DISTRIBUTED SHARED MEMORY FOR VIRTUAL ENVIRONMENTS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,
FACULTY OF SCIENCE
AT THE UNIVERSITY OF CAPE TOWN
IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Andrew Godfrey
December 1997

Supervised by
E.H. Blake and K.J. MacGregor

Abstract

This work investigated making virtual environments easier to program, by designing a suitable distributed shared memory system. To be usable, the system must keep latency to a minimum, as virtual environments are very sensitive to it. The resulting design is push-based and non-consistent.

Another requirement is that the system should be scaleable, over large distances and over large numbers of participants. The latter is hard to achieve with current network protocols, and a proposal was made for a more scaleable multicast addressing system than is used in the Internet protocol.

Two sample virtual environments were developed to test the ease-of-use of the system. This showed that the basic concept is sound, but that more support is needed. The next step should be to extend the language and add compiler support, which will enhance ease-of-use and allow numerous optimisations. This can be improved further by providing system-supported containers.

Contents

List of Figures

Chapter 1

Introduction

1.1 Aims

The primary goal of this research is to make distributed virtual environments easier to program. This will be explained in detail in the next section.

This work also has the following secondary goals, in order of decreasing priority:

1. **Scalability:** The system should scale to large numbers of users, and it should scale over large distances.
2. **Portability:** The system should be portable to multiple architectures, and machines of different architectures should be able to interact in the same environment.
3. **Efficient bandwidth use:** Where possible, the system should not waste network bandwidth.

1.2 Virtual Environments

A virtual environment, as discussed in this work, is a distributed simulation of a *virtual world* in which a number of human *participants* may interact.

This definition includes:

- **Multiplayer games:** Game environments let a number of people compete against, or play with, each other. There is a wide range of possibilities — action games, sports, card games, and even text-based adventure games can be the basis for a virtual environment.
- **Collaborative education environments:** These allow students to collaborate in simulated environments which would otherwise be unreachable, uncomfortable or dangerous. Another use is to allow students to meet and collaborate when they are physically distant from each other.
- **Cooperative work environments:** This includes such applications as distributed product design and teleconferencing.

Virtual environments from any of these areas can choose from numerous user interfaces, such as a text interface, a graphical user interface or a three-dimensional view of the world (virtual reality).

Any virtual environment should present an up-to-date view of the world. A significant obstacle to this is *interaction latency* (or just *latency*) — a delay between one participant doing something and the displays of other participants showing that action.

Increasing the latency of a virtual environment decreases its quality. Some types of virtual environment (such as text-based ones) might tolerate higher latency than others, but all are susceptible to it. As will be shown, latency has a positive, significant lower bound, and can be increased substantially through poor design choices.

It is important, therefore, that a system which eases virtual environment programming does so without increasing latency.

1.3 Approach

The approach taken implements a *distributed shared memory* (DSM) [4, 9, 48, 32, 60, 34]. This programming paradigm simulates a memory system which can be accessed by all participating processes.

DSM is a natural choice. Virtual environments simulate a single, shared world. Participants are presented with a view of the most up-to-date *world state*. Thus, all participating machines need access to the world state, and the intuitive way to store this state is in distributed shared memory.

Indeed, virtual environments written in an ad hoc manner have features similar to shared memory systems. Yet ready-written distributed shared memory systems cannot be used for virtual environments, because they produce high latency.

Therefore, we examined the causes of latency in DSM systems, and used this insight to design a low-latency DSM system, DSMVE. We used this to write two sample virtual environments, in order to test the ease-of-programming of the system.

1.4 Overview

The rest of the thesis is arranged as follows. Chapter 2 gives background information. It begins by describing network issues, such as communication latency, multicast, and bandwidth. Next it discusses virtual environments, including interaction latency and ownership. Finally it gives background for distributed shared memory, mentioning granularity, consistency and atomicity.

Chapter 3 (Theory) first shows the decisions taken to produce the design for DSMVE. The key properties of this design are push-based distribution and non-consistency. The next section gives recommendations for an improved multicast addressing system which would greatly enhance scalability, and the last section examines polling and its effect on latency.

Chapter 4 details the implementation of DSMVE. The first section describes the programmer's interface to the shared memory system, while the second shows how that interface was implemented. Section 4.2.3 suggests associating a *latency tolerance* value with each object, which can be used to lower bandwidth use.

Chapter 5 covers the two sample applications. It describes each application in turn, and then discusses the experience gained from these applications. In particular, this showed how some graphical user interface libraries can make it more difficult to use

a DSM library, because they have their own storage for the state of interface elements. The chapter concludes by identifying the limitations of DSMVE, and giving suggestions for future improvement.

Chapter 6 summarises the results of the thesis. It shows to what extent the aims were satisfied, by giving a section on each one. The most significant shortcoming is that scalability was hampered by poor multicast support, which is why a proposal for better multicast has been made (in Chapter 3).

Chapter 7 gives an overview of the thesis, and summarises the results. DSMVE is fairly easy to use, portable, and fairly efficient. It is not as scalable as it could be, because of the lack of an explicit multicast protocol. Finally, the chapter gives directions for future work — system-supported containers, compiler support and persistence.

Chapter 2

Background

2.1 Networking

Research and development in network performance usually concentrates on bandwidth — lowering the load on the network, or increasing the capacity of the network. There is a growing consensus that not enough attention is paid to *latency* — communication delay [15, 62].

> The entire focus of the industry is on bandwidth, but the true killer is latency. [15, quoting M. Satyanarayanan]
>
> It is high throughput, and not low latency, that has been the target of most newer networks and controllers. [62, page 201]

The result is an industry which uses bandwidth as its primary performance measure — which is akin to the automobile industry using traffic flow rate as its performance measure. What might also perpetuate this is the fact that available bandwidth can be increased arbitrarily, but reducing latency is challenging [25].

The industry's lack of concern over latency has produced some questionable designs. For example, when connected to the Internet using a typical modem, the modem incurs a completely unnecessary 50ms latency on each packet sent [15].

The following sections examine various network topics and their effect on latency.

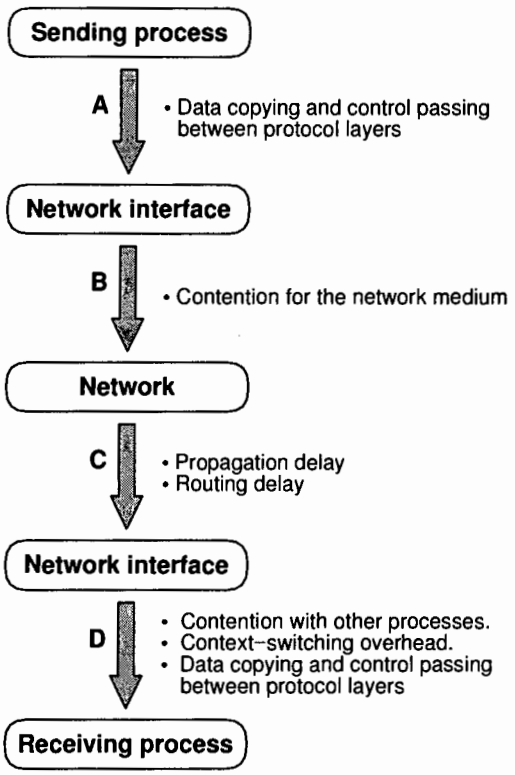

Figure 1: **Communication latency.** This diagram shows a message between two processes on different hosts. For each stage, it lists some activities which contribute to latency.

2.1.1 Communication latency

This work will mention a few different types of latency (e.g. the latencies of a network message, an operation, or interaction between users). They are all related. For example, the latency of an operation composed of network messages depends on the latency of a network message.

For this reason, the latency of a network message — the time taken to communicate a message from one process to another — will henceforth be referred to as *communication latency*. Figure 1 illustrates the latency sources which can contribute to communication latency.

For widely-distributed systems, the factor which dominates communication latency is *propagation delay* — the time taken for the message to travel to the destination. The speed-of-light limit implies a lower bound on propagation delay which depends on the

distance between hosts. For two hosts on opposite sides of the planet, the absolute minimum propagation delay (using a vacuum tube drilled through the centre of the earth) is 43ms.

Cheshire [15] derives a more conservative limit, through glass fibre over the earth's surface, of 100ms. He notes that for certain well-connected Internet hosts spanning the United States, the observed communication latency is within a factor of two of the theoretical limit for that distance.

To conclude, hardware advances will eventually not be able to improve communication latency. It is therefore clear that software decisions which increase latency should be avoided. This is a key difference between communication latency and bandwidth. Software decisions which increase bandwidth use are acceptable, because available bandwidth is increasing all the time, apparently without bounds [12, 33].

2.1.2 Message chains

In many cases, a single higher-level operation comprises multiple messages which are causally related — that is, the sending of one message relies on the receipt of another. The result is a serial 'chain' of messages. Latency is additive, so the latency of the operation is the sum the latencies of each message. This includes the communication latency and the time taken to process each message.

An example is the remote procedure call (RPC), shown in Figure 2. The latency for an individual RPC operation is the sum of the latencies for the request and reply messages, each of which has a communication latency component and a processing time component.

Another example is the use of a separate 'communications' process on each host (Figure 3) [10, 63]. This process performs all inter-host communication on behalf of other local processes. For each end-to-end message, the average latency is increased by twice the inter-process communication (IPC) latency. The impact of this depends on the IPC latency of the architecture and operating system, as well as the load on the processor.

A widely-used technique to increase the rate of work of an enitity (in operations per second) is *pipelining*. It works by allowing different stages of different operations

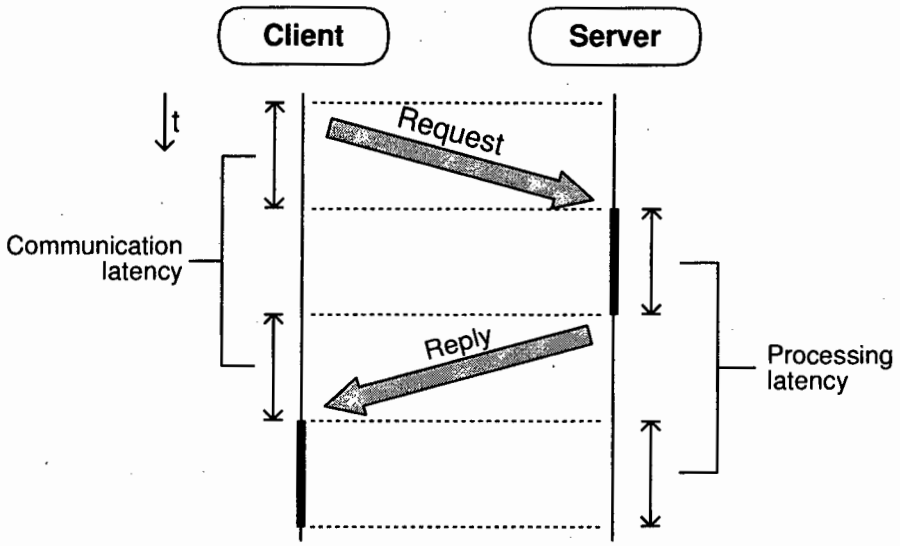

Figure 2: **Remote procedure call.** This figure shows the causally-dependent messages used in a remote procedure call. If a reliable protocol like TCP is used, there will also be acknowledgement messages for each message shown here, but the RPC operation does not necessarily wait for them.

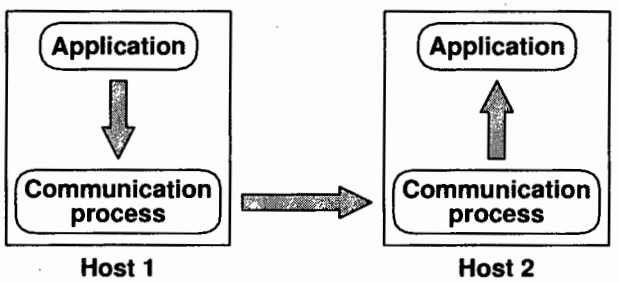

Figure 3: **The 'communication process'.** If a separate process is used to perform all network communication, latency is higher. Compared to direct communication, latency is increased by twice the inter-process communication latency.

to overlap in time. One example is asynchronous RPC, which allows processing to continue while an RPC call is in progress.

The key point is that pipelining does not reduce the latency of each operation. In order to reduce the latency of an operation, one must reduce the communication latency, the processing latency, or the number of causally-dependent messages which comprise the operation (the 'length' of the chain).

2.1.3 Multicast

Traditional networking has focused on point-to-point, or *unicast* communication. Such communication involves only two hosts — a sender and a receiver.

Multicast communication allows a single message to be sent to a group of hosts. Current protocols — TCP/IP and ATM — provide *subscription* multicast. That is, the recipient group for a message is identified by a multicast address, which interested parties use to subscribe to the group.

If the underlying network protocol does not support multicast, the application programmer must implement it using repeated unicasts.

The advantage of protocol-supported multicast over repeated unicasts is considerable: If n hosts are in a multicast group, one message sent to that group accomplishes the work of n unicasts. Thus there is a reduction in latency and bandwidth use, but the extent of this depends on how the multicast protocol itself is implemented. For example, the multicast protocol may itself be implemented using repeated unicasts. In this case, the only advantage of using the protocol-supplied multicast is ease of programming.

Multicast on a local network

Regardless of the physical network used (e.g. Ethernet, token ring), one often finds the Internet broken into small groups (around 10 to 100 machines) which share a single network resource (e.g. the Ethernet cable, or a connecting hub). These groups are connected together with routers and gateways. We first consider a group of machines on a single Ethernet. Figure 4 illustrates the following discussion.

