University of Oslo
Department of Informatics

A Distributed Persistent World Server using Dworkin's Generic Driver

Geir Harald Hansen

Cand. Scient. Thesis

July 31, 2002

Foreword

This is a thesis submitted to the University of Oslo in partial fulfillment of the requirements for the degree of Cand.Scient.

It is satisfying to have completed a project that required a significant amount of work. Even if motivation was not always with me, the topic was one that I found truly interesting. Perhaps the most interesting part was seeing the ideas working in practice in the implementation. Now that it is done, I'd like to thank the people who helped me along the way.

I would like to thank Ragnar Normann for being my advisor on this thesis project. Thank you for your constructive comments and your positive attitude towards students doing their own thing.

I want to thank Felix A. Croes, the author of Dworkin's Generic Driver (DGD). Thank you for answering my questions and for the very quick bugfixes when my implementation used features that were still experimental in DGD. Finally, thanks for DGD which I often used and enjoyed also before starting on this thesis.

My thanks also to Joan Tortorello, Martin Tostrup Šetek and Michael Jonas for their feedback. With your help this paper's readability was heightened significantly.

Abstract

This thesis describes the design and implementation of a persistent distributed object oriented system and programming environment. It is built on an existing non-distributed server, Dworkin's Generic Driver (DGD). Focus is placed on achieving high efficiency, while preserving single-threaded semantics, allowing the programmer to write code as if there was only a single server. The goal is to design a server solution capable of supporting large persistent worlds, such as those found in massively multiplayer online games (MMOG). Results from testing the implementation are presented and evaluated.

Contents

Chapter 1

Introduction

This paper discusses aspects of creating a distributed system supporting a persistent world, more specifically, one built upon Dworkin's Generic Driver (DGD [4]). Persistent worlds are introduced in the following section, while the relevant details of DGD are investigated in section 1.5. The computers supporting the world in the form of servers will be referred to as server nodes, or simply nodes.

1.1 Background

"Persistent world"; the type of world this refers to is a virtual reality, shared by many users simultaneously. These users may experience it, change it and interact with each other. Persistence here means that the objects that make up this world are stored in a database. So the world stays the same after a shutdown and restart of the server, including a reboot of the host computer. Technically, this "world" is just a database. What makes it a world is the user's perception of it. This does not imply any specific way of presenting the world to the users, which might be textual, graphical or otherwise.

These types of system have been used extensively for text games, but also as teaching ("virtual university") and collaboration tools. A new class of games has also appeared and become quite popular, often referred to as MMORPG – massively multiplayer online roleplaying game. "Massively multiplayer" denoting the fact that there is a large number of users participating simultaneously.

As persistent worlds grow beyond a certain size and get more complex, they cannot run on a single server. This is already true of the MMORPGs, which are distributed in varying ways. Often, though, this is combined with making several completely independent copies of the world. The downside is obvious; two users can never interact unless they are in the same world copy. Also, the transition in the world be-

tween two server nodes is seldom seamless.

For large persistent worlds, distribution is not only useful, but an absolute requirement. That makes it very interesting to investigate ways that such distribution can be done. DGD provides a good software foundation upon which distributed worlds can be created and run. However, it was made to run on a single host computer.

1.2 Objectives

The main objective of this thesis is to develop a system on top of DGD for supporting distributed persistent worlds. A simple proof–of–concept implementation will also be created, and some aspects of its behavior observed.

A persistent world is a highly interactive medium. In this context response time is very important. Response to user input should, whenever possible, come fast enough to be perceived as immediate. Greater demands for speed are placed upon a distributed system of this kind than may be the case for many other types of distributed systems. For this reason, speed and efficiency are aspects that will be focused on.

The goal is to present a layer to the programmer, where well behaved objects will function properly in a distributed environment. This layer will not be invisible, but it should not without necessity get in the way. Making it possible to program the system in a fashion as close as possible to traditional DGD programming would be beneficial. An important plus here is if the single-threaded semantics of DGD (see section 1.5.4) can be kept. Doing so would allow developers to program the system without worrying about deadlocks or any other problems related to the coordination of multiple threads.

In summary, the solution should have the following qualities:

- One persistent world on multiple server nodes.
- Persistent as a whole.
- Stay close to traditional DGD programming.
- Single-threaded semantics.
- Efficient enough for a highly interactive system.

The purpose of the proof–of–concept implementation is to try the solutions in practice and see whether they have the qualities listed above.

1.3 Scope and limitations

First of all, DGD being the basis for building upon will affect choices on the way towards a solution. Building on a different basis might make other solutions more suitable. Another approach might have been modifying DGD itself, rather than building on top of it.

The server nodes are assumed to be interconnected through a local area network (LAN) or a similar configuration. Connectivity is assumed to have low latency and high bandwidth.

Node failures and failures in the network connecting the nodes will both be considered fatal, causing all remaining nodes to shut down.

Security aspects will not be examined. Total trust between the server nodes is assumed.

DGD is programmable in nature. It may compile new code or update old code while the system is running. It will be assumed that a source code repository is kept on a fileserver that all server nodes have access to through a networked filesystem. An alternative, which will not be examined here, would be to transfer an object's code in addition to its internal state when it is moved from one node to another. Perhaps sending code only when the same version of that code is not already present on the destination node.

It may be possible to implement distribution in DGD in a way transparent to the programmer. Even if it is possible to do this in an efficient way, it is beyond the scope of this project and will not be attempted here. It should be kept in mind, though, that a completely transparent system may not always be a good thing. Transparency can lead to inefficient code, because the consequences of a piece of code are less visible to the programmer. This makes it easy to treat local and remote objects the same and forget about the very different cost involved with accessing them.

The persistent world can be made up of room objects. Connections between these room objects allow users to travel from one place to another. The proof–of–concept implementation will only have a few rooms and other objects which can be manipulated in simple ways. It will be tested by a very limited number of users. Observations from such a system are bound to be of limited use.

A system such as this has little meaning if one server node ends up handling everything. Therefore a scheme for dividing work is necessary. This is not the main focus here, however, and advanced load balancing is not attempted.

1.4 Organization of the paper

The following chapters are organized as follows. Chapter 2 examines relevant prior work, while chapter 3 contains the analysis and design decisions. The most relevant parts of the implementation are covered by chapter 4. Chapter 5 presents and evaluates testing results and other observations. Finally, chapter 6 looks at possible directions for future work.

1.5 DGD: Dworkin's Generic Driver

Dworkin's Generic Driver (DGD) is a multi-user, programmable server for persistent worlds. DGD supports a persistent world on a single host.

DGD is similar to Java in that its compiler generates bytecode which is later run by an interpreter as part of a virtual machine. However DGD's compiler is built into the server; new objects are compiled and old ones upgraded with new code while the server is running. DGD uses a language called LPC which has a syntax similar to C. It is object oriented and has multiple inheritance. The LPC language was designed by Lars Pensjö. Its first implementation was in Lars Pensjö's own LPmud driver [9]. DGD uses a somewhat modified dialect of LPC.

Arrays and mappings (associative arrays) are automatically garbage collected when no longer referenced. The same is not true of objects however, which have to be explicitly destructed. Existing references to an object automatically become "nil" (a special null value) when the object they reference is destructed.

Persistence is provided through a feature called state dumps, which is explained in detail in section 1.5.2.

1.5.1 Communication

DGD uses TCP/IP for communication. For security reasons it does not in itself have the capability of opening outbound connections. It opens two TCP ports. One to receive telnet connections and one to receive binary (not interpreted by DGD itself) TCP connections.

There is an independently developed networking package for DGD. It extends DGD's networking functionality, allowing it to open any number of ports, open outbound connections and receive and send UDP packets.

In a distributed system, nodes need to open outbound connections to other nodes, therefore the networking package is a natural choice. However, the package is not always available for the latest version of DGD. For this project a small program was written instead. It connects

to DGD's binary port, allowing DGD to make outbound connections, and multiplexes those connections over its single connection to DGD.

1.5.2 Persistence

As mentioned earlier, persistence is provided by DGD through a feature called state dumps. A state dump is a file containing (almost) the entire internal state of the server. After the creation of a state dump, the server may be shut down and later restarted using the state dump file. All objects and callouts (timers) are then restored.

A state dump contains all objects (their bytecode and internal state) and all pending callouts. Network connections are not stored however, and must be established anew when the system later comes up again.

1.5.3 Thread Model

Threads in DGD always execute in sequence and do not overlap each other. The thread model is event-driven. Threads are expected to run only briefly and cannot wait for events to occur. Instead, when an event occurs, a new thread is started to handle it. The thread then terminates when the event has been handled.

If a thread did not terminate it would hang the system forever. To ensure this does not happen DGD has a programming language construct to place resource limits on a thread. The resources are CPU time, measured by the interpreter in "ticks", and stackspace. If a thread runs out of either resource by running too long or nesting function calls deeper than allowed an error results which may terminate the thread.

