any more. In the experiment in figures 15 and 16, this maximum bandwidth is reached already.

Note that in Adam, actions are only created and evaluated during a tick. Since ticks are spaced 50ms apart, using a local lag of any value above 50ms will usually cause the action to be evaluated two ticks later, at which time it will have just been

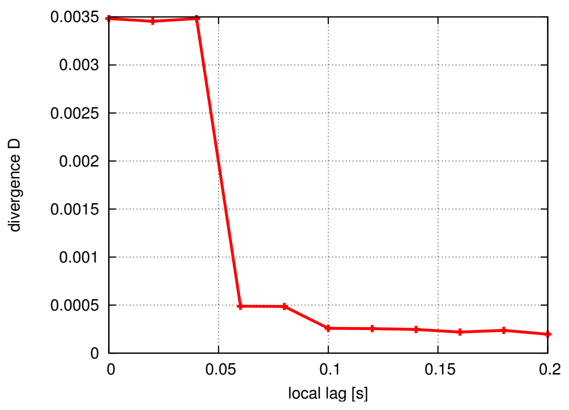

Figure 15: Divergence under Local Lag

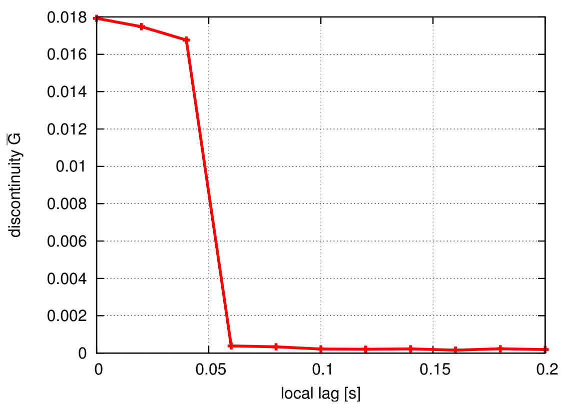

Figure 16: Discontinuity under Local Lag

received if the average delay is 100ms.

It should be pointed out here that local lag is still not the solution to all problems, as the resulting decreased local reactiveness may not be suitable to the scenario. For virtual environments, the maximum allowable local lag has been reported as between 100ms and 225ms ([Hen01]).

Chapter X
Further Work

The biggest challenge in the industry and scientific research on the subject of virtual environments is currently the question of how to create a truly scalable solution, i.e. one where millions of users can interact at the same time. To reach this goal, the complexity and amount of network messages both have to scale either by $O(1)$ or perhaps $O(logN)$, where N is the amount of hosts participating in the DVE. There are already a number of solutions regarding this subject. Our testbed was always intended to benchmark these solutions, as well.

To allow us to test these kinds of solutions, partial replication has to be implemented, so each host no longer replicates the entire world, but rather just a small part of it (e.g. the area within the view-range of his avatar). This requires objects to dynamically join and leave a host's world view and allows independence of the total world size in calculation complexity. Changes in world state need only be messaged to other hosts nearby in the virtual world, which allows independence of the total population size in network costs.

A prerequisite to achieve such a solution is that each host only knows a subset of other hosts (no host can know all other hosts), which may change over time. Thus hosts must be able to create and destroy connections to new hosts. This requires an extension to our currently flat network model.

When these preliminaries are done, we are going to install a few of the more important algorithms running partial replication schemes. These range from classic multicast approaches to the latest peer to peer systems. Our goal is to implement the different replication schemes in the form of pluggable add-ons, just like everything else in Adam. This would allow us to compare any combination of replication scheme and consistency algorithm for a given scenario.

There is a difficult sub-problem involved with the combination of partial replication and optimistic consistency, if exploiting determinism. Basically, what appears deterministic to one host may seem non-deterministic to another. For example, suppose host A replicates object X, which is influenced by an object Y that host A does *not* replicate. The host B replicating object Y must then send a message about this interaction to A. On the one hand, it may not be very easy to even notice when this situation occurs. On the other hand, the message sent by B appears deterministic to B. Due to the optimistic playout, it may later be necessary to invalidate the deterministic messages by anti-messages. One must take great care to prevent anti-message avalanches in such a situation.

In another subfield, it is possible to define an objective target function Z by something akin to $Z(\eta) = Q(\eta) - K(\eta)$, where Q would be a measure of quality, while K would be a measure of cost. It is then possible to find a local or even global maximum of Z in the parameter space using the well-known array of nonlinear numerical optimization techniques. Thus, one could quickly and automatically find the "optimal" parameter settings for a given combination of scenario, consistency, and network model. This has not yet been implemented. The problem is that we currently have no agreed-upon Z-function. This function would have to include most of the quality measures we defined in chapter VII and their weighting to each other. Similarly, all costs have to be included and weighted. This is very subjective and we have not yet found a general approach to define Z.

Lastly the currently used timeouts are configured via a single, fixed number. Shorter timeouts mean increased traffic and therefore higher network costs. On the other hand, shorter timeouts cause shorter overall delay and thus better consistency. There is a tradeoff between these two factors whose solution depends on the given problem, i.e. the allowed network capacity, the required quality, etc. However, if the messages are not all of equal importance, the optimal tradeoff varies from message to message. We are preparing a paper providing the tools, functions, and approximations to find a near-optimal tradeoff for every message.

Chapter XI
Conclusions

We have introduced Adam, a testbed to estimate the quality of a distributed virtual environment. We have shown that this testbed is practical and can be used to directly compare algorithms with each other, leaving other algorithms in place.

To achieve this, we first had to define quality measures for distributed virtual environments. We then implemented various consistency schemes, scenarios, measurements and standard experiments.

We used these experiments to show that optimistic consistency performs a lot better than loose consistency except at extremely low network bandwidths. Also, local lag is a very efficient way to improve optimistic consistency's performance further.

References

[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.

[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.

[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.

[EPM99] G. C. Ewing, K. Pawlikowski, and D. McNickle. Akaroa2: Exploiting Network Computing by Distributing Stochastic Simulation. In *Proc. European Simulation Multiconference ESM'99*, pages 175–181. International Society for Computer Simulation, June 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.

[GSK$^+$07] Sven Grottke, Jan Sablatnig, Andreas Köpke, Jiehua Chen, Ruedi Seiler, and Adam Wolisz. Consistency in distributed systems. Technical Report in preparation, TU-Berlin, 2007.

[Hen01] Tristan Henderson. Latency and User Behaviour on a Multiplayer Game Server. In *NGC '01: Proceedings of the Third International COST264 Workshop on Networked Group Communication*, pages 1–13, London, UK, 2001. Springer-Verlag.

[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.

[KW07] Andreas Köpke and Adam Wolisz. Delay Measurements on the Internet. Technical Report in preparation, TU-Berlin, 2007.

[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.

[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.

[RS97] David J. Roberts and Paul M. Sharkey. Maximising concurrency and scalability in a consistent, causal, distributed virtual reality system, whilst minimising the effect of network delays. In *Proceedings of the IEEE Workshops on Enabling Technology: Infrastructure for Collaborative Enterprise '97*, pages 161–166, 1997.

[Var01] András Varga. The OMNet++ Discrete Event Simulation System. In *Proceedings of the European Simulation Multiconference (ESM 2001)*, June 2001.