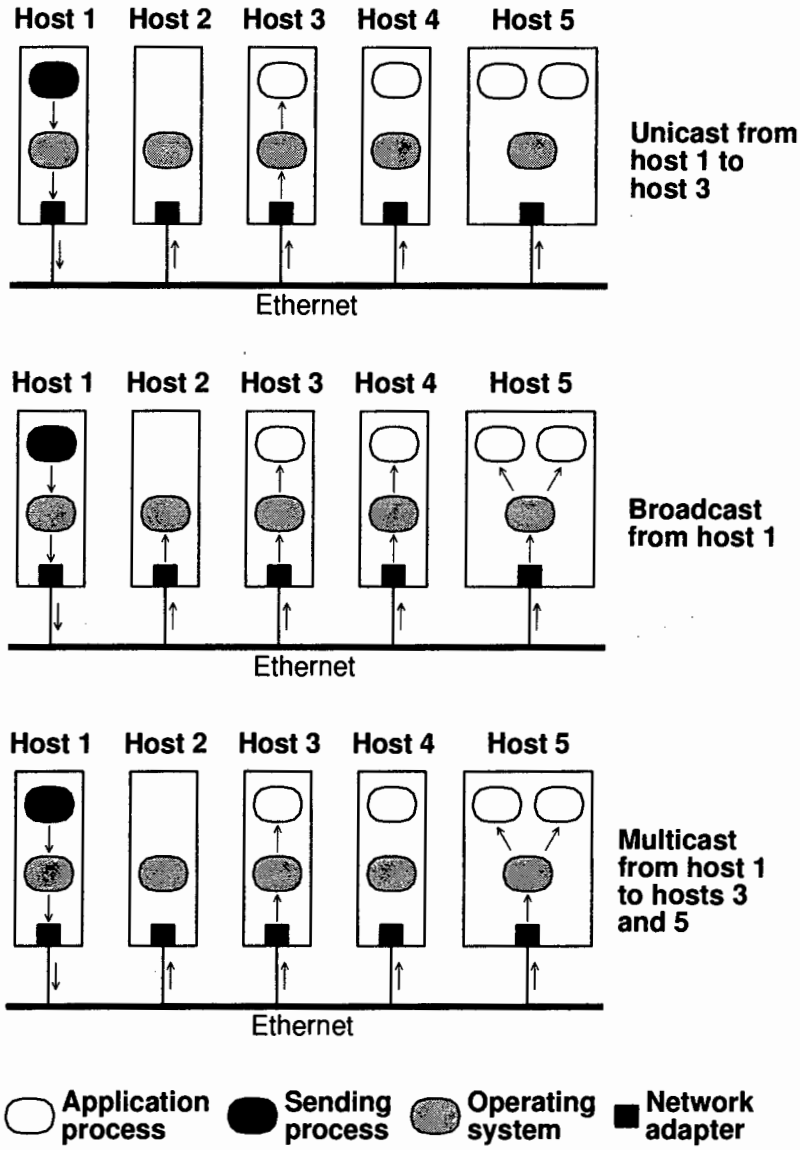

Figure 4: **Unicast, broadcast and multicast on a local network.** The application is running on hosts 1, 3, 4 and 5 (with two instances running on host 5). On an Ethernet, every packet sent is visible to all network adapters. With unicast addressing, other adapters ignore the packet. With broadcast addressing, every host is interrupted as its adapter forwards the packet to the operating system. (This includes hosts not even running the application, like host 2 in the diagram.) With multicast addressing, only hosts of subscribed processes are interrupted.

Each network adapter has a uniquely-assigned address.[1] Network hardware on each machine examines the address field of each packet on the network. If the hardware recognises an address, it interrupts the host, and delivers the packet.

In normal unicast, the sender marks the packet with the address of the destination host. The destination host's network hardware recognizes this address and delivers the packet to its host, while network hardware on any other host which sees the packet ignores it.

Ethernet, like the higher-level TCP/IP protocol, uses a subscription multicast scheme. A portion of the available address space is assigned to multicast addresses, and network hardware can be configured to recognize a particular multicast address in a packet's address field. The sending hardware marks the packet with the multicast group address, and all hosts subscribed to that group receive the packet.

One special multicast address is the *broadcast* address. All adapters recognise this address automatically, so a broadcast packet is received by all hosts. Since all hosts are interrupted, including those not subscribed to the multicast group, a multicast operation should not be implemented using broadcast.

For a multicast operation to n machines, an implementation using unicast will need n times as many packets as one which uses hardware multicast — i.e. hardware multicast uses less bandwidth by a factor of n.

The latency of a multicast is lower than that of the equivalent n unicasts, but a quantitative comparison is difficult and depends on many factors. The total propagation delay is n times lower,[2] but on a local network, propagation delay can be a small part of the total message overhead. The reduction in traffic from the use of multicast may also reduce contention for the network. This also lowers latency, but again the degree of improvement depends on the type of network and the load on it.

Multicast routing

[1] IP addresses are distinct from these addresses; there is a conversion protocol to map between them.

[2] This is only true because the propagation step of the n unicasts cannot be pipelined on a local Ethernet.

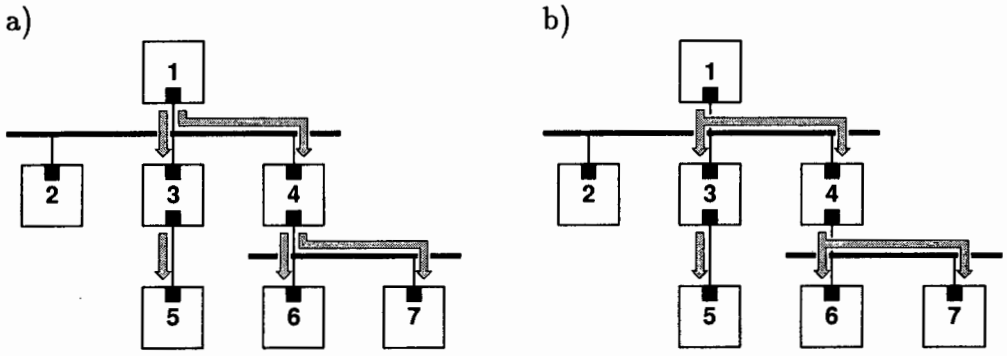

Figure 5: **Multicast routing.** Host 1 sends a message to hosts 3, 5, 6, and 7. Hosts 3 and 4 act as routers. The paths to hosts 6 and 7 overlap, so only one message needs to be sent on the first link (from 1 to 4) to cater for both. In figure (a), hardware multicast is not used, so 5 messages are needed. Figure (b) shows the same situation, but using hardware multicast, in which case only 3 messages are needed.

Multicast over the Internet is more complicated because of lower connectedness. Clusters of local networks are connected to others typically by only a few *links* — routers and gateways.

Whenever the paths from the sender to the receivers overlap, multicast distribution saves on bandwidth as the packet is only sent once across each overlapping link (Figure 5a). In addition, if the multicast implementation can use hardware multicast at each stage, two links in a path can be considered 'overlapping' if they are on the same local network (Figure 5b).

The latency for a packet traversing more than one link is additive — the total latency is the sum of the latencies for each link. For each overlapping link, repeated unicast requires a number of messages to be sent over that link, which will serialise the messages to some degree, depending on the link itself.

This serialisation is only imposed on messages queueing for an individual link. Repeated unicasts traversing a path of multiple links are pipelined, so that in the best case, the latency of a repeated unicast is only marginally higher than that of a protocol-supported multicast. However, contention at a link will have a greater effect on the latency of repeated unicast, making it more variable than protocol-supported multicast.

In conclusion, while a quantitative comparison is difficult because of the many dependent factors, it is clear that multicast distribution can reduce bandwidth use substantially, and latency to some degree [4, 10, 12, 25, 52].

2.1.4 Message grouping

TCP/IP packets have variable size, up to a certain maximum. Because each network packet has a fixed header size independent of the packet size, larger packets are more efficient. That is, when fragmenting a block of data into packets, it is better (from a bandwidth point of view) to break the data into a few large packets than many small ones.

The implication of this is that it is efficient to group a number of small messages so that they can be transmitted in a single network packet. However, this can increase the latency of the messages if the system buffers messages waiting for more.

2.1.5 Other issues

Many popular network protocols have features which are not suited to real-time systems. This includes TCP and ATM. The following sections describe these features.

Reliability

Reliable protocols ensure that, once a message has been issued to the protocol, it is delivered to its destination.

Reliability at the protocol level is inefficient for real-time systems. A reliable protocol assumes that a message should be delivered no matter how long delivery takes. But often in real-time systems there are sequences of update messages of which only the most recently-sent is of interest. If a packet is delayed for some reason, it should be discarded if a more up-to-date equivalent is available [25, 41].

Since a reliable protocol has no knowledge of the semantics of the messages it delivers, it cannot provide this functionality. Thus, if this issue is significant, reliability should not be implemented at the protocol level.

Total and causal ordering

Totally ordered and *causally ordered* protocols ensure that messages arrive at their destinations in the same order as they were issued, or in an order which preserves causal dependencies between messages, respectively. TCP and most of the ATM protocol are totally ordered.

Cheriton and Skeen [13] give a thorough discussion of 'Causally and Totally Ordered Communication Support' (CATOCS), and find that it has serious drawbacks. The problem, again, is that an ordered protocol has no knowledge of the semantics of the messages it is transporting.

Firstly, this means that it cannot relax its ordering constraints for messages which do not actually need to be ordered ('false causality'), which makes it inefficient.

> The limitations of CATOCS in real-time systems are significant. ... the CATOCS inefficiency of delaying message delivery because of false causality and its general communication overheads detracts, not just from the performance, but from the correctness of a real-time system. [13, page 10]

Secondly, dependencies can exist between messages which even an ordered protocol can violate. Cheriton and Skeen give an example in which a fire control system can erroneously turn off the alarm when a fire is raging.

In conclusion, though ordered protocols seem convenient to many, they are inefficient as well as inadequate. When the added efficiency is important (as happens in real-time systems), ordering should only be done when necessary, and false causality should be avoided.

Streams

TCP and most ATM service models use the stream paradigm — a connection is treated as a continuous flow of data. Besides total ordering, two other problems for real-time systems arise from this paradigm.

Flow control allows a receiver to limit the data rate if its buffers are nearly full. This does not work well with multicast, since the state of one receiver should not affect the updates for the entire group.

> Conventional conversational transport protocols such as TCP ... are focused on supporting reliable, bi-directional two-party communication, as appropriate for conversations. They also view that flow control is an issue to be negotiated between source and destination. These attributes are clearly at odds with the basic multicast requirement of the dissemination model where a source generally cannot slow down to accommodate one slow receiver. ... The multicast and flow control considerations also made it infeasible to provide reliable communication in the conventional fashion. [12, page 6]

The second problem with streams occurs when the protocol assumes a given data rate — as happens with ATM bandwidth-reservation. Real-time traffic is not necessarily regular. It may violate maximum-burst rules even though, on average, it uses little bandwidth. "The key point is that while these applications need low latency, they don't need a *continuous* stream of low latency packets the way voice does." [15]

2.2 Virtual environments

Virtual environments are distributed simulations of a *virtual world* in which a number of human *participants* may interact. Of particular interest are environments which are sensitive to latency. Since the lower bound on latency increases with distance, any virtual environment will become more sensitive to latency as the degree of distribution (distance) increases. We have termed such environments 'collaborative visualisations', for historical reasons.

Many kinds of virtual environment are sensitive to latency — multiplayer action games, collaborative education environments, and cooperative work environments. The 'critical latency' for each application can vary widely — roughly, from 100 milliseconds to 10 seconds or more. However, in each case one can expect increased demand for larger and more widely-distributed environments, until the lower bound on communication latency becomes significant.[3]

[3]Though in the case of a 10-second critical latency, this point may only be reached once virtual environments span planets.

We call the execution environment supporting a single user an *instance* of the distributed application. If a virtual environment is widely distributed, it is likely that each host will run only one or at most a few instances for a given virtual environment.

2.2.1 Ownership

Whatever their purpose, most virtual environments contain 'virtual objects' which the participants can manipulate. Often there is a potential ambiguity when more than one participant tries to manipulate the same object simultaneously. This is usually solved with the mechanism of *ownership*. This states that at any point in time, an object can have at most one owner; only the owner may manipulate the object. Ownership of an object can change at any time, and may do so frequently — 'controllership' may be a better term.

This protocol is taken directly from human social protocol. For small objects, such as a pen, the person holding it is its owner. For larger objects, such as a vehicle, ownership may be determined from the person's relative position.

Virtual environments are less successful at allowing participants to cooperate in manipulating an object — e.g. carrying a table. This is due primarily to a simplifying assumption which eases simulation — namely that a held object has no momentum; instead it moves instantaneously whenever its owner's hand moves. In a virtual environment which instead tracks the forces exerted by participants on objects (currently very unusual), such cooperation may be easier to implement, and ownership may be unnecessary.

For some objects, ownership may be permanently assigned for the duration of the environment. A common example is a participant's *avatar* — an object representing the participant in the environment. For other objects, ownership may be *claimed* and *relinquished* when a participant takes and drops the object, respectively.

2.2.2 Interaction latency

> Latency represents the dark side of constructing global environments [41, page 77].

Interaction latency is the delay between an action of a participant and the action's corresponding representation in the output given to other participants. Low interaction latency is critical to virtual environments [15, 25, 42, 52]. High latency increases the time taken to complete a task, and if it is too high for a given collaborative task, human social protocols begin to break down. An example of this can be seen in long-distance international telephone calls.

Figure 6 shows the stages which contribute to interaction latency. It is important to include all stages between one user and another, to avoid the temptation of thinking that the latency problem is easily solvable.

For example, some systems use *motion prediction* as a traffic-reduction mechanism [25, 52, 56]. An application instance attempts to predict the actions of other participants based on previous actions; communication only needs to occur when a prediction will be sufficiently inaccurate.

While this can significantly reduce traffic, it is not a solution to latency. Latency is certainly made less noticeable while the predictions are accurate — because the user is interacting with a computer simulation of the other participants, not with real participants. But unless the other participants are redundant, the predictions must sometimes be wrong. When this happens, not only is interaction latency visible, but the system must also undo the results of its incorrect prediction. This can be extremely distracting [15].

Some of the stages in Figure 6, such as A and F, are only significant in very latency-critical environments, such as multiplayer action games. The stage of most interest to us is stage C, the communication latency. This is because, as the distribution distance of a virtual environment increases, all other stages remain nearly constant, while the communication latency increases in at least linear proportion.