Threads are invoked by the following events in DGD:

- When the system starts up, either for the first time or when restoring from a state dump.
- When a connection is established or closed.
- When the last part of buffered output has been sent to the network.
- When the server process receives a kill signal. This gives the system a chance to save its data before shutting down.
- A timer expired (dubbed a "callout" in DGD, a term originally derived from the BSD Unix kernel sourcecode). A callout may have no delay at all, in which case this thread is invoked as soon as possible after the current thread has terminated.

State dumps are performed immediately after the thread that requested a state dump has terminated and before executing another thread. Thus, no thread is executing at the point of the state dump.

The same holds for upgrading the code of an existing object (recompiling it). The object has the old code until the current thread (which initiated the recompilation) has terminated. When the next thread is started, the object will have the new code and behavior.

Errors in DGD are similar to the exceptions of Java or C++, but they are always of type `string`. When an error occurs, the current thread of execution is traced backwards to a point where the error is caught and execution may resume. If an error is not caught, the thread terminates.

One of DGD's most interesting features is that a function can be declared to be atomic. If an error occurs and is not caught during the execution of an atomic function, all actions performed since execution entered the atomic function are undone before the error is passed up from that function. An analogy would be the transactions of database systems.

1.5.4 Single-threaded semantics

The previous section already mentioned that on a single DGD system no more than one thread may be running at any given time. The implications of this deserve a little further examination, though.

Programming a system where you have to deal directly with the issues of multiple threads accessing the same data, usually involves using some form of locking or other concurrency control. This places an added burden on the programmer, especially with the possibility of deadlocks. Programming a system which is single-threaded or which appears that way to the programmer is much easier and more straightforward.

As an example, imagine a closed door. One user executes a thread attempting to lock the door. Another user's thread is attempting to open the door, at the same time. It seems reasonable that only one of these threads should be able to execute their user's command successfully.

In regular DGD programming, the thread opening the door would first check that the door is unlocked, then execute code that opens the door. But when transitioning to a system with multiple threads executing concurrently, the thread locking the door might execute inbetween the check and the code that opens the door. To the users it would appear as if the door was first locked, then opened. To avoid this, the underlying system could handle the issues of concurrency on its own, preserving single-threaded semantics. Or it could present the programmer with thread synchronization primitives. In the latter case it would be the responsibility of the programmer to make sure only one of these threads could access the door at a time.

Chapter 2

Related Work

This chapter reviews prior work that is relevant to this paper. Much research has been done in the field of distributed systems, but only that which was found most directly useful for this thesis is discussed here. During the course of this thesis project, one prior project was found that had many similarities; the DOME project, which is investigated in the following section.

2.1 DOME

The DOME [10] project's goal was to make a version of Lars Pensjö's LPmud driver capable of parallel / distributed execution, while not requiring programmers to worry about synchronization and deadlocks. DOME has a strong focus on keeping single-threaded semantics (see section 1.5.4) while staying efficient.

LPmud is a starting point similar to DGD and the goals are similar to those of this paper. DOME, however, takes a very different approach in that the LPmud driver itself is modified on a low level rather than building on top of it with LPC. It would have made a very interesting case for comparison of performance and experiences, but unfortunately the implementation work seems to have been abandoned.

To provide the programmer with conventional single threaded programming semantics, DOME handles the problems associated with concurrent access to data transparently. This is done with an optimistic concurrency control in the form of a lock-less protocol which rolls back changes as necessary, when multiple threads are in conflict. When two threads attempt to make changes to the same data, all changes made by one of the threads are rolled back. That is, the system appears as if the changes never took place, much like the rollback that takes place in a conventional database when a transaction is aborted. Afterwards the thread that was rolled back can attempt to run again.

The DOME system consists of four software subsystems; communication, object management, synchronization and execution management.

The communication subsystem acts as an interface to the host machine's communication system. Output is buffered so that it may be discarded along with object changes when a thread is rolled back, as described below.

The object management subsystem handles the global name space. Objects are named by a tuple containing the ID of the server where the object was created, and a unique number on that server. The object management subsystem handles translations between such tuples and an object's position in the object table of the server where it currently resides. A forwarding mechanism is used to handle objects that have migrated to other servers.

The synchronization subsystem contains the synchronization primitives that are needed by the lock-free protocol.

The concurrent execution of threads is managed by the execution subsystem. It interfaces with application code and handles requests for object method access, instantiation and destruction as well as I/O and other services. Access to remote objects are handled on top of a traditional remote procedure call system.

When a thread modifies the state of an object, a copy is made first, and the copy is what is actually modified. At the end of the thread all the objects it modified are updated to use the new state copies, unless other threads also modified some of those objects in the meantime. In the case that an object was modified since its state was copied the thread is rolled back discarding all the new state copies. The thread is then started again from the beginning, making new copies of the object states and attempting the operations again. Using this scheme, the thread that attempts to commit first will succeed. Parallel threads accessing the same objects will fail to commit, roll back and be rescheduled — hopefully to succeed the next time.

When objects on multiple server nodes are affected, a two-phase commit protocol is used. The node that initiated the thread sends a message to the involved nodes asking whether the changes to their objects can be safely accepted. Only after each node has given a positive answer, will the initiator tell all of them to commit the changes. Again a strong analogy can be seen in traditional database systems. If one or more of the nodes give a negative answer, the thread is rolled back and attempted again, as described in the previous paragraph.

2.2 Emerald

Emerald [7, 6] is an object-based language and system designed for the construction of distributed programs. It has fine-grained mobility; not only processes, but any object including small data objects, are freely mobile. An object may be attached to another object in which case it will also be moved whenever that object moves. Emerald was designed with efficiency in mind and it was intended to run on a local area network with a modest number of nodes.

2.2.1 Object naming and object references

Objects are given unique network-wide object identifiers (OID). An OID is 32 bits and consists of an 8-bit node identifier and a 24-bit sequence number. An object reference is represented as the address of an object descriptor. Object descriptors are local to each node, so when an object reference is moved to another node it must be updated to point to the correct object descriptor on that node. Each node has a hashed access table mapping OIDs to object descriptors, which makes this translation simple.

Using a direct address rather than the OID for object references is done for performance reasons. Looking up entries in the access table need only be done when an object reference is moved between nodes. Had the object reference consisted of the OID instead, the access table would be needed for every invocation, which would imply a performance penalty.

2.2.2 Execution of threads

In Emerald, process objects make invocations on other objects which again invoke other objects and so on. Invocations may be local or remote. The arguments are object references, but the referenced objects may be passed along with a remote invocation. By default, only the references are sent. The objects are included if the programmer specified that they should be, or it was determined to be beneficial by the compiler.

It is often efficient to include the objects referenced by the arguments of an invocation to a remote node when the invocation message is sent. In most cases the remote node will subsequently invoke those objects. This may result in many remote invocations from that node back to the first node if the objects are not included.

Emerald supports three types of argument passing semantics. Call-by-remote-reference passes only the references to the remote node. Call-by-move sends also the objects referenced. Finally, call-by-visit also in-

cludes the objects, but when the invocation finishes the objects move back to the initial node automatically. Whether moving an argument object is worthwhile depends on the size of the object, current and future invocations on the object, the number of times the object will be invoked by this remote invocation and the relative costs of mobility and local versus remote invocation. It was shown [7] that call-by-move can greatly improve performance. In some cases the compiler can choose a good argument passing mode. In other cases the programmer can use application specific knowledge to choose one.

2.2.3 Locating objects

Emerald uses forwarding addresses to locate objects, as described by Fowler [5] but with some modifications.

Each object has an object identifier that is unique not only to one node, but unique to the whole network. The basic principle is that whenever an object migrates from one node to another it leaves behind a forwarding address at the node it left. Only the two nodes (source and destination) involved in the migration update their information on the object. No message is sent to other nodes. A forwarding address consists of the tuple <timestamp, node>. "Node" specifies the node it moved to while "timestamp" tells the age of the forwarding address. As an object moves, it leaves behind a trail of forwarding addresses which can later be used to locate it. Given two different forwarding addresses, the most recent one is easily found by comparing the timestamps.

The process of locating an object involves obtaining the object's forwarding address from the local node; then, if that node does not contain the object in question, repeatedly querying the remote nodes for their forwarding addresses until the object is located.

Should any of the steps fail, for instance if no forwarding address is found, a broadcast is used instead. A message is sent to all remote nodes asking for the object. Only a node which actually has the object responds. If there is no reply within a certain amount of time, a second message is broadcast to all remote nodes. This time all nodes are expected to give a reply, positive or negative. Any node not responding this time is asked again by direct communication. If the object can still not be found, it is deemed to be unavailable.

When invoking a remote object, the invocation message is sent along the chain of forwarding addresses. Only if it fails to reach the destination object this way will the location algorithm above be used.

2.3 MIT SchMUSE

MIT SchMUSE [2] is a Scheme-based Multi-User Simulation Environment, thereby its name. Scheme is a dialect of the programming language LISP. SchMUSE is described [2] as a "concurrent, distributed, delegation-based object-oriented interactive environment with persistent storage." One big difference from the type of system aimed for in this paper, is that it is designed to run in a "capricious network environment". Server nodes are expected to regularly become unavailable, and to reappear at a different networking address. Also, clients can act as servers which makes security a more difficult and important issue with MIT SchMUSE.