Interaction latency can vary, since the latency of each stage (especially communication) can vary from event to event. Latency variation is commonly known as *jitter*,

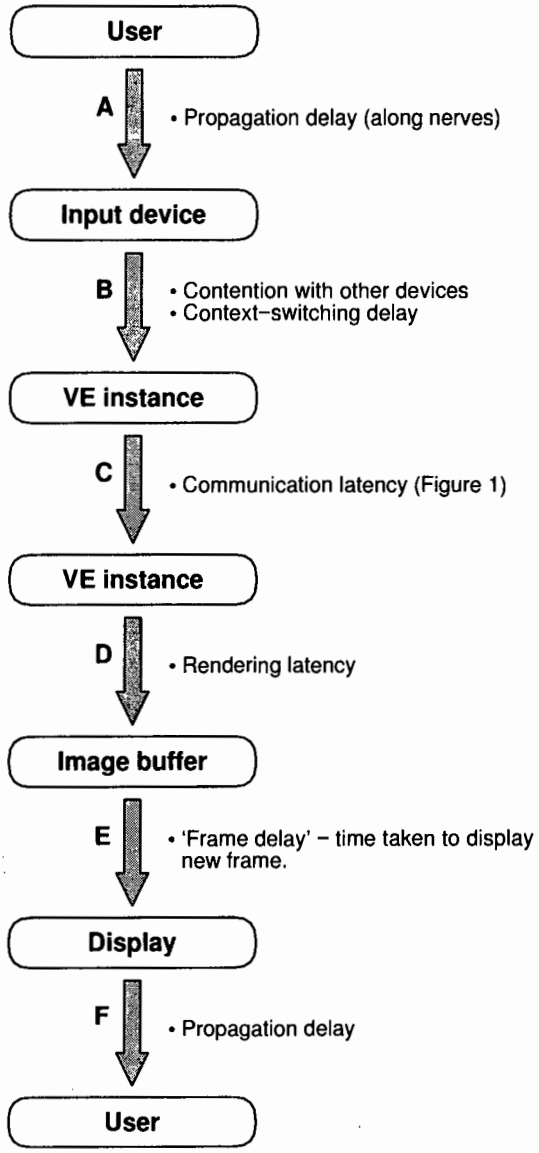

Figure 6: **Interaction latency.** One user does something, and another user sees the result. The component stages of the interaction are shown here, and for each some latency sources are listed. Stage C is process-to-process communication, which is expanded in Figure 1.

and it can be as distracting as high latency [42, 52]. It is non-trivial to describe the level of user distraction caused by a given variation in latency — distraction seems to depend on the variance, but the magnitude of latency peaks is also important. In any case, it is desirable to make latency less variable if possible.

2.3 Distributed shared memory

2.3.1 Introduction

The simplest, most direct paradigm for distributed programming is the *message passing* model, which follows directly from the network protocol's provision of primitives for sending and receiving messages. This model is to data distribution what the 'goto' statement is to flow control — without further abstraction, it is error-prone and difficult to use [4, 9, 32, 34, 48, 60].

A popular improvement is the *remote procedure call* (RPC). A client process invokes a remote procedure call with (ideally) the same semantics as a local one. In the implementation, the process sends a message containing the call parameters to a relevant server, and blocks awaiting a reply. The server executes the procedure and replies with a message containing the return values.

While this does ease programming — particularly because RPC systems provide other properties such as architecture-independence — it is still message-passing [60]. With RPC, machines are very much separate entities — it is difficult to work with the distributed system as a whole.

> With a message-based interface, control must be transferred between modules even if the interaction between these modules is limited to sharing data. Data must be stored in one of the modules and it can be accessed by the other module through a request-reply protocol. In other words, a very simple situation is programmed in a complicated way. [4, page 930].

In addition, the conceptual simplicity of RPC is not borne out in practice.

> The programmer, however, still has to be aware that the semantics of remote procedure calls are different from those of local procedure calls. For example, passing pointers as parameters in an RPC is difficult, and passing arrays is costly. [39, page 986].

Virtual environments offer participants a view of a shared world. The ideal programming paradigm should allow the programmer to manipulate the state of this shared world directly. Distributed Shared Memory is such a paradigm.

2.3.2 Distributed Shared Memory

A distributed shared memory (DSM) system emulates a single shared memory. The simplest form provides two primitives:

1. *read(address)* Returns the value stored at the given location.
2. *write(address, value)* Sets the variable at the given location to the given value.

The key property key DSM is the use of a *global address space*, which ensures that, at the DSM interface level, a given address or identifier refers to the same location on all machines.

Granularity and structure

DSM locations are usually grouped together, so that administration decisions operate on entire groups instead of individual locations. The *granularity* of a DSM system is a measure of the size of these groups.

DSM grew from *shared virtual memory* — physically shared memory on tightly-coupled multiprocessors. Early DSM systems were aimed at simulating this in systems on which memory was not physically shared [40, 60]. The read and write primitives operated on individual words of the same size as the processor word.

Hence it was natural to group locations into fixed-size *pages*, as is done in shared virtual memory [34, 49]. This makes administration more efficient than per-location administration, as it takes advantage of *locality of reference.*[4], and administration can use the processor's virtual memory system.

However, larger page sizes also increase the likelihood of *false sharing.* This occurs when two processes access different locations which happen to be on the same page. The symptoms of false sharing depend on the implementation, but the result is always diminished performance.

One approach, taken by Munin [9], is to place each shared variable or structure on a separate page. This suggests strongly that the fixed-size page is the root of the problem. Instead, many researchers advocate a *structured* shared memory, in which the unit of granularity is the *object* [1, 4].

The term 'object' is used because structured systems are, or can be, the basis for an object-oriented language. In some, the DSM itself knows the objects only as blocks of memory — an address and a length — leaving interpretation of the object structure to higher layers [29, 31].

In others, the DSM knows the exact object layout. This is necessary for the DSM to be heterogeneous, since variables may need to be translated for some architectures (e.g. swapping byte order between little-endian and big-endian machines) [35, 39].

2.3.3 Consistency

An important goal of shared-memory research was to make shared-memory programming similar to single-processor programming. One consequence of this is the requirement of *consistency.*

The strongest form of consistency is *strict consistency*, which states that a read access to a variable returns the value most recently written to that variable. This is provided by single-processor systems.

In a distributed system, distributed processes execute asynchronously, which makes this definition ambiguous. Instead, the strictest form of consistency found in DSM

[4]That is, if a processor accesses a location, it is likely to access other locations near it, in the near future.

systems is known as *sequential consistency*, which ensures that accesses appear to occur in some sequential order which is an interleaving of the accesses of each process [1, 20].

Sequential consistency is inadequate by similar reasoning to that used in Section 2.1.5 concerning causally and totally ordered communication.[5] Firstly, it is inefficient, as it forces ordering of operations which could possibly have been concurrently executed. Secondly, it is incomplete. It may guarantee the consistency of individual operations, but making groups of operations consistent is left up to the programmer.

More relaxed forms of consistency have been developed which address the inefficiency issue. *Weak consistency* provides synchronization operators with which the programmer marks synchronization points in a program. DSM accesses between these points are given free range, and the results are collated at the synchronization points.

Release consistency expands the synchronization operators into two distinct operators: *acquire* and *release*. These delimit critical sections, and the distinction allows their implementation to be more efficient. Various modifications to basic release consistency have been developed [9, 32, 34].

The drawback with all these protocols is that they ignore real-time considerations. Read and/or write operations are implemented using chains of causally-dependent messages (discussed in Section 2.1.2). For this reason, consistency maintenance causes significantly high latency.

Virtual environments can often tolerate inconsistency. In a DSM used for a virtual environment, most state variables represent the state of objects in the virtual world. If an object is being moved, for example, its corresponding variables are written frequently (tracking the motion). Inconsistency in some of these updates is virtually unnoticeable (i.e. maintaining consistency is wasteful), and can certainly be less noticeable than the added latency caused by consistency maintenance (i.e. maintaining consistency is actually inferior). That is, as long as program correctness is not at stake, low latency is more important than consistency.

"For example, the value for the oven temperature stored by a computer-based oven control in a factory should be close to the actual temperature of the oven, what

[5]Indeed, sequential consistency can be implemented very simply using a reliable totally-ordered multicast primitive [39].

we call 'sufficient consistency'. Sufficient consistency is normally provided by the sensors transmitting periodic updates, the communication system giving priority to the most recent updates (dropping older updates if necessary), and the monitoring system interpolating, smoothing and averaging updates to accommodate lost updates, replicated sensors and erroneous readings." [13, page 10]

Consistency management of some kind is required, however. If frequent updates to a variable cease, processes must agree on its final value, to avoid more permanent inconsistency.

2.3.4 Replication and distribution

A DSM system's *replication* strategy decides whether and where copies of variables are kept. The low-latency requirement of virtual environments guides replication decisions.

Assuming the 'ownership' mechanism of Section 2.2.1, locality of reference is expected for write operations (since writes are done by the object's owner, and ownership changes relatively infrequently), but reads do not exhibit locality in the normal sense. Instead, there is usually a group of processes simultaneously reading a variable, and the membership of this group changes relatively infrequently.

The latency of reads and writes hinges on the way in which object changes are distributed. *Pull-based* distribution is a commonly-seen feature of DSM systems, in which the new values are 'pulled' to the reader — that is, distribution happens when the variable is read. *Push-based* DSM systems are more rare. Here, the new values are 'pushed' by the writer — distribution to all potential readers happens when the variable is written. This requires replication — all potential readers must hold a local copy, so that the results of a push can be kept to serve read operations locally.

Traditional DSM systems use pull-based distribution because their target applications show strong locality — there are in general far more reads than writes, but far fewer non-local reads than writes.[6] Thus it would be bandwidth-inefficient to push updates to other machines because most of these pushes will be a complete waste of traffic [21].

[6] 'Local' here refers to operations performed by processes on the same machine as the owner of the variable.

Virtual environments, however, produce a high ratio of non-local reads to writes. This is true whenever a group of people is interested in the actions of another person.[7] Push-based distribution is more efficient in this case, especially if it is implemented using multicast. An update to a variable can be sent to the entire group with one multicast, while pull-based distribution will require two unicasts for each group member (unless the writer can accumulate replies and multicast them — but this would increase latency).

Push-based distribution is common in virtual environments for another important reason — it has lower latency. (Most DSM systems are pull-based, which may be why latency-critical applications such as virtual environments do not use off-the-shelf DSM systems.)

This can be seen by comparing the two types of distribution. Pulling incurs latency on a read operation. The reader must send a request message, and wait for the reply. The process or subprocess which needs the result must block until the reply is received. The latency of such an operation is the latency of both messages plus the time taken by the owner to serve the request.

Pushing, on the other hand, incurs latency on a write. In consistent DSM terminology, push-based distribution is the *write-update* algorithm.[8] Here, the writer sends the update message to all processes holding copies; there are various alternative methods of ensuring consistency. One method [60] is to use a global sequencer to sequence writes. This can be done by the bus in shared virtual memory systems, or a totally-ordered reliable multicast primitive in DSM.

However, a global sequencer is a bottleneck, and such total ordering is inefficient and induces latency (see Section 2.1.5). If inconsistency is tolerable, however, push-based distribution produces the lowest latency possible — reads are local, and writes use a single multicast.

The inefficiency involved with pushing data to processes which are not going to use it can be controlled with a more robust, hybrid algorithm. A number of such algorithms

[7]The ratio can be lowered through reader-side prediction, for example, but given a large enough interacting group, the ratio will still be significantly high.

[8]This should not be confused with *write-invalidate*. This algorithm does send messages on a write, but they do not contain shared memory state. Memory state is instead propagated during read operations.

have been developed [9, 21, 32]. They use a heuristic to invalidate copies which are not likely to be accessed in the near future.

2.3.5 Atomicity

Applications often desire that changes to a set of variables be *atomic* — that is, the changes should execute as one operation. Any process reading the variables should read either the old set of values or the new set, but not a combination. This concept is similar to the database concept of a *transaction.*

In a simple, unstructured (page-granular) DSM, operations on individual locations are atomic, but it is more difficult to guarantee atomicity for groups of locations. In a virtual environment, cases often arise when atomicity is desired. For example, the X, Y and Z co-ordinates of an object's position may be stored in separate locations. If the group is not updated atomically, erroneous positions may be reported for the object.

A DSM could potentially take advantage of knowing that a group of variables is a single, atomic unit, by grouping the new values into a single message or set of messages. Often, a page granular system accomplishes this because the variables in an atomic group can be placed on the same page.

Atomicity is also supported by release consistency (Section 2.3.3). The acquire and release operations are used to surround each *critical section* (code which modifies a group of shared variables). Accesses in a critical section can be grouped and distributed on the release operation, so that the code appears to execute atomically.

A more elegant, implicit solution uses a structured DSM. Orca [1] is such a system, which ensures that at any given point in time, for any given object, at most one instance of one method of the object is executing. This ensures atomicity of the private state of all objects (pointers and global variables are disallowed.) It may still happen that an atomic operation involves two separate, unrelated objects; the system should provide other synchronization primitives for this.

Chapter 3

Theory

3.1 Distributed Shared Memory for Virtual Environments

This research seeks an elegant programming paradigm for large, widely-distributed virtual environments. The critical factor for virtual environments is latency — bandwidth is only secondary. The relative importance of latency can only increase in the future; available bandwidth is steadily increasing, while lower bounds to latency are already being neared [15].

Currently, the most prominent examples of low-latency virtual environments are multiplayer games, such as [14, 30, 38, 54, 61]. This is not because only games are latency-critical; rather it is due to the difficulty of the problem and the lack of research to find a *simple programming paradigm* that focuses on low latency. Games lead the development because of necessity — the intense action demanded by users necessitates low latency. Designers produce low latency games by writing the networking code by hand, at a low level. This effort is justified by the commercial gain promised by success.

Other environments, such as collaborative visualisation, education and design environments, have avoided the issue, because usable systems can be built far more easily using existing easy-to-use tools which are not geared to low latency.

The problem is compounded by the secretive nature of the game industry. Game designers are loath to provide technical information, so insight into the methods employed by these environments must rely mainly on hearsay and observation.

The DSM paradigm is well suited to the 'shared world' which virtual environments maintain. In fact, many current virtual environments bear a strong resemblance to DSM systems. One such environment is mWorld [20], in which each client keeps a local copy of the world state as an Inventor [64] scene graph. The update system resembles a very basic push-based DSM implementation — state change events are identical to the DSM 'write' primitive.

mWorld uses ownership to maintain consistency, though the ownership is what one might call 'world granular'. There is one synchronisation token which represents the entire world, so that only one process may modify the world at a time.

A few virtual environment systems deliberately provide DSM. One example is RhoVeR [2], in which certain object attributes, such as an object's type and position, are placed in 'Virtual Shared Memory', which holds a small structure for each process. It is implemented using a shared memory block on each machine, with changes transmitted between machines using TCP. RhoVeR also uses the ownership mechanism for consistency.

The RhoVeR designers hint at network latency problems. They suggest changing to UDP and producing benchmarking and monitoring software to improve RhoVeR's performance.

This again suggests that an off-the-shelf distribution system is needed, instead of burdening virtual environment researchers with the task of producing one. It could also remove other restrictions which should eventually become problematic, such as RhoVeR's fixed shared memory structure, which does not allow programmer-defined data to be added.

In conclusion, the aim is to produce a low latency, DSM system which is suited to virtual environments. This means a fully-replicated, push-based system which abandons consistency in the traditional sense, since consistency ignores real-time constraints. Instead, the system will use ownership as its consistency mechanism, to allow low-latency distribution.

3.2 Explicit multicast

Subscription multicast is used in TCP/IP, Ethernet and ATM. When a host subscribes to a group, routers are configured to include that host when distributing packets marked with the group's multicast address.

This relieves the sending machine from the burden of maintaining the recipient list. Another advantage is that this simple extension of point-to-point communication does not require changes to header formats, since the multicast address can be the same length as a unicast address.

However, subscription multicast assumes that the set of receivers is similar from message to message. Subscription and unsubscription should not happen frequently, because they both involve communication with a possibly large number of routers.

For many virtual environments, this property does not hold. Instead, for each message, only some subset of the processes in the system are actually interested in the message, and there are many such subsets, possibly overlapping [42, 52, 56]. One approach [42, 56] is to establish many separate multicast groups, with each host subscribing to a set of these groups. For example, a multicast group can be established for each square in a terrain grid; hosts subscribe to groups for squares which are potentially of interest.

This is efficient when group membership is static, but when membership changes, it causes additional traffic and possibly additional latency. This makes some potential optimisations expensive. For example, a system could be made orientation-sensitive — that is, an observer's host only receives updates for objects which are in the observer's field of view. This is infeasible, however, because the subscription and unsubscription traffic caused when the observer turns around could be prohibitively large.

Another problem is the large number of multicast groups required. Singhal and Cheriton [56] note that object grouping may well be based on criteria other than proximity. For example, in a military simulation, a commander might want a view of platoon leaders only, over the entire battlefield. Here grid-based partitioning alone would be of no use. They propose 'projection aggregations' — in which grouping is based on object type as well as position.

Refining grouping in this way (i.e. by adding grouping criteria) causes a combinatorial increase in the number of multicast groups. Internet applications cannot use large numbers of multicast addresses, since address space is limited, and this combinatorial increase suggests that the problem cannot be solved by increasing the address space.

Thus, virtual environments typically use just one multicast address, to which all processes subscribe; processes simply ignore packets which do not interest them. This scheme does not scale well, because every message interrupts every subscribed host.

Instead, we propose *explicit multicast*, an alternative multicast addressing scheme in which the sending machine marks the packet with a list of the addresses of the recipients. That is, the address field in the header would be of variable length, and would *explicitly* state the recipient machines for the message.

The advantages of this are numerous. Firstly, the overhead involved in subscription to a group is reduced if, for each group, only a few machines ever send to that group. Only the senders need to be notified of membership changes, instead of a large group of routers.

Secondly, subscription can be avoided entirely when the sender can determine which hosts need information. For example, if a sending host knows the position of other observers in a virtual environment, it can determine which observers will need to be told of a change. This suits the push-based distribution advocated in Section 2.3.4.

Thirdly, knowledge of the group membership gives the sender more flexibility to reduce traffic. For example, it could partition a receiving group according to the accuracy required by each recipient. Some might need more detailed information about an object (e.g. its orientation), while for others the sender may group objects together into 'impostors' [56]. Section 4.2.3 discusses a variation in which the rate of updates to each recipient is adaptive.

Similarly, since the sender knows the size of the group, it could smoothly degrade the quality of its output as the group size increases. This would make the application more scalable, and would give it more control over the total quantity of produced traffic.

Explicit multicast does have some disadvantages. Firstly, explicit multicast would be more difficult to implement in a connection-oriented protocol suite like ATM. This is because explicit multicast is connectionless; it does not need initialisation

(subscription) like subscription multicast does. However, the implementation would somehow need to set up connections in the underlying connection-oriented protocol. This disadvantage is really a consequence of the restrictive nature of connection-oriented service.

Secondly, since the sender must maintain the recipient list, it is not efficient when many senders wish to send to the same, static-membership group, as then each sender would need to hold the entire list.

Finally, the addressing information in an explicit multicast packet could be large. For example, if the recipient list contains unicast IP addresses, the header would have to be a variable size, and it would typically be quite large, causing inefficiency.

One solution to the last two disadvantages is to make a hybrid of explicit and subscription multicast. For example: All participating processes subscribe to a single multicast group, and processes in the group are numbered. The addressing information in the message sent out contains the multicast group address plus a bit field indicating the recipients. It would be acceptable to impose a limit on the number of recipients in one multicast packet, say 64, in which case the header size could be made constant.

3.3 Polling

Polling is the repeated sampling of some entity's state to determine when an event occurs. Polling is common at many levels. For example: a processor polls its interrupt line every clock cycle; many graphical user interfaces use an 'event loop' which polls the event queue; a virtual environment using message-passing may poll the network message buffer.

Ideally, any system in which latency is paramount should be completely event-driven. That is, when a significant event occurs, the relevant handling process should immediately be invoked. However, this is not always possible. Often, one finds that the system does not provide adequate support for the relevant events. Another important reason is that the system designer may want to restrict the times at which an event may be serviced (as in the above example of processor interrupts).

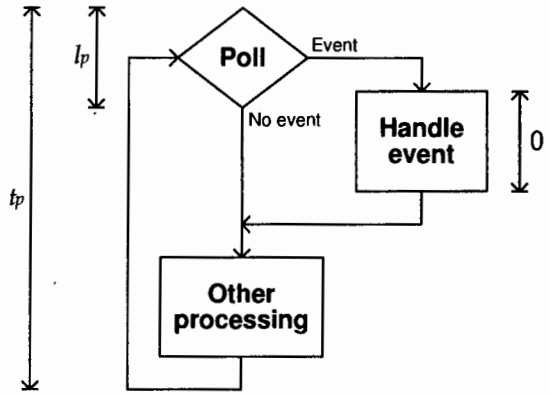

Figure 7: **The polling loop.** This flowchart includes the time taken for each stage — event handling takes zero time, polling takes l_p seconds, and the entire polling loop takes t_p seconds.

Thus, polling is sometimes a necessary evil. It is therefore important to examine its effect on latency.

3.3.1 Polling model

We assume that the time taken to process an event is negligible. Figure 7 illustrates the polling loop.

Events are independent of the exact time of polling, and occur, on average, every t_e seconds. The process polls once every t_p seconds. The time taken to perform one poll is l_p.

Then the added latency due to polling, l, has an expected value of $\frac{1}{2}t_p$. To characterise the jitter caused, we note that the latency is uniformly distributed over the interval $[0, t_p)$, (and with a peak t_p).

To examine the efficiency, note that t_e/t_p is the expected number of polls per event. If this value is greater than 1, then $l_p((t_e/t_p) - 1)$ seconds of processing are spent on unsuccessful polls.

3.3.2 Example — the World Wide Web

Bob is interested in an upcoming release of Acme corporation's latest product. It will be released some time in the next 60 days, but he isn't sure when exactly, and he wants to know as soon as possible so that he can take action.

So, every day, he visits the Acme Web site. It takes him 1 minute to download the page and check for a release notice.

If the product is actually released on the 30th day, Bob will spend 30 minutes in total, downloading and checking the page. Thanks to his diligence, the maximum latency — the time between the product release being reported on the change and Bob knowing about it — is one day, with an expected value of half a day.

Contemporary browsers now contain features which do this polling in software. Only the document timestamp needs to be checked, instead of downloading an entire page, but this must still cause considerable load on servers which host popular pages.

Curiously, the World Wide Web has features in common with virtual environments. As with virtual environment objects, Web pages are typically written by a small group of hosts, but read frequently by a large group of hosts. In addition, latency is important to users (though on a larger time-scale). One could therefore envisage the Web evolving to a push-based, replicated system.

3.3.3 Polling in a virtual environment

The DSM system to be developed will use a polling function, called by the application, which processes received network messages.

This decision is motivated by the following factors:

1. The library must be able to update application objects. In the absence of portable shared memory support, this means that the library must run in the same process as the application, which in turn means using either multithreading or the polling function.

2. In order to maintain atomicity, processing of object changes should be restricted to times in which the application is not accessing objects. Using a polling function accomplishes this simply [37].

Virtual environments typically have a display loop which repeatedly generates video frames.[1] The call to the polling function can be placed at the beginning of this loop, thereby guaranteeing frequent invokation as long as the frame rate is high.

For at least the visual side of the virtual environment, this display loop causes a latency of up to one frame interval to be added to any event, since the event's screen representation will only appear on the next frame. Therefore, using polling does not increase latency — the display loop is much like a polling loop, and the latency it causes will mask the library polling latency.

[1]This, in itself, is much like polling — the display loop repeatedly samples the world state in order to generate a screen representation for it.

Chapter 4

Implementation

DSMVE is an object-granular push-based DSM. A local copy of an object is represented by a C++ 'shareable' object. An object is shareable if its class is derived from the `SObject` abstract class.

The DSM uses ownership as a non-traditional coherence mechanism. DSMVE has a central server which is used for session management and ownership management, but not for state distribution.

The following sections describe the implementation of DSMVE, first by outlining the programmer's interface, and then by detailing its implementation.

4.1 DSM interface

4.1.1 Marshalling

SObject is itself descended from the `Networkable` class, which means it contains the following two marshalling functions:[1]

```
void read(NetworkBuffer &)
void write(NetworkBuffer &) const
```

[1]These marshalling functions should not be confused with the conceptual `read` and `write` operations of a DSM.

These transfer an object's data to and from a network buffer. They must be defined by the programmer for each shareable class. However, they are typically easy to write; usually they simply invoke the corresponding `read` and `write` member functions for the class' data members. Future work should include automated generation of marshalling code.

DSMVE supplies `Networkable` base classes (currently only 16-bit and 32-bit integers). These are architecture-independent, since translation to and from the network-neutral representation is done by the `read` and `write` functions.

4.1.2 Ownership

DSMVE uses ownership as a compromise between consistency and interactivity. Class `SObject` defines two member functions for ownership:

```
int claim()
void relinquish()
```

The `claim` function claims ownership of the global object. It is not guaranteed to succeed (failure is indicated by the return value). If it does succeed, the invoking process becomes the new owner.

The `relinquish` function relinquishes ownership of the object, making it unowned and available to be claimed by other processes.

Only the owner may modify the state of an object. (In fact, currently there is no mechanism to detect violation of this rule, so non-owners may modify their local copies of objects, but in such cases the changes will not be distributed.)

4.1.3 Registration

For each DSM object,[2] the server maintains a record showing the current owner, its current state if unowned, and a list of 'readers' — processes which hold a local copy.

[2] One `SObject` represents one *local copy* of a DSM object.

A string identifier is used to bind an `SObject` to a DSM object. This binding ('registration') is done when an `SObject` is initialised, either in the constructor or in an initialisation function.

When an object is registered with the server for the first time, its current state is transferred to the server. When an already-registered object is registered by other client applications, the server or object owner transmits the object's current state to the registering processes.

Thus, barring the first registration of an object, registration operations are like a cache miss 'read' operation in a normal DSM — the process registering its copy of an object *pulls* the latest value.

However, all subsequent data transfers involving that copy are guaranteed to be push-based, until the copy is deregistered. That is, no provision is made for invalidating copies when there is memory contention. This is because such invalidation is not really a solution to memory contention, because the latency caused by subsequent reads could violate the real-time constraints of a virtual environment.

With such a guarantee in place, and given the ownership method of memory consistency, a non-owner process does not need to perform any communication when reading an object — it simply uses its local copy.

4.1.4 Object modification

DSMVE does not provide explicit `read` and `write` functions; it instead strives to make reading and writing implicit.

Reading is completely transparent to the programmer, since a process can at all times read a local C++ object as if it were the global DSM object (once the local copy has been registered.) The price paid for this is that the programmer cannot assume any consistency between different objects — objects are updated completely independently of each other.

The programmer must explicitly mark object writes by calling the `update` member function of the `SObject` class. This is necessary for efficiency as well as to provide atomicity. It may be likened to the 'release' operation of release-consistent DSM, except that ownership is not relinquished when `update` is invoked.

`update` should only be called for objects that the invoking process currently owns. If the application modifies a local copy of an object that it does not own, updates are not distributed, and the change will be overwritten the next time the real owner modifies the object.

Figure 8 gives the code of an example class, which represents the 2-dimensional position of an entity.

Here, registration is done by the constructor, and the `set` function calls `update`. The application uses this class through its interface in the standard manner, and the library keeps object copies up to date.

4.2 DSM Implementation

The implementation of DSMVE is divided into 3 layers, as shown in Figure 9.

The networking layer encapsulates all operating-system networking functionality, and defines needed low-level classes such as the Networkable abstract base class.

The session layer handles server connection, disconnection and maintenance of the list of connected processes. It implements a reliable multicast, which is used by the connection and disconnection code as well as the next layer.

The DSM layer implements the distributed shared memory, defining the SObject class and handling all DSM-related protocol. It implements its own protocol for update distribution.

4.2.1 Networking

Two different network protocols are used. The first is a locally-ordered, reliable multicast which is used by the session and DSM layers for all communication except object updates. This is implemented in the session layer.

The second is a locally-ordered, non-reliable multicast which is used in the DSM layer for object updates. Ordering is done on a per-object basis, which is why the protocol is implemented at such a high level. This protocol is the critical one since object updates dominate the network traffic in a virtual environment, and are latency-sensitive.

```
class SPosition : public SObject {
private:
  NInt32 x,y;

  // Marshalling functions
  void read(NetworkBuffer &buf) {
    x.read(buf);
    y.read(buf);
  }
  void write(NetworkBuffer &buf) const {
    x.write(buf);
    y.write(buf);
  }

public:
  // Constructor
  SPosition(Identifier name) { init(name); }

  // Update function
  void set(int newX, int newY) {
    x=newX;
    y=newY;
    update();
  }

  // Reading functions
  int getX() { return x; }
  int getY() { return y; }
};
```

Figure 8: **An example shareable class.** One could make this class act more like built-in type, by renaming set to operator=.

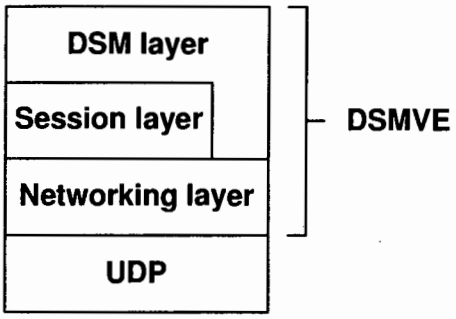

Figure 9: **The layers comprising DSMVE.** DSMVE is built on top of UDP and consists of three layers. DSM layer uses the networking layer directly when it implements the update protocol.