The delegation-based object system deserves a closer look. An object instance consists of separate nodes, each a partial object that corresponds to one of the inherited classes. A node is said to delegate to its parent nodes. Its parent nodes are its inherited classes which are not necessarily present on the same server. A distribution of these nodes across several servers is called remote delegation. Each node holds the data members of the class it represents and references to the node instances of its parent classes. This allows for extensive object and class sharing, while retaining a high degree of privacy control in a system where the servers are not all controlled by the same authority.

Object references in MIT SchMUSE are called globally unique tickets. There are object tickets that refer to objects and class tickets that refer to classes. The components of the tickets are as follows:

ObjectTicket : ClassTicket, btime, bmachloc, RealmTicket

ClassTicket : classname, btime, bmachloc, RealmTicket

RealmTicket : realmname, btime, bmachloc

In each ticket, "btime" is the time that the entity was created, its "birth time". The time is a millisecond reading of the real-time clock on the server where the entity was born. "bmachloc" is the "birth machine location", the network address of the computer where the entity was born. These two components make the tickets unique thus separating all objects, classes and realms from each other.

A realm is an own namespace with a collection of objects. Each separate realm can be stored in persistent storage and later be brought back on the network, possibly at a different network location.

In addition to the birth time and location components which makes an object ticket unique, it also contains a class ticket and a realm ticket. The class ticket is used for debugging purposes. The realm ticket, however, is used for locating the object when the object ticket is used. If

the object ticket's realm is the local realm, the object is simply looked up in the local object table. If the realm is remote, its current network location must be found, so the message (invocation) can be forwarded to it.

A realm ticket says nothing about the current network location of a realm. Locating a realm is done through a distributed hierarchical mechanism named "Central Services." When a new SchMUSE node connects to the network, it is assigned one server that acts as its realm resolution authority. The new node informs this server of the realms it is making available. From there the information is propagated through the Central Services hierarchy on demand, much the way domain name servers on the Internet function.

At the time that the SchMUSE paper [2] was written, object migration was not yet part of SchMUSE. The paper does, however, point out Emerald [6, 7] and its forwarding addresses [5] as a solution likely to be chosen for the planned object migration support.

2.4 The distributed snapshot algorithm

This algorithm was designed by Leslie Lamport and K. Mani Chandy. A description is given here that is hopefully intuitive and easy to understand. A more detailed and technical description, including a proof, can be found in the paper [3] by Lamport and Chandi.

Determining the global state of a distributed system is not easy. It is impossible for two computers to record their state, or do any other coordinated action, at exactly the same time. The reason for this is the variations in the time it takes to pass a message between the two computers. Synchronizing the clocks of two computers across a network is not possible.

When determining a global state one must also consider messages that are on the network at the time, that is, messages that have been sent but not yet received at their destination.

An obvious solution would be for all involved computers to halt normal processing, record their state and finally resume normal processing again. However, this delay can be a high price to pay for systems where a high degree of responsiveness is required. An important property of the snapshot algorithm is that it is completely transparent; normal processing is not halted or affected in any way other than the extra work of storing state.

It is not always necessary to acquire a global state that actually occurred at some point. A consistent state that may have occurred often suffices. The purpose of the snapshot algorithm is to find a consistent global state that *may* have occurred in a distributed system. Figure 2.1

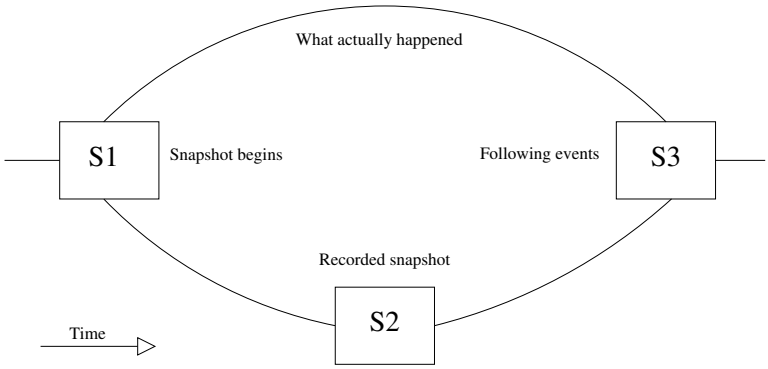

Figure 2.1: Distributed snapshot: a consistent global state that *may* have occurred

shows how the recorded snapshot may deviate from the actual situation. The snapshot preserves the internal consistency of the distributed system. The snapshot, state S2, is reachable from the initial state S1, and the future state S3 is reachable from S2. S2 is a state that may have occurred, but not necessarily one that did. S1 and S3 are actual past and future states, respectively.

The snapshot may falsely record that a special condition is present, or not present, in two or more computers at the same time. This, however, can only happen if the events are not dependent on each other. If event X is a prerequisite for event Y and event Y is in the snapshot, then event X will be as well. This guarantee is important for consistency.

The distributed system consists of processes and channels. The state of a channel is said to be the messages sent along that channel which have not yet been received. Each process has a set of input channels, where it receives messages, and output channels, where it sends messages. The following assumptions are made:

- Channels and network do not fail.
- Channels are error-free.
- Messages are delivered in the order they are sent.
- Any process can reach any other process through a sequence of channels.
- The delay experienced by a message in a channel is arbitrary but finite.

Any process may initiate a snapshot and the processes continue with their regular processing while the snapshot is taken. It is the responsi-

bility of each process to record its own state and the state of all its input channels.

The processes and channels form a graph. The processes are the nodes and the channels the edges. The algorithm is based on simple graph traversal. One process takes the initiative to start the snapshot. It records its own state and sends a special marker on all its output channels. The first time a process receives the marker, it records its own state, then immediately sends the marker on all its output channels. All messages received by a node on a particular channel between the time that the node recorded its state and the time that a marker arrives on that channel, are recorded as the channel's state.

What happens when a process p receives a marker along channel c, is shown by the following pseudo-code:

if p has not recorded its state **then**

begin
p records its state
p sends the marker on all its output channels
p records that input channel c is empty
p begins registering messages received on all the other input channels

end

else
p records the state of c as all the messages received through that channel after p recorded its state

Note that all processes send one marker on each of their output channels, and receive one marker from each of their input channels. When a marker is received, it is known that the sender has recorded its state.

A snapshot is consistent if causality is preserved. If X causes Y and Y is in the snapshot, then X must be as well. That is what it boils down to. For the result to be inconsistent, X's process would have to record its state first, then event X take place, followed by event Y and finally Y's process recording its state. This cannot happen because the snapshot algorithm requires a process to send a marker immediately after recording its state, and messages are received in the order they are sent. This is a crucial part of the algorithm. The message sent by X that causes Y in the other process will be sent after the marker. When it arrives, the process where Y will occur has already recorded its state, thus neither event is recorded in the snapshot.

Chapter 3

Analysis and Design

This chapter discusses, one by one, the problem areas that had to be dealt with when designing the distributed DGD server.

3.1 Object naming and object references

In a distributed object oriented system each object must obviously have a globally unique name or identifier by which it can be referenced. This minimum requirement ensures that specific objects can be found and different objects told apart. On the other hand there are systems in which a name is more than just a unique identifier. One example is the hierarchical host names of the Internet — the name of a computer gives a clue as to which server to query to find that computer's address. The hierarchical names ensure both that the work of answering such queries is nicely divided between many servers and the correct server is easily found by clients. The size of the Internet and this system today proves how well this technique scales.

Since the goal here is a relatively small number of server nodes connected through a local network, scalability is not a great concern. Speed and flexibility are more important factors. As we shall see in the following section, binding names to locations is not the fastest way to locate objects. Our requirement, therefore, is merely that every object has a globally unique identifier that tells it apart from every other object in the distributed system.

A straightforward way of constructing a globally unique identifier is to use two elements. One element holds a unique identifier for the node where the object was created. The second element is an identifier that makes the object unique within that node. Each node generates a sequence of numbers to be used for this second identifier in objects created at that node.

MIT SchMUSE uses the location of the birth machine, coupled with

a timestamp from that machine. Object identifiers in Emerald consist of an 8-bit node identifier and a 24-bit sequence number unique within that node. This was discussed in chapter 2 in more detail.

For the DGD based system, a simple node identifier and sequence number combination will be used. When a new node is connected to the system, it is given a unique number. Whenever an object is created, the object sequence number is incremented, so that each object the node creates is given a different one. Thus an object reference is comprised of two integers, node number and object number, uniquely identifying an object. Each node holds a table that translates such references to native DGD object references for all objects present at that node at the time.

To provide new nodes with unique node numbers, a special object is created, called the central daemon. It is given a special object identifier, 0:0 — node 0, object 0. Node number 0 is never used anywhere else. This object moves freely just like other objects. It is not central in location, rather it is central in an authoritative sense. It is the only object in the system with an object identifier (OID) known to all objects. If other such special objects are needed, the central daemon can be given the responsibility of holding their OIDs and providing them to code that needs them.

3.2 Locating objects

Given that objects are allowed to move freely, being able to locate them in any given situation is not a trivial task. Because the system is intended to be highly interactive, delays should be kept as short as possible. Furthermore, local processing is fast compared to sending network messages and waiting for a reply, so it is important to keep the number of messages passed low.

In our setting, objects can theoretically move from one node to another, and just as they arrive, move again. So even constantly moving objects are possible. One option for locating objects would be to keep a supervising node for each object, perhaps the node where that object was created. That node would always know the whereabouts of the object and when that object needs to be located, its supervising node can be queried. Whenever an object moves between nodes, it would send a message to update its position with the supervising node. The problem with this and other centralized schemes is that highly mobile objects incur much overhead in the form of update messages to the central node. Also, by the time this information gets anywhere it may be outdated because the object may have moved again. It must be expected that some objects migrate often. This solution does not appear to be a good one.

A decentralized scheme would seem more suitable. Also, since a

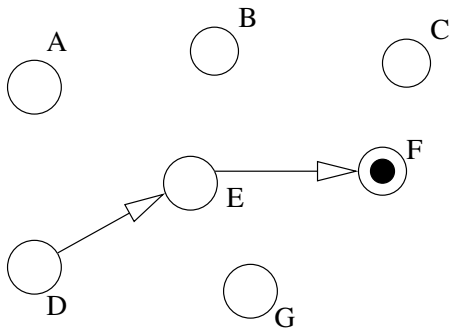

Figure 3.1: Forwarding addresses; the basic case

LAN (local area network) is assumed, broadcasting messages from one node to all the others is a possibility. Still, one should remember that having to process many broadcasted messages may hurt performance in a case with many server nodes. The more nodes in the system, the more nodes searching for objects. Avoiding excessive broadcasting will thus be important to allow more than a handful of nodes to function together.

Emerald's [6] combination of forwarding addresses and broadcasts is a good starting point. The basic idea of forwarding addresses is that when an object moves, it leaves a forwarding address pointing to the node it is moving to. As it keeps moving, it leaves a chain of forwarding addresses, which can later be used to locate the object.

Figure 3.1 shows a very basic case with forwarding addresses. The nodes are shown as circles, named with single letters. The object in question is the black spot, currently located at node F. The arrows show the forwarding addresses the object has left as it travelled from node D to E, then from E to F.

Each node has a table translating object references to forwarding addresses. If a node does not know anything about an object, there will be no entry for it. If the node knows that the object has been destructed, the special value zero is used, which is not a valid node ID. Being able to quickly determine that an object no longer exists, makes it possible to avoid doing an extensive search for the object whenever a reference to it is used. Each forwarding address is also associated with a timestamp, which purpose will be explained shortly.

Objects are located only to deliver a message from a thread, telling it that it is needed at another node. Such a message is sent when it is realized that the thread needs a remote object. Any given thread sends such a seek message to an object only once. When the message has been received, the object knows what it needs to know and further communication is unnecessary. Therefore, each search is unique by the

tuple <thread ID, node ID, object ID>, specifying which object is wanted, which thread needs it and which node the thread runs on. Each seek message also contains a timestamp, as discussed in a following section.

As a seek message arrives at a node where the object is not present, it is stored on that node and forwarded to where the forwarding address points. In this way the search forms a chain of nodes. In addition to the seek message itself each node stores pointers to the next and previous nodes in that chain. Thus, when the message has been delivered, the chain of seek messages can be followed from the point where the object was found. Each node is told to remove the seek message it has stored, and also given a chance to update its forwarding address for the object. It is important that the forwarding address is only updated if it can be replaced with information that is actually more recent. Otherwise, forwarding addresses would soon point the wrong way, and following a trail of them would not always lead to the object. A simple counter is kept as a timestamp for each forwarding address. Every time an object moves, the counter it gives to forwarding addresses is incremented. Thus the forwarding address with the highest counter holds the latest known location of the object. Fowler [5] has shown this to be sufficient.

In the case of figure 3.1, a search starting from node D would locate the object in two steps. The "seek" message would go from D to E, becoming stored at E. Then it would continue from E to F, following the forwarding addresses. As the object is found at F, the message is delivered and a "found" message is passed back. The found message contains the tuple <thread ID, node ID, object ID>, thereby uniquely identifying the search. It also contains the forwarding address to the node where the object was found, including the timestamp of that forwarding address. As the "found" message arrives at E, the stored "seek" message is removed, and the found message passed on to D, following the search chain. When D receives the found message, it will find the timestamp on the included forwarding address to be newer than the one it has. Node D then updates its forwarding address. Thus a subsequent search for the same object will be faster, not having to pass through node E.

The common case is likely a forwarding address leading directly to the object. Messages are then delivered immediately to the correct node without passing through intermediate nodes. This is likely because forwarding addresses are updated with every search and with a geographical object distribution (see section 3.7.1) a large portion of object migrations will be back and forth between two nodes, when objects are on the geographical borderline between two nodes.

If a search had started at node A, in figure 3.1, there would be no forwarding address with which to start. This is the only case when broadcasting is used. A "seek" message is sent to every other node in the system. The message is the same as other "seek" messages, ex-

cept that it can be recognized as a broadcast "seek" message. There are four possible replies to this message, and each node must reply. If the node has the object; it delivers the message to it and replies that the message was delivered. If it does not have the object, the reply either contains the node's forwarding address for that object, or says that the node knows nothing about it. The node may also reply that the object has been destructed if it has recorded that fact. If all the replies have been collected and the message was not delivered to the object, nor was the object found to have been destructed, a search can now be conducted in the usual manner. Either way, the node's forwarding address for the object will be updated when it receives replies to its broadcast.

The common case will likely be that the object is found during the broadcast and no subsequent search is necessary. This is possible because the broadcast message contains all the information of a "seek" message. Emerald does not include the request in broadcast messages; probably because the messages could get very large. In our case it means the message must hold 6 integers (thread ID, node ID, object ID, timestamp, the last two consisting of two integers each) instead of 2 (object ID). The payload of the message becomes three times as large, but it is a significant performance gain in terms of speed because it cuts down on message passing — and the message is still quite small.

Broadcasts will be implemented as multiple copies of the same message sent to all other nodes individually rather than a single "real" broadcast message addressed to all computers on the network. This allows the snapshot algorithm to fit in easily, as we shall see in section 3.6. It should be noted that network messages do not travel between nodes at a set speed. If A broadcasts a message for an object it might receive the message at node F then move to node C and receive the same broadcast message at node C. Therefore objects must be prepared to receive the same "seek" message multiple times. In theory it could even happen that the object receives the message at node F, goes to node A to service the request, then back to node F, and finally to node C where it receives the broadcast a second time. At this time, the thread requesting the object has already executed. This is extremely unlikely and not a big problem, as long as node A is prepared for the object arriving a second time to service a thread that no longer exists.

An extreme case is shown in figure 3.2. The object is in transit between F and C. As the forwarding address arrows show, the object has been moving in a circle through F, C, B and E. The "seek" messages may never catch up with the object if it keeps moving. But the object will sooner or later land on a node where the seek message has already been stored, completing the search.

The fact that an object no longer exists may be known by some nodes but not by others, as depicted by figure 3.3. Seek messages would follow

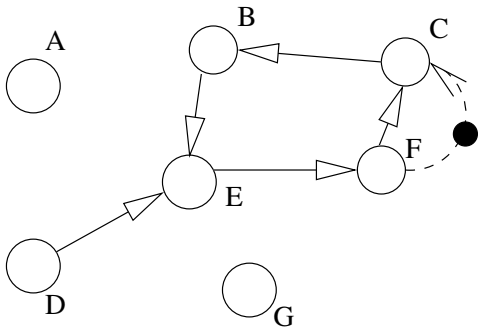

Figure 3.2: An object in transit

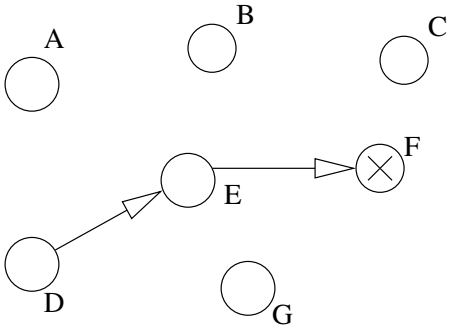

Figure 3.3: A destructed object

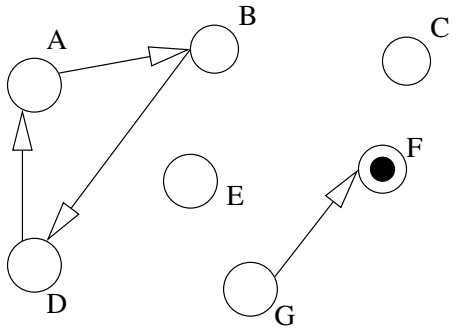

Figure 3.4: Impossible situation

the forwarding addresses as they normally do. Then when it is discovered, at node F, that the object has been destructed, a “destructed” message is passed back along the search chain. This allows both node E and D to update their information on the object. Finally node D can inform the thread in question that it is referencing a destructed object.