Both of these protocols must allow any machine to communicate with any other. This precludes the use of a connection-oriented unicast protocol, because the $O(n^2)$ connections required for n participants severely limit scalability. Connections are not lightweight enough that an application may use hundreds or thousands of them; for example, the setup time for so many connections would be unacceptably large.

Thus, TCP and ATM's connection-oriented services are unsuitable for the communication in DSMVE. Instead, it uses UDP, which is the connectionless, unreliable protocol provided by TCP/IP.

We examine the two protocols in DSMVE separately.

4.2.2 The message queue

The message queue is an explicit multicast primitive. It is locally-ordered (i.e. all messages sent by a process are received in the same order as they are sent), and reliable (i.e. if packets are lost, they are re-sent). It uses total ordering; this makes the message queue easy to implement, but means that the message queue should not be used for messages which are latency-critical.

The message queue uses a simple positive acknowledgement scheme; each message is acknowledged, and messages are re-sent after a fixed timeout period. The session layer has no concept of a dropped connection, so the system does not tolerate host failure. These properties should be addressed in the future (see Section 5.4).

The ideal explicit multicast implementation for the message queue would use protocol-supported explicit multicast,[3] but TCP/IP does not provide this. Instead, it uses multicast by unicast.

This is not at all scaleable. A possible future improvement would be to use a single UDP multicast group to implement the multicast. However, this would only be moderately scaleable, and is no substitute for a protocol-supported explicit multicast.

4.2.3 DSM object updates

The update mechanism implements an unreliable, ordered, explicit multicast. Ordering is done on a per-object basis — updates to separate objects are independent.

As in the message queue, object updating should ideally use protocol-supplied explicit multicast. Currently it uses unicast, and a possible improvement would be the use of a single multicast group.

The following sections outline some of the interesting features of the update mechanism.

Local pointers

A side issue explored by DSMVE has to do with object naming. A message containing an object update must have a field which identifies the object in question. Objects in DSMVE are identified with a string; the size and speed impact of using a string identifier for object updates could be prohibitive.

Instead, this experimental scheme uses 'local pointers' to identify objects to be updated. The owner of an object holds information on each reader,[4] and this includes the address, in the reader's address space, of its read copy.

The owner cannot itself derive any meaning from this pointer, as it is valid only in the reader's address space, but it includes it in update messages, in order to identify the object being updated.

[3] See Section 3.2.

[4] A 'reader' is a process which holds a read copy of the object.

The advantage of this is that a reader, on receiving an update message, does not need to look up the object's identifier in a table. Instead, it uses the pointer contained in the message to directly update its local copy.

This experimental technique was implemented purely out of curiosity, and in the current setting it has no hope of making the DSM significantly more efficient. The memory accesses it avoids are negligible in comparison to network latency.

However, such a technique may be valuable when network latency is extremely low; then the increasing difference between processor speed and memory access time may make it viable. What is interesting about the technique is that it works in a heterogeneous environment; all that is needed is provision for a variable-sized pointer field in the update message.

The main disadvantage of the local pointer technique is that it is not very scaleable. For every recipient, the update message must include a separate local pointer. This is acceptable in the current implementation, because it uses unicast anyway, and so each unicast only needs to contain the local pointer. If the system is changed to use, for example, a UDP multicast group, the update message would need to contain all these pointers — one per recipient. It would be better to use a more conventional, global identifier.

One alternative is to use a string identifier and a hashing function. Another, which we expect to use in the future, is to have the server assign an integer identifier to each object. Given such an identifier, a reader can obtain its local pointer to the object using a single table lookup.

Latency tolerance

In DSMVE, each object has associated with it a *latency tolerance*. This is a value, in milliseconds, which states the amount of latency which will be tolerated by the application, for that object.

This is used to suppress the distribution of unnecessary updates. For example, if a virtual object is being moved using the mouse, its position may be updated 50 times a second. But a latency of 100ms would usually be acceptable [15]. If this

latency tolerance is specified, DSMVE need only distribute one update every 100ms — producing 10 update messages per second.

One side-effect of this technique is that it could lower the perceived smoothness of motion. This should be solved with interpolation, as described in Section 5.4.2.

This use of the latency tolerance effectively limits the update rate of an object. It achieves a traffic saving when the application updates the object at a higher rate than the limit. Section 5.4.2 discusses some other possible uses.

Filtering

In a simple virtual environment, every update is sent to every host. This strategy is not scalable, as it causes $O(n^2)$ traffic for n participants. It is also unnecessary; in very large environments, it is unlikely that every participant needs to know about the actions of every other. Identifying and suppressing such superflous updates is known as *filtering.*

Various methods of filtering are used. RTime [52] uses a filtering system which defines a circular area around a participant's position to be the 'filtering area'. By default, the participant receives updates for all objects inside this area, and none for objects outside it, though the application may change this for specific objects.

Such discrete (all-or-nothing) filtering methods are common. Another example is the division of an environment into 'rooms'. Participants only receive updates for objects in the same room as the participant. Multi-user dungeons (MUDs) and their descendants have used this for a long time.

Researchers are now looking at more continuous filtering methods. Singhal and Cheriton [56] describe a system which identifies objects which do not need complete accuracy. It groups together objects with similar requirements into an *aggregation.* This aggregation is a statistical description of the number and distribution of the objects, and its state is much smaller than the combined state of the objects themselves. Thus a dramatic traffic reduction can be achieved.

The key feature of this technique is a reduction in simulation quality of unimportant objects. Another way in which simulation quality can be reduced is to reduce the

update rate of less important objects (mentioned in [43]). This can be done in DSMVE simply by allowing the application to increase the latency tolerance of such objects.

4.2.4 The central server

On startup, the application must 'connect' to a server process. This server holds a list of all DSM objects, and manages local copy registration and ownership.

The primary reason for using a server is that it simplifies the ownership mechanism. A single process must arbitrate the claims made on an object; using a server to perform all such arbitration is a simple solution. Section 5.4.3 examines some alternatives.

The server handles three operations at the shared-memory level, illustrated in Figure 10.

Register operation

The *register* operation is used by a process to register its local copy of a DSM object. If the object is unowned, the server replies directly. Otherwise it notifies the object's owner, and the owner replies to the registering process.

This is necessary because the reply contains the current state of the object; if the object is owned, the owner is the authority on the object's current state. The server maintains the current state of unowned objects for this purpose.

One special case occurs when the object is registered for the first time. Then the server has no knowledge of the object; in particular, it does not have an initial state for the object. For this reason, the registration request contains an initial state, which is only used in this case. This inefficiency is addressed in Section 5.4.1.

Claim operation

The *claim* request contains the object's string identifier. The server's reply contains a result value indicating success or failure.[5] If the claim is successful, the reply also holds the current state of the object and the list of registered read copies.

[5] 'Failure' occurs if the object is already owned.

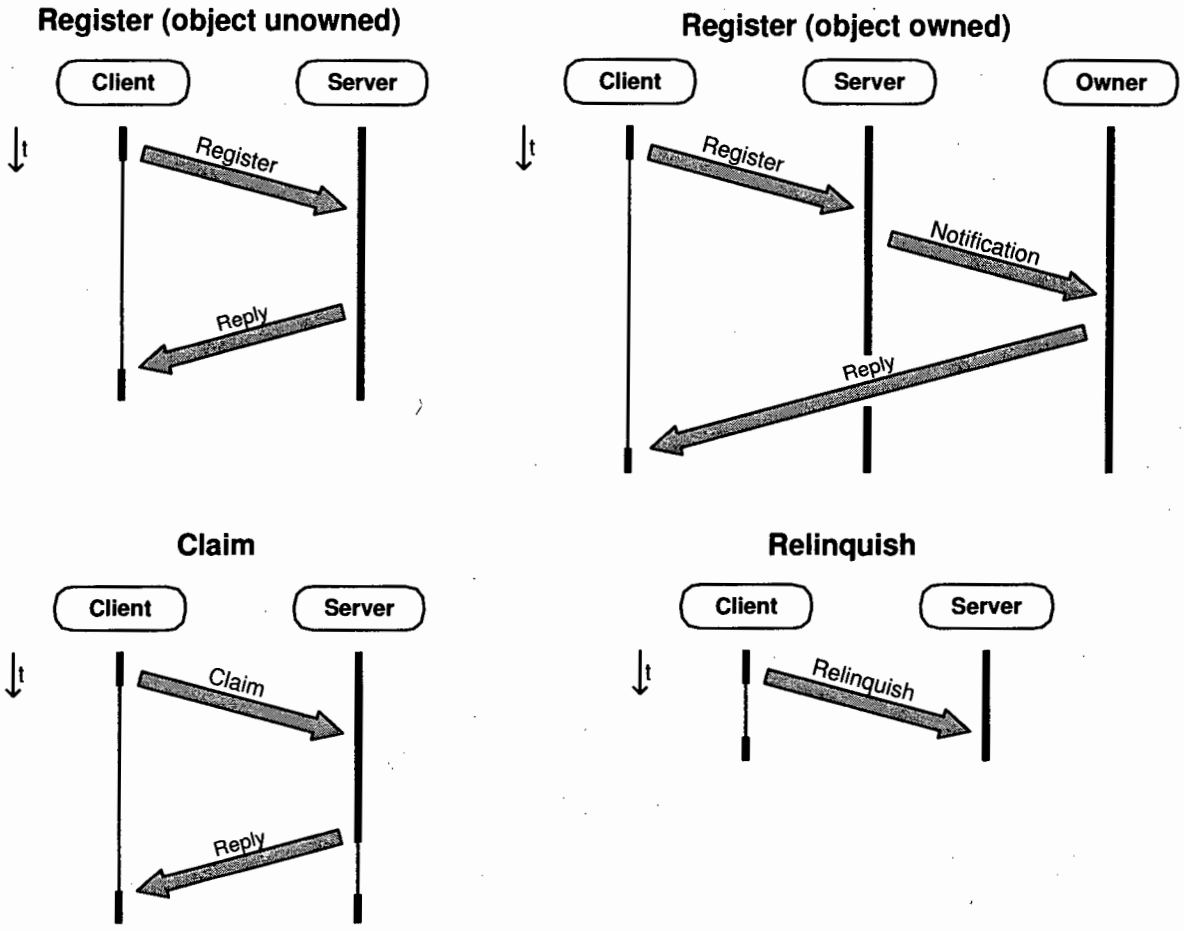

Figure 10: **DSM operations.** These event diagrams show the protocol for operations handled by the server. The register operation has two cases — based on whether or not the object in question is currently owned. The thin-line gaps show intervals in which a process blocks, waiting either for a reply or for a message to be delivered.

Currently, the server must block until the claim reply has been successfully delivered. This is a significant limitation; Section 5.4.3 discusses alternatives.

Relinquish operation

The *relinquish* operation is a simple notification to the server. With reliable updates (see Section 5.4.2), the relinquishing process must first ensure that the last update has been delivered.

It should also ensure that the relinquish message is successfully delivered before doing anything that assumes non-ownership of the object (e.g. exiting the application). Currently, the client simply blocks until the message is delivered; Section 5.4.4 discusses alternatives.

Update distribution

Unlike DSMVE, many virtual environments use the central server for distributing updates as well [30, 52].

Such systems are usually synchronous. Updates are clocked, with the timing controlled by the server. Each cycle has two phases — a collation phase, in which each client transmits updates for objects it owns, and a distribution phase in which the server transmits the collated updates to each client.

Using a server for distribution reduces traffic in at least two ways (compared to the peer-to-peer distribution approach used in DSMVE).

Firstly, a significant traffic savings can be made when the update from one client occupies only a fraction of one network packet. Then the distribution phase uses very few packets since the updates from a number of clients can be grouped into a packet. This reduces both traffic and latency. Section 5.4.2 describes a similar collation scheme for DSMVE.

Secondly, since the server has up-to-date knowledge of the entire world state at each cycle, it can filter the traffic sent to each client, in a manner that is easy to code (see Section 4.2.3).

Unfortuntely, since updates are so frequent in virtual environments, the server becomes a bottleneck. The server must receive and process every update in the system, and this limits scalability.

Another major problem is higher latency. The client-server approach uses a causally-related chain of two messages (Section 2.1.2), while the peer-to-peer approach uses one message only. The latency of client-server distribution will therefore be higher than that of peer-to-peer distribution.

The degree of this increase depends on the location of the hosts. If the server is on the path between two clients, the total propagation delay between those two clients is unchanged. In this best case, latency is increased only by other factors such as server load, task switching overhead, etc.

In the more typical case, clients may be 'distant' from the server. Even clients near each to other are then afflicted by high latency, because the updates must travel the long distance to the server, and back, instead of just the short distance between the two clients.

These two problems — latency and the server bottleneck — are addressed by RTime [52], which uses *multiple* servers, and tries to place the servers near to clients. Each client attaches to its nearest server. Using multiple servers also reduces traffic, because the distance that unfiltered, uncollated data must travel is now shorter.

It is interesting to take the multiple server system to an extreme which minimizes latency. Then for every client, there would be a server on the same local network. Servers would also be dedicated machines.

For minimum latency, servers would be linked together in some sort of network, so that they could do their own routing instead of using IP routers elsewhere on the local network.

When one examines this extreme system, the distinction between peer-to-peer distribution and multiple-server distribution begins to blur. The multiple servers act as multicast routers specialised to perform collation and filtering.

If all the IP routers on the Internet supported explicit multicast, then peer-to-peer distribution as used in DSMVE would strongly resemble the above ideal system.

Efforts are currently being made to promote IP multicast and routers which support

it [58]. Since explicit multicast avoids the subscription step which makes IP multicast difficult to implement scalably, it is conceivable that current routers could change to support explicit multicast. In this case, scalable peer-to-peer distribution would be achievable.

In the multiple server approach, servers also perform collation and filtering. Section 5.4.2 describes how collation might be added to multicast routers, but filtering must be done by the clients. Each client performs filtering for updates it will send — hence the need for an *explicit* multicast. One bonus of this is that the filtering load is distributed more evenly.

For a client to perform filtering, it must have reasonably up-to-date values of the variables used to make filtering decisions. At the very least, this includes the position of every participant. One attractive feature of latency tolerance is that the positions of distant participants[6] need only have a low update rate (i.e. a high latency tolerance) for this to work.

4.2.5 Polling function

DSMVE defines a function which polls the network and its internal structures (for message resends, etc.), `shm_poll`. This function should be called frequently, e.g. every 20ms. The frequency is flexible, but should be high enough that delays between calls do not add significantly to the interaction latency. As mentioned in Section 3.3.3, the latency caused by polling can be hidden.