Our rule is that by following forwarding addresses, one eventually ends up on the node holding the object. This rule would be broken if a situation such as that in figure 3.4 could occur. Since an object always leaves a forwarding address when it moves, this could only happen by overwriting a forwarding address. As long as a node’s forwarding address points to a node which has newer information on the object, which again points to a node with newer information, the object will be at the end of the chain. This is not true if one of those nodes are “updated” with old information. Therefore, nodes must only be updated with information newer than what they already have. That is what the timestamps of the forwarding addresses are there to take care of.

Note that failing nodes would also break the paths of forwarding addresses. That will be ignored here, however, since tolerance of failing nodes is not attempted.

3.3 Object migration

Compared to other parts of the system, actual object migration is a simple matter. Objects are destructed, their contents passed across the network, and then they are reconstructed on the other side. Turning the object’s data into a sequential representation fit for transport across the network is often referred to as marshalling. Reconstruction on the other side is known as unmarshalling. Special care must be taken with data structures containing recursive references. For example, an array could contain an element referencing the array itself. The array must be

passed only once, further references to it are passed as just references.

Native DGD object references are passed as `nil`. It is not possible to reconstruct such references as they have no meaning on another server node. DGD uses `nil` as a null value, given to references to destructed objects among other things. It is different from zero or the empty array in that nil is "nothing" while zero is still a number and the empty array is still an array. Distributed objects should not use native DGD references as they would be lost every time the object migrates to another node.

Code is assumed to be available in a central repository, so no code is passed with the object's data. Just a reference (filename) is passed so an instance of the correct object can be created at the destination.

When an object leaves a node, a forwarding address is created and left behind as discussed in the previous section. The forwarding address is recorded in a table for easy lookup.

When an object arrives at a node, after it has been reconstructed from the network data, it is entered into the node's translation table for object IDs and native DGD object references. Then, if any threads were waiting for this object they are awakened, unless they are still waiting for other objects. This is explained further in section 3.5.

3.4 Attaching and detaching server nodes

When attaching a new node to the system, it connects to a single one of the existing nodes first. That node queries the central daemon to give the new node a unique node ID, and also passes the new node a list of all existing nodes in the system. The new node then opens a communication channel with each of the other nodes.

Removing a node involves a little more work. First the node must rid itself of all objects. As an object is forced off the node, a message is also sent to all nodes telling them to forget any forwarding addresses they may have for the object. It may seem more efficient to give all nodes a new forwarding address pointing to the object's new location through this broadcast, but the broadcast message may arrive at some nodes before the object arrives at its destination. Thus the forwarding address would point to a place where the object will be some time in the future, not a place where it is or has been and the search algorithm would fail.

When room objects leave the node they must be given a new "home node" (see section 3.7.1) to prevent objects from trying to come back. The new home node could be specified when initiating a node removal.

After all objects have left, the node can broadcast a message saying it is shutting down. This prepares the other nodes for a disruption in

communication. Other nodes would otherwise believe that the node has crashed when the connection with it is lost. After this has been done, the node shuts down all communication channels and shuts itself down.

3.5 Execution of threads

As with multitasking in an operating system and multithreading within a single process, when threads execute in parallell in a distributed system, concurrency problems can occur. The problem lies in the threads' need to share access to resources in a reasonable manner without causing inconsistencies.

The basic correctness issues in the design of concurrent programs are safety and liveness. Formal definitions can be found in the paper by Alpern and Schneider [1]. Simple informal definitions will suffice, however. Informally, safety means that nothing bad will happen, while liveness means that something good will happen. Two threads simultaneously accessing the same object may lead to inconsistencies. This is a safety problem. On the other hand, it must be guaranteed that a thread eventually gains access to the objects it needs, which is a liveness issue.

The DOME paper [10] has a nice example of this problem. It is taken from a classic adventure game setting. An adventurer chased by a dragon is attempting to close a door to save himself, while the dragon is attempting to go through the door and eat the adventurer. Assuming a rather sturdy door, it should never happen that the adventurer closes the door just to be eaten by the dragon an instant later. Yet, this is exactly the kind of inconsistencies that may occur. The dragon's code checks the door, seeing it is open. The adventurer's code then closes the door. Thinking it has made sure the door is open, the dragon's code then moves the dragon through the door and lets it eat the adventurer.

The problem is well known from the database world where the results would be rather catastrophic were two transactions allowed to freely access the same data at the same time. Traditional databases use locks to solve this problem. Data locked by one transaction may not be accessed by another before the first transaction has finished with it.

Locks may seem like a quick solution, but they introduce the problem of deadlocks. Deadlocks are especially cumbersome and slow to locate and resolve in a distributed system because it involves the passing of messages across the network as nodes must cooperate to solve the problem.

DOME does things a little differently, using a form of optimistic concurrency control. Each thread works on its own temporary copy of data. When one thread finishes, it is allowed to overwrite the "real" data with its temporary copy. Any threads that had accessed data which now be-

came overwritten are rolled back. Their temporary data copy is discarded and they start executing from the beginning again with a new temporary data set. They must start over because their work was based on outdated data such as the door no longer being open in the example above.

Actually, the type of inconsistency described by the example cannot occur in the distributed DGD system as long as an object is only allowed to exist at one node at a time. Since threads in one DGD server are executed sequentially, only one thread could access the object at a time. In a way the safety problem regarding inconsistencies is already out of the way. The problem is for a thread to get all the objects it needs to execute, fetching them from other nodes if necessary. There are two interesting properties here. The first is that it is generally not possible to know in advance all the objects a thread will access. The second is that a thread cannot stop to wait for anything in any way. It must execute from start to finish without pause.

3.5.1 Remote objects

In DOME's case a half executed thread might discover that another thread has modified data it was dependent on. With DGD the case is that a half executed thread might discover it needs access to an object currently held by another node. While these two cases sound very different, they are similar enough that the same solution lends itself well to both. Though for different reasons, the threads are unable to complete in both cases.

By declaring a function atomic in DGD, any changes caused since execution entered that function can be undone by passing an error from that function. Any effect the function had is rolled back. Only the error message survives. Using an atomic function to run most of a thread, rollbacks like those of DOME can be performed. Placing object identifiers inside the error message tells the code higher up why the thread was rolled back. That code can then perform a search for the missing objects as described in section 3.2. When the objects arrive, the same thread is attempted again by another call to the atomic function.

3.5.2 Locking objects

One further complication arises once a thread has been rolled back and a request for the missing objects has been sent out, however. Once the objects begin arriving at the local node, it will be necessary to decide if they should be held indefinitely on that node awaiting action, or if they should instead be released when other threads request them, accepting

the possibility that the thread which first requested them did not have a chance to execute yet.

Not locking objects on nodes where they are needed will cause them to move more as they will eventually have to move back so the thread can finally execute. This can be inefficient, but in the worst case it can even prevent progress indefinitely. An example would be threads on nodes A and B both needing objects 1 and 2 to run. Object 1 is located on node A, and object 2 on node B. Each node will request the object it does not have. Then both nodes will give up the only object they do have, only to realize the thread still cannot run when the requested object arrives. The objects may bounce back and forth in this manner indefinitely, preventing the threads from ever finishing. This behavior was witnessed in a half finished version of the test implementation.

As mentioned previously, locking introduces deadlocks which are complex and slow to detect and resolve in a distributed system. If both nodes in the example above are unwilling to give away the object they have, neither will ever finish. They will sit and wait for each other forever. Given the time and complexity involved with solving this situation it is better to avoid that it occurs in the first place.

If threads could agree on which of them has a higher priority, threads with lower priority could release their objects when two or more threads of different nodes attempt to access the same object. We must be careful, however, that we do not introduce another liveness problem. Each thread must be guaranteed to finish in finite time. To guarantee this is to guarantee that newer threads do not constantly take priority over older threads, delaying them indefinitely. This can be achieved by creating a total ordering of threads where threads are given higher priority the older they are.

To decide which of two threads are older some sort of timestamp is needed. Absolutely required here is that a thread will eventually become the "oldest" thread if it keeps getting delayed because other threads are taking its objects. Timestamps do not have to be entirely accurate to do this. In fact, Lamport [8] has shown that the physical clocks of distributed computers cannot be accurately synchronized. Even if absolute accuracy is not required, there is a problem when distributed clocks differ significantly. One node should not constantly be issuing older timestamps than another node, giving its threads priority until those of the other node have aged enough.

Logical clocks [8] suit our purposes well. They do not give any node an unfair advantage, and there is no need to alter the system clock. The clocks are simply numbers that are counted upwards. Every time a node needs to issue a new timestamp, it uses one that is one step above the last one it has seen. This ensures that one node won't be constantly issuing older timestamps than the others. However, it can happen that

two nodes issue the same such numbers to two threads and those two nodes must be able to agree on which thread has priority. Including a random value with the timestamp solves this, the random part only being compared if two threads have the same logical timestamp. Should the random values also match, final resolution can be based on node ID.

When remote objects are requested by a thread, the logical timestamp, including its random part, is included with the request. This way the object can build a prioritized queue and move to the nodes where it is needed one by one, visiting the oldest threads first.