The presence of this function is unfortunate, but necessary. The shared memory library must run in the same process as the application, so that the library can update application objects. Section 3.3.3 motivates the use of a polling function, and Section 5.4.1 examines alternatives.

[6]This refers to participants far away in the virtual environment, not in network distance.

Chapter 5

Sample applications

Two sample applications were written to demonstrate DSMVE. The first is a cooperative work application — a multi-user whiteboard. This contains just the necessary basics for demonstration, as it was used early in development to test the DSMVE library. The second application is a three-dimensional virtual environment.

The next two sections describe the applications separately. They are then discussed in the following section. The final section describes limitations and future improvements, some of which were identified with the help of the sample applications.

5.1 Whiteboard

The whiteboard allows participants to view a large work area on which some objects are placed. Participants may scroll to view different sections of the area, and move the objects around by dragging them with the mouse. Figure 11 shows three participants viewing the same whiteboard.

The whiteboard was implemented using an excellent user-interface development library, Amulet [8]. Like most similar libraries, Amulet is not distributed, so DSMVE was used for distribution.

A possible conflict that may arise occurs when participants try to move the same object at the same time. This is solved by the ownership mechanism. When handling

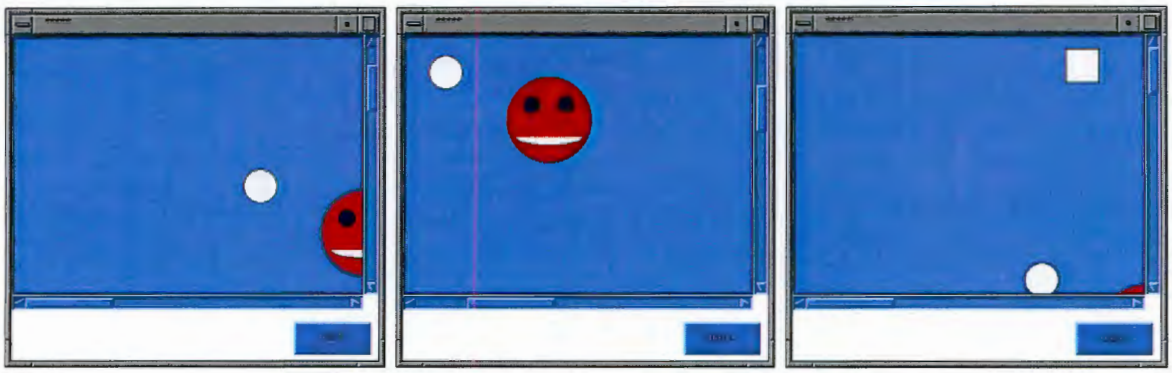

Figure 11: **Screen shots from the whiteboard application**

the mouse-down event that starts object dragging, the application claims the object. Subsequent motions cause updates, until the mouse-up event occurs, at which point the object is released. If a claim fails (i.e. some other user is already moving the object), dragging is aborted.

Amulet allows the programmer to code the event loop explicitly (though this is not the default). This was done so that a call to the DSMVE polling function (`shm_poll`) could be inserted. Testing showed that this function was invoked over a thousand times per second. To make the program more efficient, code should be added to lower this frequency.

The whiteboard example showed a minor conceptual incompatibility between the user interface and DSMVE. As is common in user interface libraries, Amulet provides objects to encapsulate the state of interface elements. However, the C++ class used for this, `Am_Object`, does not represent an Amulet object, it represents a *reference* to one. An `Am_Object` is therefore treated as a value, not as an object.

This makes it impossible to use inheritance for making the Amulet class 'shareable'. Instead, the application had to create a DSM object for each interface element. The two resulting objects for one interface element are linked together using pointers in each object,[1] and member calls to keep the two copies consistent with each other.

This complication decreases efficiency only slightly. The main impact is on ease of use — linking the two objects together has to be done as a separate step, after both

[1]Amulet uses a 'prototype-instance' object system. Amulet objects can be extended to accommodate user-defined *data*, such as a pointer to the corresponding DSMVE object.

have been initialised, and this is a burden to the programmer.

5.2 Virtual reality example

The virtual reality example simulates a room which contains a number of objects. Figure 12 shows three views of the same environment. The user can move freely around the room, and can select an object by clicking on it. The currently-selected object is shown locally in bright red, and it copies the user's movements by keeping its relative position constant.[2]

This environment was written with the OpenGL graphics library [47]. Unlike the whiteboard example, there was no problem making objects shareable, because the encapsulation of virtual reality objects into C++ objects was done in application code. This is done with a class called `VEObject`, which inherits its shareability from the `SObject` class.

DSMVE does not support machine-independent floating point types. Since OpenGL uses floating point values, it was necessary to write conversion code between integer and floating point in the `VEObject` code.

The ownership mechanism was used for object selection. When the user clicks, any currently-selected object is relinquished, and the clicked object is claimed. Movement of the user causes updates to the currently-selected object.

The polling function is unobtrusive, since the virtual reality example has a display loop that generates new display frames frequently. The call to `shm_poll` is simply inserted into this loop. Measurement showed that it was called at least 60 times per second, giving a peak polling latency of 17ms.[3]

5.3 Discussion

The basic operations on objects — claiming, relinquishing, reading and updating — are concise and fit well into both sample applications. This is true even in the

[2]This method of moving objects provides more freedom of movement than the traditional method of dragging objects with the mouse. Users found it confusing at first, but grew accustomed to it.

[3]For visual output, this latency is hidden as discussed in Section 3.3.3.

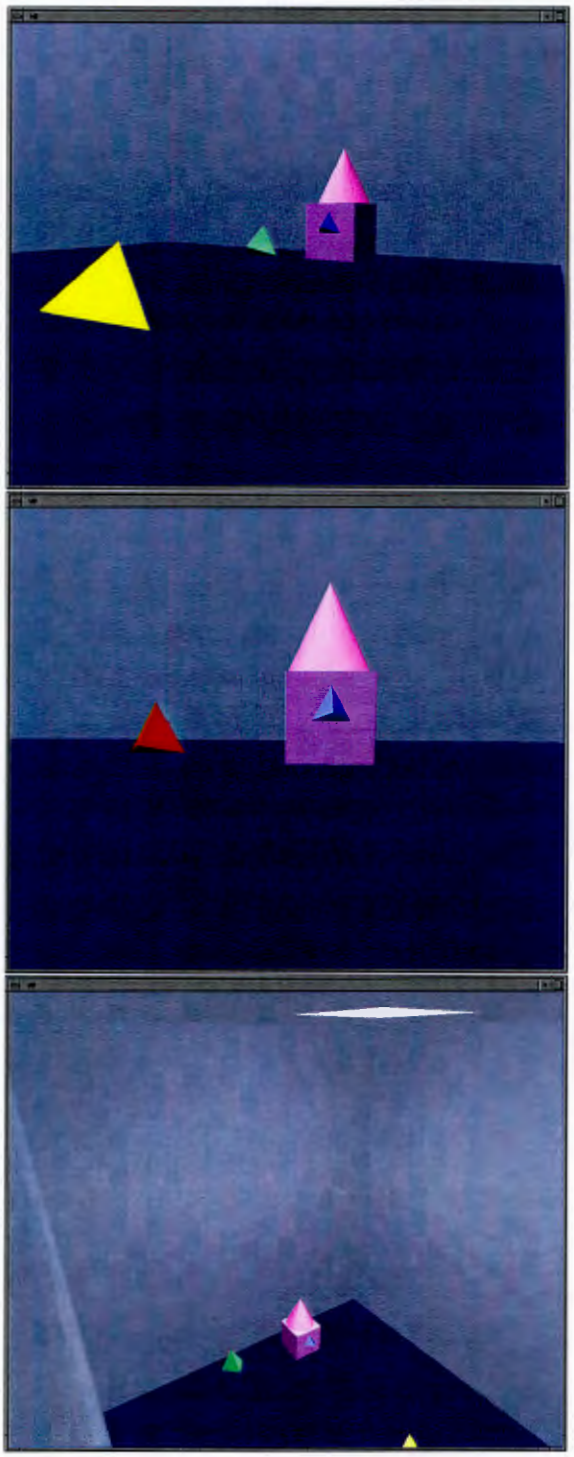

Figure 12: **Screen shots from the virtual reality example.** In the centre picture, the user has selected an object, so it is drawn in bright red.

whiteboard application, because code which links the two object representations can be hidden in the implementation section of the shareable class. Once this is done, external code which uses the shared objects has a simple interface. In both cases, the ownership mechanism was also useful for resolving conflict over control of objects.

A weak area of DSMVE is object creation and initialisation. Prime among the problems is that of object naming — requiring the application to register objects explicitly means that the application has to know the names of all objects in the system.

Thus it is difficult to allow a user to add an object to the environment. To illustrate, it can be done as follows: One shared integer variable holds the number of the next object to be created. To create a new object, a process must claim this variable and increment it. The value of the counter allows the object to be given a unique name, and other processes must notice that it has been incremented. Each process responds by creating and registering a new read copy.

This scheme is a burden to the application programmer, and it is inefficient. The counter variable is a significant bottleneck which limits application scalability.

Another example of the same problem is the provision of avatars — objects which represent participants in the virtual world. Here, a new object is added to the environment whenever a process joins, and is deleted when the process leaves. The application can find out which processes are currently in the session, so it can maintain local copies for the avatars, but this is also a burden to the programmer. It would also require DSMVE to notify the application whenever another participant joins or leaves.

These problems should be solved by providing better support for dynamic allocation; one way of doing this is described in Section 5.4.1.

5.4 Limitations and future enhancements

5.4.1 Programmer's interface

Compiler support

The marshalling functions (`read` and `write`) have to be coded by the application programmer for each shareable class. They are generally very simple, as they invoke the corresponding `read` or `write` function on each data member of the class in turn.

It is therefore desirable that `read` and `write` be automated. This cannot be done in C++ as there is no mechanism for iterating over the members of a class. To allow this, one could extend C++ by adding DSMVE keywords, for example by writing a preprocessor.

Incorporating DSMVE support into the language allows for other improvements as well. These will be mentioned in the following sections.

Registration

Registration is cumbersome as it is not done automatically by the constructor. This is because connection to the server (using `ses_init`) must happen before object registration, and a conflict would therefore occur if a `Shareable` object were made global. (The global object's constructor would be invoked before the application started executing.)

Registration is a menial task, made more complicated by the need to provide a string identifier for the object. With compiler support, registration of global objects could be done automatically by the DSMVE library on connection. To do this, the library would need access to run-time type identification, and a list of the global objects in an application.

Initialisation

Currently, any read copy must be initialised before it is registered, in the event that the server has no record of the object. (The server must obtain the state of the first

copy of a DSM object, so that it may initialise other copies to the same state.) If initialisation of objects is complicated, the application programmer might well desire more control over the initialisation process.

This could take the form of a callback member function, which the application programmer codes to initialise the object. When an object is registered, the server's reply indicates whether or not this is the object's first registration. If it is, the callback function is invoked to initialise the object, and the resulting object state is returned to the server.

This would increase the latency of the first registration, as it adds another message to the operation. It also makes the interface more complicated, as it uses a separate function to initialise objects, instead of using the constructor.

The heart of the problem is that the compiler is not aware that objects are shared, so the constructor is invoked once per local copy. With compiler support, these complications could be removed, so that the object's constructor is only invoked once for each DSM object.

Asynchronous execution

Another possible improvement is to remove the need for explicit polling, as it could be a burden to the programmer.

Without explicit polling, the application and the DSMVE library will have to be asynchronous. One way is to use multithreading, and have the library execute a sleep/poll cycle in its own thread. This would require a thread library which allows precise control over how the library thread executes, to ensure consistently low latency.

Another way is to make the library event-driven. This involves configuring the process so that it receives a signal from the operating system when a packet is received, and another signal when a timer times out (for use with message resends in the reliability code). This is more preferable as it avoids the inefficiency of polling. However Unix signals and timer support are not widely portable, and are far from ideal.[4]

[4]For example, signal reliability is not supported everywhere, and signals cannot pass data (message parameters).

But however it is done, asynchrony would make atomicity more difficult to provide, since it would allow the library to update an object while the application is accessing it.

One solution uses a lock on each object. The lock controls entry to critical sections, forcing one of the threads to block until the other is finished. (If the library is event-driven rather than multithreaded, the library cannot simply block, as that would block the application. Rather, it must store the update somewhere else and defer it until later.)

This increases the latency of an update. There is a fixed increase due to the locking overhead, and a variable increase due to the delaying of updates when there is a conflict. The latter factor will also increase jitter.

Object granularity

The granularity of a virtual environment object may be too big. One object may contain many distinct 'atoms', such as its position, orientation, size, and colour. Object operations may only modify a subset of these atoms, and it seems wasteful to distribute the state of an entire object when only some components have changed.

It would be infeasible for the library to monitor object changes at run-time. However, an intelligent compiler could note what sections an operation modifies, and create a separate update message for each operation.

Such an improvement has two direct effects: it lowers bandwidth use, and it lowers the size of update messages. The effect on latency is indirect. Lower bandwidth use means lower network contention, which will in turn slightly decrease the average latency, and will decrease jitter.

Inheritance

The requirement of an explicit call to the `update` member function complicates inheritance. For example, consider an operation in a derived class which invokes two atomic operations in the base class.

As DSMVE stands, it is impossible for the new operation to be made atomic, since the base class definitions explicitly call `update`. In fact, the call to `update` should not appear explicitly; it should be done implicitly whenever an atomic operation ends.

This requires a language extension which allows the programmer to mark operations needing atomicity, and compiler support to indentify the points at which `update` should be invoked.

Operation grouping

The server serialises all operations besides object updates. A group of n operations therefore takes n times as long to complete. For this reason, using DSMVE with a large number of objects will make initialisation and registration unacceptably slow. 'Claim' and 'relinquish' operations will also be slow if objects are claimed/relinquished in large groups.

Serialisation is unnecessary when an operation is performed multiple times on a set of unrelated objects. Therefore, protocol could be added to group such a set, sending the requests to the server in one message.

It has already been mentioned that compiler support would help registration and initialisation. Grouping of registration and initialisation could then also be done implicitly, by the compiler, making the programming interface no more complicated, and yet dramatically reducing the latency of these operations when there are many objects.

Containers

In Section 5.3, an argument was given for the provision of better dynamic allocation support. One way which seems suited to the needs of a virtual environment is to provide a system-supported extensible container, as follows.