3.5.3 Destruction of objects

When a thread destructs an object, the fact that the object no longer exists is registered at that node. This way the node can respond to any future searches for the object. Most of the time this is all that needs to be done.

If multiple threads are attempting to access the object simultaneously and one of them destructs it there may be threads at other nodes waiting for the object at the time when it is destructed. To prevent those threads from locking up forever, their nodes must be told when the object is destructed. As the object holds a prioritized list of threads that have requested it, it is a trivial task to notify them. The other threads will then be allowed to run, and references to the destructed object will evaluate to nil for them. Consistency is not a problem. These threads have already been ordered sequentially by priority. The thread that destructed the object ran before the others. When they run the object no longer exists.

3.6 Persistence

The distributed system must be able to save its state, shut down, and later start up, restoring its state. This ensures that changes to objects do not disappear under normal conditions such as the servers being taken down for maintenance and later brought back online. It is a different question, though, how much should be attempted to prevent loss of data in the case of a crash.

The question is what data it is acceptable to lose if the system should happen to crash. None really, but some data are more important. On one hand, credit card information, and anything related to credit card transactions such as memberships are clearly things one does not want to lose. On the other hand, if a user picks up a worthless pebble in a virtual world, it is probably not a disaster if it disappears in a crash. To make things a little more difficult, though, virtual items that only exist

in online games are being bought and sold by players of the games. This actually creates an exchange rate between virtual currencies and real currencies. Users would probably not be happy to see expensive items disappear.

To be on the safe side, one could treat all threads as transactions and store all their changes in a traditional database system. This would be extremely expensive though. The virtual world wouldn't need to be distributed if it wasn't a large one with many users. The database servers would have to be fast to keep up. Even then, having to store the result of each thread like this would slow everything down.

A better option is probably to store a backup copy of the system state at regular intervals. Important data could be kept in a traditional database or in both places. If a backup is made each hour, at most one hour of changes would be lost in a crash. DGD has a builtin functionality for storing the state of one DGD server to disk, and it does so relatively fast. The problem is that the server nodes can not be synchronized to dump their state at exactly the same time. Besides, there may be data on the network that has been sent but not yet received at the destination — this information must also be stored.

The snapshot algorithm is perfect here. It can be implemented fairly straightforward with DGD; storing a node's internal state as a statedump and storing network state by writing the messages to files directly. The algorithm requires communication channels between nodes, such that each node is reachable by every other node either directly or through other nodes. Each node being directly connected to every other node covers this requirement. Also, each communication channel should have one sender and one receiver. For this reason broadcasts are best implemented by sending multiple copies of a message, one to each node individually, it allows the snapshot algorithm to fit in without any problems.

3.7 Distribution scheme

A difficult question is how to decide which objects go on which server nodes. The division of objects among servers should be done in a way to allow efficiency to be as high as possible. There are mainly two ways in which the location of objects negatively affect performance. The first scenario is when objects accessed by a thread are on different nodes. Rolling the thread back and fetching the objects is a bad performance hit on that thread. A variation of this is when threads on different nodes frequently need the same objects, causing objects to migrate back and forth. The other scenario is having too many objects on the same node, causing that node to become overloaded while other nodes are idle.

Keeping related objects, objects that interact with each other, on the same node reduces the likelihood of having to fetch remote objects for a thread. The problem is knowing which objects are related. Also, some relations are stronger with more interaction than others. At the same time, objects must be spread out among the server nodes to balance the load between them. Just spreading the data equally between the nodes does not necessarily do the job. If one node had a lot of objects, some objects that were not being used would be swapped out to disk, so holding many objects without much activity is not a big problem. Central to the load balancing problem are objects that start threads. The threads take processing power on the node where they run and also increase memory usage by fetching objects into physical memory whether they are objects that were swapped out to disk earlier or remote objects from other nodes. Any object can start timers that set off new threads. It can be hard to predict which objects do this. But the majority of threads are likely started by the objects that accept input from users and take actions to respond to that input. Adding to this, handling input and output to and from users is some work in itself. It would seem that these user objects are central to load balancing the distributed system. While we are not going to attempt any complex active load balancing, objects should be distributed in a way that ensures some level of load balancing. This is necessary to prevent the system from easily breaking down and becoming unusable.

3.7.1 Geographical division

A solution should spread the user objects on different nodes and keep with them the objects that they interact with. How this can be done depends on the type of system in question. For a virtual world, it can be expected that geographically close objects interact more with each other than two objects that are geographically far apart. A user moves from one room or area to the next, objects are picked up, carried objects may be left behind — the majority of actions involve the immediate environment.

Splitting the virtual world up in zones by geographic location and keeping the objects for each zone on its own server node is not a new idea. Several online games have been implemented this way, although the borders between zones are easily visible and a significant delay is incurred by crossing those boundaries. Such unwanted side effects are not a necessary consequence of a geographical division, but there is one major shortcoming to using this scheme which will be reviewed in the section below. Finding a solution to this problem is not easy, but neither is it easy to find a better way to partition a virtual world than by geography.

Lacking any revolutionary solutions a geographical division will be used, including its well–known problem. Objects should be spread among the nodes depending on their geographic placement, but some flexibility for free movement should still be maintained. At least objects must be allowed to move to another node for a short time if a thread on that node wishes to access the object. Some objects are mobile while some are stationary. Objects describing the geography are naturally stationary and they are a good starting point for a geographic division. By giving each such object a "home node", the geographical borders between server nodes have been drawn. Home nodes can be assigned by a developer based on knowledge of the computing power of the different server nodes and expected activity levels in different areas of the virtual world. The idea could be expanded on with a dynamic load balancing algorithm that would change objects' home nodes for a smoother balance between nodes.

There are many possible ways objects could be moved between servers. Emerald allows objects to be grouped together in such a way that they all move together when one of them initiates a migration to another server node. However, it is extremely difficult to foresee object usage patterns, especially in a highly interactive system where the user might move between server nodes and access objects at will. As an example, a user might move between nodes several times before eventually interacting with any carried objects. It is more efficient, then, to move those objects directly from the first to the last node rather than have them follow the user each step of the way.

Following this philosophy, most objects will move freely to wherever threads need them. The exception is user objects. As mentioned above, it is important to spread them out. As a user enters a location, his user object is moved to the "home node" of that location if it is not already there. As the user interacts with the environment, any objects describing that environment are moved to that node because that is where the thread is started; in the user object. Thus there is no need to force any objects to that node except for the user object when it arrives at the given location.

3.7.2 Flash crowds

Geographical division makes sure the objects most likely to interact are on the same node. However, it does not take equally well care of the other end of the problem — ensuring that users are spread out on different server nodes. The problem is, of course, that keeping closely interacting objects together while keeping users apart are two partially contradicting goals. One might design a virtual world and partition it in such a way that activity is expected to be roughly equal on all nodes,

and under normal operation it might be. However, there is always the danger of a special event of some kind drawing a huge number of users into a small geographic area. This phenomenon has been called “flash crowds” and “stampeding hordes” among other things. Suddenly a large portion of the users move to the same server node and the system breaks down. Forcibly moving them to other nodes is not much of a solution, though probably better than letting the system become unusable.

3.7.3 User connections

With mobile user objects handling users' input and output, there is the question of how network connections between servers and users should be maintained and how input and output travels between this connection and the corresponding user object. There are two straightforward ways of doing it. Either connections follow the user objects, or users connect to one server node while input and output has to travel to and from the node holding the user object at any given time except when it happens to be on the same node as the network connection.

Obviously, doing a lot of forwarding of user input and output would increase traffic between nodes. Furthermore, this forwarding is not suited to high–level LPC code where it might have a significant impact on performance. Therefore, user objects will notify users so they can establish a new connection to the correct node when the user enters an area with a different home node. This way connections follow user objects and do not cause unecessary work for the server nodes.

Chapter 4

Implementation

This chapter discusses the implementation. It is over 15 000 lines of LPC code in addition to a small C program. It took some work before it was stable and everything ran smoothly. The implementation was built on top of the kernel library that comes with DGD which provides some basic functionality and a framework to build on. As most of the details are irrelevant to this paper, the focus is on aspects discussed in the other chapters.

4.1 Node interconnection

DGD does not by itself allow making outbound network connections. To make this possible a small program, external to DGD, was written. This program connects to DGD, allows DGD to tell it to connect to other nodes, and accepts inbound connections from other nodes. Traffic going to and from other nodes are multiplexed over the single connection to DGD.

On each node a single object, `icond` (interconnection daemon), handles most of the network logic. This object talks to the external program presented above. Broadcast messages are emulated by sending a copy of the message to each connected node.

The distributed system starts with a single initial node. New nodes are added by letting them connect to an existing node. After connecting, the new node receives a node ID and a list of all other nodes in the system. It can then connect to each of the other nodes. To make sure node IDs are unique, they must come from a central source. This source is an object called the central daemon (centrald). It is actually a distributed object, but it is central in the sense that there is only one of it and it has a well known object identifier. This object is created by the system's initial node at boot time before it assigns itself a node ID.

4.2 Distributed objects

For some objects, such as the `icond` mentioned above, it does not make sense to have distribution properties. All the distributed objects inherit `/lib/distributed` which provides all necessary functionality. When such an object is created it is automatically given a unique ID consisting of the node ID of the node it was created on, and an object ID which is unique on that node.

For an object to move from one node to another, its state must be gathered and converted into a serial format suitable for transfer across the network. This process is known as marshalling. There are functions in `/lib/distributed` that marshal and un-marshal an object before and after network transportation. Since there is no way for the marshalling code to get a list of the object's data members and their contents, all object state must be placed in a special mapping. A mapping is a native datatype in DGD representing an associative array. Marshalling can then be done since this mapping is a known variable.

A prioritized list is kept of threads that have requested the object. Migration is automatically triggered to service the thread with the highest priority. Thread priorities are discussed in section 3.5.2 on page 24. When the thread at the top of the list has completed, the object immediately migrates to service the next in the list. Migration can also be requested by a function call, but it has no effect unless the list of requests is empty.

4.3 Object references

As long as DGD's native `object` datatype is not modified, something else is needed to hold references to distributed objects. A simple array was chosen for this purpose. `OBJREF` was defined as an array of mixed datatypes.

An `OBJREF` has two elements. The first is another two-element array containing the node and object identifiers which together constitute a globally unique object identifier. The second element functions as a kind of cache. It is either `nil` (nothing) or contains a (native to DGD) object reference to the actual object. As an object arrives at a node, any references to it on that node have `nil` in their cache. When an `OBJREF` is dereferenced, the object is found by looking up its node and object identifiers in local tables. A native object reference is then placed in the cache to speed up subsequent uses of the same `OBJREF`. Should the object leave the node, the cached native reference automatically becomes `nil` again. As the object's local copy is destroyed when it moves, all native DGD references to it become `nil` per normal DGD behavior for

destructed objects.

Unlike with C++ it is generally not possible to redefine operator behavior in DGD. However, the dereferencing operator (->) is equivalent to calling the function `call_other()`. Overriding `call_other()` made it possible to use an `OBJREF` in most ways just like a native object reference. The regular `o->f()` invokes a function, whether `o` is an `OBJREF` or a native `object`. However, seeing whether two `OBJREF`s reference the same object must be done by calling `same(a,b)`, a function that had to be written for that purpose.

4.4 Locating objects

Each node holds a lookup table indexed by object identifiers. The table holds the node's knowledge about each object. It may know nothing about an object, in which case it has no entry. The object may be present at that node, the table holding a native DGD reference to it. It may know that the object has been destructed. Finally, the table may hold a forwarding address for the object, with a timestamp. In this case it is the latest forwarding address for the object seen by this node.

Objects are located using the algorithm described in section 3.2 on page 16. The implementation of the protocol logic and routing of messages is straightforward and won't be discussed here.

4.5 Object migration

Recursive data structures are possible in DGD and that is taken into consideration when marshalling (serializing) the data. Callouts (delayed function calls) are also included with an object's state. After marshalling the object state, the source node destructs the object and sends its state across the network to the destination node. The object's type is also included in the message.

When the state is received at the destination node, that node creates an object of the correct type. The state is restored (un-marshalled) from the network message and the object is then complete.

The repeated destruction and recreation of objects presents a problem. Like in many other object oriented systems, DGD objects have a creator function that is called automatically when they are created. It is normally named `create()`. However, `create()` is called each time an object has migrated, and before its state has been restored. A creator function may have unwanted side effects that are not undone by overwriting the object state. Therefore, a new creator function was introduced, named `new()`, which is only called once at the creation of a

new distributed object. The `create()` function in `/lib/distributed` was given logic to call `new()` only at the appropriate time.

4.6 Execution of threads

The majority of threads are expected to run to completion without problems and in that case they are not even given a priority. Still, all threads go through two special functions before they do anything else. These functions make sure the rest of the code is executed atomically and if necessary rolled back and given a priority for later execution. Figure 4.1 on page 35 should make the following discussion easier to follow.

The first function calls the second and catches the error thrown by it, if any. If there was an error, it contains the IDs of the objects the thread needs that were not present. The thread is given a priority, the necessary information stored about it, and the missing objects are requested across the network. As those objects arrive from other nodes, they are taken off the thread's list of missing objects. When the list is empty, another attempt at running the thread is made.

The second function is atomic. Meaning, that if an error is thrown from it, all changes made since execution of that function started are undone. The only noticable changes are that ticks (processing resources) have been spent and there is an error in the form of a string that can be caught by code somewhere up the call chain. A specially formatted error string is used to communicate which objects the thread needs. The `error()` function is overridden to prevent normal code from generating errors of this reserved format.

The atomic function calls the function that starts the thread's actual work. If it invokes an object that is not present at the node in question, an error is thrown containing that object's ID. This is not a complete solution, however. Any code between the atomic function and the point in the call chain where the error is thrown is able to intercept the error with a `catch()`. Therefore the error condition is also recorded in an object along with the ID of the missing object. If execution finishes without an error, the atomic function then checks this special object for an error condition. If there is one, the error string is regenerated and thrown again. As it passes up from the atomic function all changes are reversed.

There are two main problems with the implementation described above. The first is that an object is not determined to be needed and missing until it is dereferenced. If the thread needs to access 100 objects that are on another node, it will be rolled back 100 times. The `require()` function is intended to remedy this problem. Passing a list of objects to this function makes it possible for code to declare some-

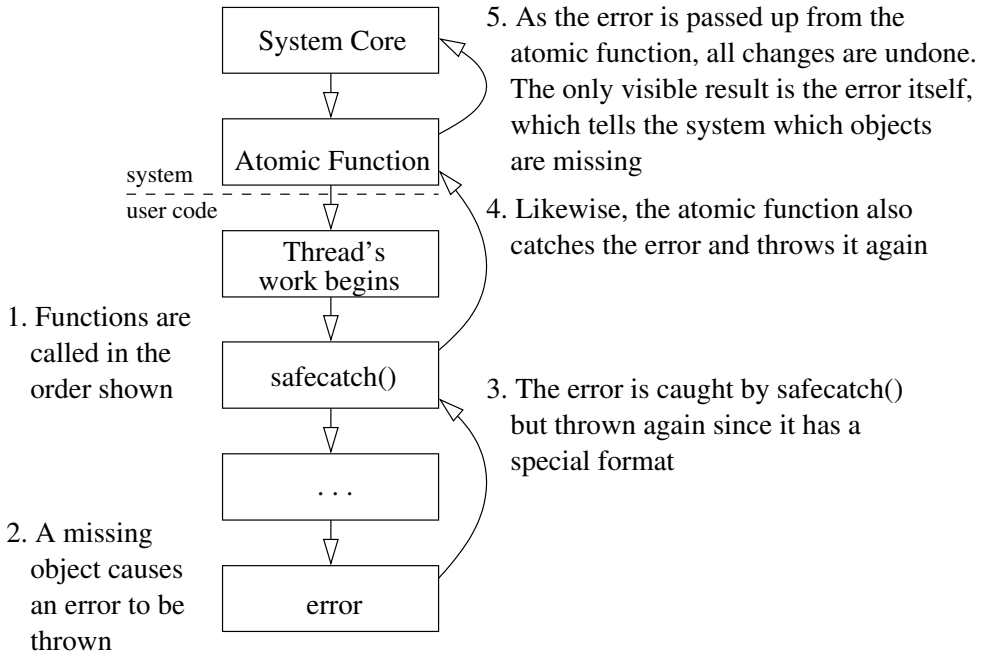

Figure 4.1: A callchain example with error propagation

what in advance which objects it is going to access. This can cut down dramatically on the number of rollbacks, increasing performance, but it places more responsibility upon the programmer. It seems difficult to find a more automated solution, however.

The other problem is that the special errors can be caught by normal code before they reach the atomic function. This is no threat to consistency as the atomic function will still find the error in a central object as described above. Still, the thread continues to execute, wasting processing resources past a point where it is known that all its work will be rolled back. This hurts performance and `safecatch()` was added to let programmers catch errors without having to worry about this problem. This function catches an error, but throws it again if it has the special format, so that the function appears to only catch regular errors. Since DGD's `catch` is a language construct rather than a function, a different name (safecatch) had to be chosen. This unfortunately means the programmer must remember to catch errors the right way, or performance still suffers.

Figure 4.1 shows an example. The boxes symbolize functions, while the downward arrows are function calls and the upward arrows show how the error propagates up through the callchain. At step 2, when one or more objects are found to be missing, that information is stored in a

special object before the error is thrown. If there had been a `catch()` instead of a `safecatch()` the error could have been prevented from propagating further up the callchain. But the atomic function can still find the error condition safely stored in the special object. This way the atomic function can always determine if the thread was missing any objects or not. The user code cannot prevent this, only delay it.

The atomic function catches all errors and only throws an error itself if there were missing objects. Any other error means the thread was able to finish as far as the distributed system is concerned. When an error is passed up from an atomic function, DGD rolls back any changes made since that function started executing. The "System Core" in the figure catches and deciphers the error so that it can attempt to run the thread again when the missing objects have been fetched.