The container is arbitrarily extensible, via `add` and `delete` operations, and has array reference semantics. An iteration mechanism is also provided to iterate through the non-deleted items in the container.

Containers, therefore, provide dynamic allocation and alleviate the naming problem — a virtual environment no longer needs to name every single object in the system. Containers also help to group registration and initialisation operations, as proposed in Section 5.4.1.

A standard container allows its contents to be claimed and relinquished individually. Operation grouping is made possible by providing another type, a 'wholly-owned' container, which is claimed and relinquished as a whole.

For example, consider adding to the whiteboard application to allow polylines to be drawn on the whiteboard. The set of lines on the whiteboard would use a standard container, but an individual polyline, a set of points, could use a wholly-owned container. The advantage of this is that all operations on the polyline, including adding and deleting points, can be done without the owner having to consult other processes or the server.

5.4.2 Object updates

Reliability

Section 2.1.5 explained that the standard approach to reliability, which ensures that every update is eventually delivered, is unsuitable. Updates to an object are often so frequent that, by the time the system realises that a packet has been lost, another update will already have occurred. Standard reliability would re-send a stale update in favour of a more recent one.

What is needed, instead, is a system which ensures that only the *most recent* update to an object is eventually delivered.

A simple way is to have every recipient acknowledge updates it receives. The sending host sets up a timer, which is reset whenever a new update is sent out. If the timer triggers, the library re-sends the most recent update to any hosts which have not acknowledged it.

As it stands, this causes a great deal of traffic, as every update will prompt an unicast acknowledgement from every recipient. The overhead of monitoring acknowledgements and accessing the timer could also be a problem.

A better alternative is to use negative acknowledgement. Recipients would send a negative acknowledgement message if an update, which was expected, has not been received. The heuristic for calculating this expectation must be accurate. At a first approximation, recipients can assume a fixed update rate, but allows for slight variation. The object owner notifies recipients (using a multicast message), when updates seem to have ceased for a while.

Latency tolerance

Section 4.2.3 introduces the *latency tolerance* value associated with an object. It is currently used only to lower traffic by limiting the update rate of an object.

The latency tolerance value could also be used for collation. If a sending process has an idea of the communication latency between it and recipients, it can calculate by how much time the message may safely be delayed. If it may delay messages in this way, it can collate them so that multiple updates can be packed into a single packet. Since larger packets have a proportionately lower overhead, this reduces bandwidth use.

If latency tolerance values were attached to messages, and supported by the network protocol, routers could also perform this sort of collation. It would then be even more useful, as it could collate messages from different processes sent to the same recipients.

Another use for the latency tolerance value is to reduce jitter. Currently, updates are processed as soon as the library receives them, but it could instead regulate latency by delaying processing until the tolerable latency interval had elapsed. A global timebase would be required for this.

Motion prediction

Motion prediction, also known as *dead reckoning*, is a technique which decreases network traffic [25, 52, 56]. The effect it has on latency was discussed in Section 2.2.2. Motion prediction is not provided by DSMVE, but can be done by the application programmer.

Instead of storing the position of an object and updating that whenever the object moves, motion prediction stores additional details, such as the object's velocity. In the absence of data to the contrary, readers update the object's position according to the prediction parameters. The sender also performs prediction, so that it can compare the predicted result with the true result.

Communication of the object's true position is then only necessary when the positional error becomes significant. One must allow for the cumulative error caused by different processes doing the prediction at slightly different times, so the object's position should also be updated often enough to cater for this. That is, one should impose a minimum update frequency on the object's position.[5] This is easily done in DSMVE by modelling the object's position using a shared object. The minimum update frequency is controlled using the tolerable latency value.

So simple first-order prediction can be done by modelling an object's true position and velocity, as well as its predicted position. The true position and velocity are the communicated state of the object, while the predicted position is calculated locally. Movement of the object updates these, and the tolerable latency determines how often updates cause communication.

Animation

One side-effect of latency tolerance (see Section 4.2.3) is that it disrupts the apparent smoothness of an object's motion. While the tolerable latency for movement of an object may be quite low, the human visual system needs a high frequency of motion steps in order to conclude that an object is moving smoothly.

This can be solved through reader-side interpolation; that is, a reader interpolates the position of an object between successive position updates. Again, DSMVE does not provide this, but it can be done by the programmer.

In practice, one would expect interpolation to be combined with motion prediction, since interpolation based on an object's last two known positions would increase latency. Interpolation would be easy to add to motion prediction, simply by reducing the motion prediction step size on the reader's end.

[5]The result is a low-frequency, regular update, which is known as a *heartbeat* [25].

One other advantage of this is that it reduces the visible effects of jitter, since smoothness of motion is no longer affected by communication latency variation.

5.4.3 Scalability

DSMVE is scaleable over distance, because latency is the inhibiting factor,[6] and DSMVE was designed to keep latency as low as possible. (The latency of DSMVE is discussed in Section 6.2.)

However, DSMVE's current implementation is not scalable to large numbers of participants. This kind of scaleability is examined in the following sections.

One way of improving scaleability is to allow parts of the system to be adaptive, i.e. to allow performance to degrade smoothly if necessary. Ways of doing this — grouping objects into impostors, and reducing the update rate — have already been mentioned in Section 4.2.3.

The following sections give other ways to make the system scaleable.

Explicit multicast

An object update in DSMVE is an explicit multicast operation — the sender can name the recipients. The sender can therefore specify, on each update, exactly which recipients must receive the message. This allows for the filtering techniques discussed in Section 4.2.3.

However, the update operation is not itself implemented using an explicit multicast protocol, as there is no such protocol available. Instead, it uses repeated unicasts.

This is very bandwidth-intensive; one would expect $O(n^2)$ traffic for n hosts. In addition, the time taken to issue n unicasts is $O(n)$. If n is large, this will become a significant factor which increases communication latency. In other words, multicast using repeated unicasts is not scaleable.

[6] While bandwidth use becomes more important as distance increases (as wasting bandwidth affects a larger number of systems), bandwidth has no immediate upper bound.

An improvement would be to use subscription multicast. One would create a single multicast group, to which all participating hosts subscribe. A single multicast would then suffice to send a message to all desired recipients.

The problem with this is that each message reaches *every* participating host. An addressing mechanism would be used so that hosts which were not meant to receive the message simply ignore it. However, it still requires every host to process every update sent in the virtual environment.

As argued in Section 3.2, what is needed is the capacity to specify exactly which machines must receive a message, i.e. a protocol-supported explicit multicast.

Changing the Internet protocol to support explicit multicast would take years, and getting the new protocol widely-deployed would take even longer. The best approximation is to use many subscription multicast groups, choosing them so that they change infrequently and fit the expected recipient groups as well as possible.

Removing the central server

For update distribution, DSMVE favours direct peer-to-peer communication over client/server distribution. The main reason is lower latency, as explained in Section 4.2.4.

A second reason for this choice is that, in client/server distribution, the central server can quickly become a bottleneck. Using a hierarchy of servers [52] alleviates the problem, but more and more levels would need to be added in order to accommodate more and more participants. Increasing the number of levels increases the communication latency, so this solution is not very scaleable.

DSMVE avoids this major bottleneck, but still uses a central server for the less frequent task of claim arbitration. While this is not as dire, it is nevertheless a limit to scaleability.

To make the system properly scaleable, the central server should be eliminated. One way, made possible by DSMVE's fine granularity, is to distribute the arbitration task among clients. A simple arbitration rule would be that when a process relinquishes ownership of an object, it becomes the arbiter for the next claim operation. If an arbiter disconnects, it must pass the arbitration responsibility on to another process.

Non-blocking arbitration

When the server handles a claim request, it blocks until its reply has been successfully delivered (see Figure 10 on page 44). This is to avoid processing a registration of the same object in the critical interval between the server assigning the owner and the new owner becoming aware of the fact.

This blocking will exacerbate the server bottleneck problem. If arbitration were instead distributed, as suggested in the previous section, blocking of the arbiter will delay the application (unless multithreading could be employed).

Therefore, blocking on the part of the arbiter should be removed. One solution is that any process waiting on a claim operation should buffer 'registration notification' messages for that object, and process them once the claim has finished.

5.4.4 Other limitations

The following sections describe some miscellaneous future improvements to DSMVE.

The relinquish operation

When relinquishing an object, the client currently blocks until the server receives the message. This is to prevent the client from exiting while it could potentially have duties to fulfill as owner of the object. (One such duty is to respond to 'registration notification' messages.)

This blocking could be avoided, for example through compiler support which checks with the server before attempting to do anything which assumes that the relinquish message has already been received.

Fragmentation

The coding of DSMVE uses the `NetworkBuffer` class to encapsulate a buffer into which library and object state data is packed, for communication across the network. It currently assumes that the data in a `NetworkBuffer` will fit into a single packet.

This limitation should be removed, by allowing `NetworkBuffer` to be arbitrarily extended, and by adding protocol which fragments it into separate packets for sending, and reassembles the packets into a `NetworkBuffer` on receipt.

Fault tolerance

As with most virtual environments, DSMVE is not fault-tolerant. That is, if a process fails, the protocol will not recover. As we wish DSMVE to be highly scaleable, fault tolerance should be investigated.

Complete fault tolerance will require that any duty of a process can be assumed by some other process if the first one fails. This would be a major undertaking, but may benefit from related work in distributed systems research, particularly in distributed file systems.

Security

Like most virtual environments and DSM systems, DSMVE has no support for security. DSM research does not often mention security, but it will become a concern in large environments.

User authentication will be important. For example, it would allow an accounting system so that participants could pay to attend a virtual conference.

Besides such all-or-nothing security, various levels of security may be desirable. One could imagine imposing a *minimum* latency on certain updates — for example, on stock market prices, as currently happens on the Internet. Different people would be allowed different minimum latencies.

As with fault tolerance, adding security to DSMVE would be a major undertaking, but it may benefit from related distributed systems research.

Chapter 6

Results

This chapter summarises the results of the thesis, in the context of the aims given in Chapter 1.

6.1 Ease of programming

The primary goal of this research is to make distributed virtual environments easier to program. The approach taken was to produce a DSM system suitable to virtual environments.

In general, the DSM paradigm is easier to use than the ad hoc 'message-passing' paradigm [4, 9, 32, 34, 48, 60]. Experience with the sample applications (Chapter 5) confirms that this is true of virtual environments in particular.

This is the case not only because DSM provides a higher-level abstraction which hides communication, but also because the DSM abstraction correlates with the virtual environment concept of a 'shared world' — if virtual environment objects are modelled with DSM objects, the one-to-one corellation makes code very straightforward.

In fact, the programming interface of DSMVE is very simple. Once the programmer has created classes for the DSM objects, the use of those classes is identical to the use of local classes. The 'claim' and 'relinquish' operations of DSMVE encapsulate the ownership mechanism which would otherwise have to be implemented by the application programmer (Section 2.2.1).

The exception to this simplicity lies in operations such as registration, initialisation, and dynamic allocation. These should be improved by developing a compiler or at least a preprocessor which is aware of the DSM, as mentioned in Section 5.4.

6.2 Low latency

Implicit in the primary goal of the thesis is that the solution found must be usable for virtual environments. As discussed in Section 2.2.2, this means that the system must have low latency. Since the system must also be as scaleable as possible, the widely-distributed case is important, and in such a case, the most significant contribution to latency is communication latency.

Section 2.1 examines latency, and shows that causal chains of messages must be scrutinized if latency is to be kept low. It concludes that only lowering the communication latency, the processing latency, or the number of messages, will lower the overall latency of an operation. Communication latency has a lower bound which is already being approached (Section 2.1.1), and for simple operations like DSM reads and writes, the processing latency is relatively small. However, the third factor — the number of messages in a message chain — can be examined and manipulated.

Section 2.3.2 discusses DSM protocols with a view to this idea, and contains support for the following argument.

Since virtual environments present a frequently-updated display to the user, they exhibit a high read-to-write ratio, and read latency is especially critical. This means that the system should be fully replicated and push-based. In addition, cached copies should be updated eagerly — this is done by the 'claim' operation.

Finally, the latency of write operations is also important. A write operation will need a chain of at least two messages if copies are to be kept consistent. However, because of the desire to give people complete control of objects they are manipulating (the ownership mechanism), write operations show strong locality. So DSMVE does not maintain consistency on each write, and uses the ownership mechanism to avoid permanent inconsistency. Hence, a write operation needs only one message, and involves no blocking.

It is then easy to compare the latency of this solution (in terms of the length of causal chains) with that of other virtual environments. Like many [30, 54], reads are all local, and writes are direct, involving only one message. With systems which use client/server distribution [52], in which the message chain for writes is longer, DSMVE actually has lower latency.

6.3 Scalability

DSMVE is scaleable over distance, by reasoning given in Section 5.4.3. However, scalability over the number of participants was compromised in DSMVE.

The main reason for this is that DSMVE used unicasts for update distribution. Using unicasts to send an update to many recipients is wasteful of bandwidth as well as latency. Section 5.4.3 gives a simple improvement, used by some environments (such as [54]), which uses a single subscription multicast group. However, this is still not very scaleable. A better solution involves the provision of protocol-supported explicit multicast, as described in the same section.

Other hindrances to scaleability are more minor. If the above distribution problem is solved, the next obstacle to scaleability may well be the use of a central server for claim arbitration. Section 5.4.3 discusses ways of removing this obstacle.

6.4 Portability

DSMVE is fully portable, as it makes provisions for encapsulating architecture-dependent structures so that they are transferred correctly. Because of this, applications compiled on different architectures can participate in the same environment. All that is lacking is similar code for types other than integers, such as floating point numbers and strings. The main decision needed here is an architecture-independent standard.

Another portability issue is the networking application programmer's interface (API) used. DSMVE uses Berkeley sockets, so it is portable to any platform which supplies this API. For other platforms, code will need to be rewritten. Since DSMVE's layered

design ensures that all calls to network API's occur in the bottom layer (see Figure 9 on page 39), only the lower layer would need to be modified.

6.5 Efficient bandwidth use

In its basic form, environments written with DSMVE are not as bandwidth-efficient as some virtual environments. Environments like RTime [52] use client/server distribution to reduce bandwidth use (at the expense of latency).

However, the comparison is not so simple. With protocol-supported explicit multicast, DSMVE would be far more efficient than it is now. In addition, explicit multicast would give a sending process precise control over the recipients, enabling performance improvements which are not possible without this control. Examples are given in Section 3.2.

One other potential improvement is the use of the *latency tolerance* value to group updates together. This is described in Section 5.4.2.

In conclusion, while DSMVE sacrificed bandwidth to achieve the more important goal of low latency, at least some of that may be regained through future work. It is even possible that DSMVE's approach could be more efficient than that of other environments, if explicit multicast is used wisely.

Chapter 7

Conclusion

7.1 Overview

This research has examined the design and implementation of a low-latency distributed shared memory for use in writing virtual environments.

Section 2.1.1 showed that it is important to minimize latency. Virtual environments are sensitive to latency, of which communication latency is the most significant contributing factor. Since the communication latency of the Internet is already nearing physical lower limits, latency is not a problem that will be solved by improvements in the network infrastructure.

This led to the design of a distributed shared memory system that is push-based and non-consistent. This decision is supported by the observation that some virtual environments are converging to such a design (Section 3.1).

Though a virtual environment can tolerate temporary inconsistency, there is still a need for control to avoid permanent inconsistency. The mechanism chosen for this, *ownership*, is natural to virtual environments since they often need such arbitration anyway.

The design was implemented as a C++ library called DSMVE. This work brought to light a number of implementation concerns.

Chief among these is the unsuitability of high-level protocols like TCP to virtual environments. Such protocols provide total ordering and reliability, both of which are

unnecessary and induce latency. The result is that the implementation of DSMVE has had to provide its own ordering and reliability systems, ones which are mindful of latency.

Another major concern is poor multicast support. Subscription multicast, which is provided by TCP/IP, is not scalable. Instead, explicit multicast was proposed as an alternative multicast addressing scheme, which does not have the scalability limitations of subscription multicast. Provision of an explicit multicast protocol is a research topic in itself, so DSMVE was written without multicast — an important deficiency which should be addressed in the future.

Using DSMVE, two sample applications were developed, in order to test the ease-of-use of DSMVE and the suitability of distributed shared memory to virtual environments. With this experience, limitations of DSMVE were identified, together with avenues for future enhancements.

7.2 Results

The shared memory paradigm as implemented here was found to be easy to use. This was achieved without increasing communication latency, through the use of push distribution, and by abandoning consistency in favour of ownership.

Scalability was compromised in DSMVE, because of inadequate multicast support. Instead, a concrete plan for a scalable multicast — explicit multicast — was outlined. Development of explicit multicast could take a long time, but should substantially improve scalability.

DSMVE is portable, as it encapsulates architecture-dependent structures so that their translation is hidden. All that is needed is to extend the code to include types like floating point numbers and strings.

Finally, the last subgoal of the research, efficient bandwidth use, was addressed in numerous areas. In particular, Section 5.4.2 described the use of a *latency tolerance* value for each object, in order to reduce traffic.

7.3 Future work

A much-needed addition is improved dynamic allocation support. Currently, dynamic allocation in DSMVE is cumbersome, and no support is provided for obtaining a list of the currently-allocated objects in the entire system. Section 5.4.1 proposed a solution in which system-supported, extensible containers are provided.

Another enhancement would be to extend the language and provide compiler support for the distributed shared memory. Many areas were identified in which compiler support could significantly increase ease-of-use or efficiency. In particular, compiler support would allow automation of the marshalling functions, registration, and object updates.

Finally, it should be noted that as virtual environments grow in size, the need may grow for persistence. That is, users will want changes to be permanent, and programmers will naturally want persistence to be transparent. If this is put together with the envisioned future attributes of DSMVE, the result is a widely-distributed, object granular, persistent system with container support.

Conceptually, this is not far from a distributed database or a structured distributed file system. While these are still visibly different — distributed databases are usually consistent, many distributed file systems are not replicated — one gains the impression that these fields are converging.

Whether or not this is true, the field of virtual environments can benefit from examination of these other fields. Features like persistence, fault tolerance and security have become well established in these other fields. While such features are seldom provided by current virtual environments, they should become important as virtual environments grow in size and become more widely used.

Bibliography

[1] Henri E. Bal, Raoul A. F. Bhoedjang, Rutger Hofman, Ceriel Jacobs, Koen G. Langendoen, Tim Rühl, and M. Frans Kaashoek. Orca: a portable user-level shared object system. Technical Report IR-408, Vrije Universiteit, Amsterdam, June 1996.

[2] Shaun Bangay, James Gain, Greg Watkins, and Kevan Watkins. RhoVeR: Building the second generation of parallel/distributed virtual reality systems. In *First Eurographics Workshop on Parallel Graphics and Visualisation*, Bristol, UK, 26–27 September 1996.

[3] K.P. Birman. A response to Cheriton and Skeen's criticism of causal and totally ordered communication. *ACM SIGOPS Operating Systems Review*, pages 11–21, 1993.

[4] R. Bisiani and A. Forin. Multilanguage parallel programming of heterogeneous machines. *IEEE Transactions on Computers*, 37(8):930–945, August 1988.

[5] Jean-Yves Le Boudec. The asynchronous transfer mode: a tutorial. *Computer Networks and ISDN Systems*, 24:279–309, 1992.

[6] Donald P. Brutzman, Michael R. Macedonia, and Michael J. Zyda. Internetwork infrastructure requirements for virtual environments. In *Proceedings of the Virtual Reality Modeling Language (VRML) Symposium*, San Diego, CA, December 13–15 1995. San Diego Supercomputer Center (SDSC).

[7] Scott Burleigh. ROME: Distributing C++ object systems. *IEEE Parallel and Distributed Technology*, pages 21–32, May 1993.

[8] Carnegie Mellon University. The Amulet user interface development environment.
http://pecan.srv.cs.cmu.edu/afs/cs/project/amulet/www/amulet-home.html.

[9] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared-memory systems. *ACM Transactions on Computer Systems*, 13(3):205–243, August 1995.

[10] Sheue-Ling Chang, David Hung-Chang Du, Jenwei Hsieh, Rose P. Tsang, and Mengjou Lin. Enhanced PVM communications over a high-speed LAN. *IEEE Parallel and Distributed Technology*, pages 20–32, Fall 1995.

[11] David R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[12] David R. Cheriton. Dissemination-oriented communication systems. DSG Working Paper.
ftp://ftp.dsg.stanford.edu/pub/papers/dissem.ps.Z, 1992.

[13] David R. Cheriton and Dale Skeen. Understanding the limitations of causally and totally ordered communication. In *Proceedings of the 11th ACM Symposium on Operating System Principles*, December 1993.

[14] Stuart Cheshire. Bolo.
http://deckard.mc.duke.edu/bolo/.

[15] Stuart Cheshire. Latency and the quest for interactivity, November 1996.
http://rescomp.stanford.edu/~cheshire/papers/LatencyQuest.html.

[16] Yong-Kim Chong and Kai Hwang. Performance analysis of four memory consistency models for multithreaded multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(10):1085–1099, October 1995.

[17] C. F. Codella, R. Jalli, L. Koved, and J. B. Lewis. A toolkit for developing multi-user, distributed virtual environments. In *IEEE Virtual Reality Annual International Symposium*, pages 401–407, September 1993.

[18] Mark R. Cutkosky, Jay M. Tenenbaum, and Jay Glicksman. Madefast: Collaborative engineering over the Internet. *Communications of the ACM*, 39(9):78–87, September 1996.

[19] Michael J. Dalpee and T. James Cannaliato. Beyond RPC: The virtual network. *IEEE Parallel and Distributed Technology*, pages 41–57, November 1993.

[20] José Miguel Salles Dias, Ricardo Galli, António Carlos Almeida, Carlos A. C. Gelo, and José Manuel Rebordão. mWorld: A multiuser 3D virtual environment. *IEEE Computer Graphics and Applications*, 17(2):55–65, March–April 1997.

[21] Michel Dubois, Jonas Skeppstedt, and Per Strenström. Essential misses and data traffic in coherence protocols. *Journal of Parallel and Distributed Computing*, 29:108–125, 1995.

[22] Clarence A. Ellis, Simon J. Gibbs, and Gail L. Rein. Groupware: Some issues and experiences. *Communications of the ACM*, 34(1):38–58, January 1991.

[23] Brett D. Fleisch and Randall L. Hyde. Mirage+: A kernel implementation of distributed shared memory on a network of personal computers. *Software–Practice and experience*, 24(10):887–909, October 1994.

[24] Michael Franklin and Stan Zdonik. Dissemination-based information systems. *IEEE Data Engineering Bulletin*, 19(3), September 1996.

[25] Rich Gossweiler, Robert J. Laferriere, Michael L. Keller, and Randy Pausch. An introductory tutorial for developing multi-user virtual environments, 1994. http://www.cs.virginia.edu/~rg3h/networkVR/paper.html.

[26] Michael Greenwald and David Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, pages 123–136, Seattle, October 1996. USENIX.

[27] Andrew S. Grimshaw, W. Timothy Strayer, and Padmini Narayan. Dynamic, object-oriented parallel processing. *IEEE Parallel and Distributed Technology*, pages 33–47, May 1993.

[28] Feng Huang and Jean Bacon. Operating system support for flexible coherence in distributed shared memory. In *Proceedings of 29th Hawaii International Conference on System Sciences*, volume 1, pages 92–101, Hawaii, USA, 3–6 January 1996.

[29] Feng Huang, Jean Bacon, and Glenford Mapp. Virtual memory support for distributed computing environments using a shared data object model. *Distributed Systems Engineering*, 2(4):202–211, December 1995.

[30] ID Software. Quake.
http://www.idsoftware.com/.

[31] Kirk L. Johnson. High-performance all-software distributed shared memory. Technical Report LCS-TR-674, MIT Laboratory for Computer Science, December 1995.

[32] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. An evaluation of software-based release consistent protocols. *Journal of Parallel and Distributed Computing*, 29:126–141, 1995.

[33] Jon Knight and Steve Guest. Using multicast communications to distribute code and data in wide area networks. *Software–Practice and Experience*, 25(5):563–577, May 1995.

[34] Leonidas I. Kontothanassis and Michael L. Scott. High performance software coherence for current and future architectures. *Journal of Parallel and Distributed Computing*, 29:179–195, 1995.

[35] Anders Kristensen and Colin Low. Problem-oriented object memory: Customizing consistency. In *OOPSLA '95*, Austin, Texas, 1995. ACM.

[36] Wim Lamotte, Eddy Flerackers, Frank van Reeth, Rae Earnshaw, and Joao Mena de Matos. Visinet: Collaborative 3D visualization and VR over ATM networks. *IEEE Computer Graphics and Applications*, 17(2):66–75, March–April 1997.

[37] Koen G. Langendoen, John W. Romein, Raoul A. F. Bhoedjang, and Henri E. Bal. Integrating polling, interrupts, and thread management. In *Frontiers '96*, pages 13–22, Annapolis, Maryland, V.S., 1996. IEEE.

[38] Steven M. Lehar and Gregory W. Lesher. Xbattle. http://cns-web.bu.edu/pub/xpip/html/xbattle.html.

[39] Willem G. Levelt, M. Frans Kaashoek, Henri E. Bal, and Andrew S. Tanenbaum. A comparison of two paradigms for distributed shared memory. *Software–Practice and Experience*, 22(11):985–1010, November 1992.

[40] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[41] Michael R. Macedonia and Stefan Noll. A transatlantic research and development environment. *IEEE Computer Graphics and Applications*, 17(2):76–82, March–April 1997.

[42] Michael R. Macedonia and Michael J. Zyda. A taxonomy for networked virtual environments. In *Proceedings of the 1995 Workshop on Networked Realities*, October 1995. Boston, MA, October 26-28.

[43] 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. *IEEE Computer Graphics and Applications*, pages 38–45, September 1995.

[44] Silvano Maffeis. ELECTRA — making distributed programs object-oriented. In *USENIX Experiences with Distributed and Multiprocessor Systems*, 22-23 September 1993.

[45] Thomas W. Malone, Kum yew Lai, and Christopher Fry. Experiments with Oval: A radically tailorable tool for cooperative work. *ACM Transactions on Information Systems*, 13(2):177–205, April 1995.

[46] Bruce E. Martin, Claus H. Pedersen, and James Bedford-Roberts. An object-based taxonomy for distributed computing systems. *Computer*, 24(8):17–27, August 1991.

[47] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1.* Addison-Wesley, 1993.

[48] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, August 1991.

[49] Karin Petersen and Kai Li. Multiprocessor cache coherence based on virtual memory support. *Journal of Parallel and Distributed Computing*, 29:158–178, 1995.

[50] Jelica Protić, Milo Tomašević, and Veljko Milutinović. Distributed shared memory: Concepts and systems. *IEEE Parallel and Distributed Technology*, pages 63–79, Summer 1996.

[51] Umakishore Ramachandran, Gautam Shah, Anand Sivasubramaniam, Aman Singla, and Ivan Yanasak. Architectural mechanisms for explicit communication in shared memory multiprocessors. In *Supercomputing*. ACM, 1995.

[52] RTime. Technical white paper, February 1997. http://www.rtimeinc.com/html/wp_cover.html.

[53] Harjinder S. Sandhu. Algorithms for dynamic software cache coherence. *Journal of Parallel and Distributed Computing*, 29:142–157, 1995.

[54] Chris Schoeneman. Bzflag. http://reality.sgi.com/crs/bzflag.html.

[55] Silicon Graphics, Inc. Just play: Brave new worlds, 1997. http://www.studio.sgi.com/Features/Games/bnw.html.

[56] Sandeep K. Singhal and David R. Cheriton. Using projection aggregations to support scalability in distributed simulation. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, Hong Kong, May 1996. IEEE Computer Society.

[57] Mirjana Spasojevic and M. Satyanarayanan. An empirical study of a wide-area distributed file system. *ACM Transactions on Computer Systems*, 14(2):200–222, May 1996.

[58] Stardust Technologies, Inc. The IP multicast initiative. http://www.ipmulticast.com/.

[59] W. Richard Stevens. *UNIX network programming.* Prentice-Hall, Inc., 1990.

[60] Michael Stumm and Songnian Zhou. Algorithms implementing distributed shared memory. *Computer*, 23(5):54–64, May 1990.

[61] The 3DO Company. Meridian 59.
http://www.3do.com/meridian/.

[62] Chandramohan A. Thekkath and Henry M. Levy. Limits to low-latency communication on high-speed networks. *ACM Transactions on Computer Systems*, 11(2):179–203, May 1993.

[63] Robert van Liere and Jarke J. van Wijk. CSE: A modular architecture for computational steering. Technical Report CS-R9615, Centrum voor Wiskunde en Informatica, 1995.

[64] Josie Wernecke. *The Inventor mentor : programming Object-oriented 3D graphics with Open Inventor, release 2.* Addison-Wesley, 1994.