4.7 Persistence

The distributed snapshot algorithm described in section 2.4 on page 12 is used for persistence and the implementation is fairly straightforward. Snapshots are initiated by the central daemon, which does so automatically at one hour intervals. It is also possible to initiate snapshots manually. Node states are recorded using DGD's builtin statedump feature. Network states are recorded by simply writing the relevant messages, if there are any, to a file. A complete distributed snapshot consists of one statedump file, and possibly one network file, per node.

The snapshots are numbered, and the marker messages carry these numbers. This, in conjunction with node IDs, provides an easy way to give the network state files unique filenames.

When the system is starting from a snapshot, the nodes must first be connected, one by one. Once the last node has been connected to the others, all the nodes will see that all the other nodes are again present. They then each read their network state file. By having an object remember the number of the last snapshot performed, the correct network state file can be picked. The messages read from the file are then "played back" as if they were coming from the network. Afterwards, the system is ready for normal operation.

A system that gathers up all the files belonging to a snapshot, storing them away in an archive, would probably be a good idea for real use. However, it was not necessary for testing.

4.8 Test environment

On top of the system that supports distribution, a few simple things were implemented to facilitate testing. User objects allow users to log

in, holding a username and password among other things. Users also have bodies as their "physical" representation. Other "physical" objects are rooms and simple objects for users to manipulate. The command parser understands a few commands with a very simple sentence structure. There is one object for each command that performs the actual work. The parser and command objects are not distributed objects, instead one copy of each object exists on every server node.

A few rooms were set up, creating a very small test world. To implement the distribution scheme introduced in section 3.7 on page 27, each room was given a home node. When a user moves into a room whose home node is not the current node, the user is forced to connect to the room's home node. This keeps users on the home node of the room they are in. Since users start threads when they enter commands, it tends to keep the rooms and their contents on that node too as those objects are often invoked by commands.

Chapter 5

Results and Conclusions

This chapter discusses results and observations from the test implementation and conclusions based upon them. At first it is necessary to describe the test implementation that most of the results were derived from.

5.1 The test implementation

An implementation was written as described in previous chapters. After a few initial tests and observations, which will be discussed shortly, it seemed interesting to measure a few statistics while the system was used. These statistics might give an indication as to how well the theory discussed in previous chapters work in practice. Events were counted to see how often broadcasts were performed, how often threads had to be rolled back and how much processing power was wasted due to rollbacks.

There are many ways to run such a test. The best way is obviously to do the measurements in an existing system running a large persistent world on a dozen servers with at least several thousand users. The persistent world would be designed for a particular purpose; the same purpose the users are there for. That purpose may be entertainment, business, a virtual university or something else. Even in this case, the statistics would only apply directly for that type of system. A game world and a virtual university would probably yield completely different results because the users' behavioral patterns would differ greatly. In the game's case, users (players) might move all around the persistent world to gather some kind of points. The university might see more local activity in the form of lectures and discussions. Still, teaching material and data for collaborative projects might have to travel between server nodes in an unpredictable fashion. The bottom line is that user behavior, which is strongly affected by system type and design, will greatly af-

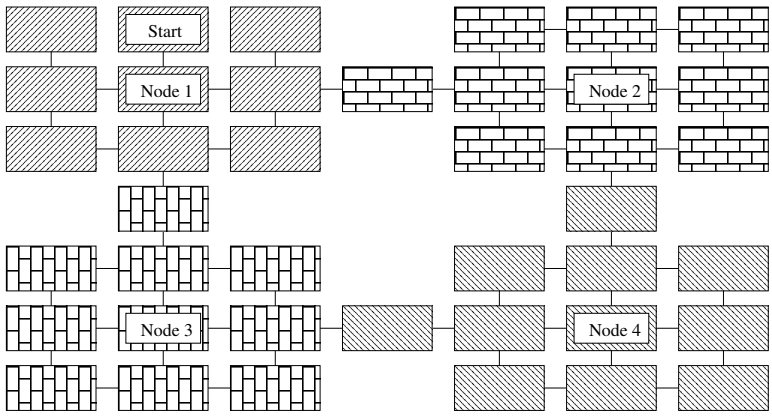

Figure 5.1: A simple room-based test implementation

fect how and how much nodes interact. This again influences the aforementioned statistics. Obviously, then, good results in one scenario do not imply good results in another. Still, it seems reasonable to believe they could give a good indication as long as differences in expected user behavior are considered.

Unfortunately, a full test of any of the scenarios mentioned above was out of reach for this relatively simple project. It was possible to have a high number of computer controlled "users", which could not possibly have actual human behavior. The other option was to invite a few people to try the system. This would at least give human behavior, although it would be influenced by their reason for being there. There was no time to construct an interesting system with a real purpose. So users would be coming for the rather synthetic reason of testing the system, most likely its distributed features in particular.

Despite the limitations, the latter option was chosen and it is hoped that the results can still give an indication of how some aspects of the system might behave in a real world situation. A small room-based world was created and a message posted in a few public forums on the Internet, inviting people to try the system. 25 people did and statistical measurements were recorded for about a month. The system was entirely text based. Users had commands to move between rooms, take items, drop them, list carried items, give items to another user, talk to everyone in the same room and a command to send a message to another user anywhere in the persistent world.

The persistent world was created as a handful of rooms, divided between four server nodes. Needless to say, the small amount of rooms and users took no noticable amount of memory or processing power from the servers the system ran on. Figure 5.1 shows a map of the world used.

Each box on the map is a room. Users would appear in the room marked "start" when they logged in for the first time. Subsequent logins would have them appear at the room where they last logged out. Each room on the map is filled with a pattern. Rooms with the same home node have the same pattern and are normally located on the same server node. Users move along the lines connecting the rooms on the map. The start room was described as a cabin, while all the other rooms had outdoor descriptions with a different theme for each node. An item or two were placed in most rooms, allowing users to pick them up and carry them, perhaps to another node.

5.2 Programming

Making distribution completely transparent to the programmer was not attempted. Still, some degree of hiding the distribution mechanisms was done. Ideally, the system should be easy to program for someone already proficient with DGD programming. While complete transparency may be undesirable, the programmer should not be forced uneccessarily to worry about issues of distribution support mechanisms. While transparency was a minor concern, one of the most important goals was to stay as close as possible to traditional DGD programming; its single-threaded semantics in particular. The rest of this section will show to what degree this was accomplished.

When programming, certain rules must be followed:

1. All distributed objects must inherit `/lib/distributed` directly or indirectly.

This rule is actually a useful one. It allows the programmer to choose whether or not to give an object distributed properties. Some objects work better when not being part of the distributed class of objects. Objects that hold only code, or whose state is only relevant to one node, may not need to be able to migrate. And without that ability, they are always guaranteed to be available on the node in question. Examples may be code for user commands, and objects that handle various local node management.

2. All distributed objects must be clonable, and the master object may not be used (since it cannot migrate).

When a file is compiled, a master object is created for it automatically. To have more objects of the same type, "clones" can be created. Master objects should not be destructed at random as recompiling the code to get a new master object would introduce a new version separate

from any existing cloned objects. However, it is necessary to destruct the local representation of a distributed object when it migrates. This is because it is relied upon that direct references (of the built-in DGD kind) to an object become nil when it migrates. Destructing an object is what makes all references to it nil. For this reason, the master objects of distributed objects should not be used in any way. This is not a big problem, however, as it is a common practice not to make use of the master object when clones are made.

3. Distributed objects may not create nor subscribe to events using the kernel library's event system. The kernel library is a library of LPC code that comes with DGD which gives a good foundation for further development.

This rule is not really a problem. If an event system is needed, one can be written that is compatible with distributed objects. It could replace or run alongside the event system included with DGD's kernel library.

4. A distributed object's creator function is named `new()`. This function is called only once right after the object is created. It is normally named `create()` when programming DGD with its kernel library.

Two separate creator functions are needed. This is because the regular one is called every time an object is created to hold the state of a migrating object received from the network. By modifying the kernel library the names (create and new) could be switched, making the issue of creator functions transparent.

5. Variables referencing distributed objects should be of type `OBJREF`. To see if two `OBJREF` variables reference the same object, `same(a,b)` must be used instead of `a == b`. An object reference of DGD's builtin `object` datatype is only guaranteed to be valid until the end of the current thread if it references a distributed object.

Regular object references in DGD cannot hold globally unique identifiers, so a different type of reference is obviously needed. `OBJREF` is actually implemented as an array, but DGD makes it easy to modify the dereferencing operator (`->`), so dereferencing can be done the normal way.

6. Data members of a distributed object must be declared as part of a special mapping (associative array) rather than regular variables. Otherwise they lose their state when the object migrates.

Since there is no way for LPC code to get a list of an object's data members and their contents, all state must be placed within a known mapping so that the state can be collected and sent to another node. This annoyance could have been avoided by modifying DGD.

To ensure good performance, certain guidelines should also be followed:

1. To catch errors (exceptions) the `safecatch()` function should be used rather than DGD's regular `catch()` construct.

The need for `safecatch()` comes from the specially formatted errors that are intended to abort a thread's execution so that it may be rolled back as quickly as possible. Should code within the thread catch this error and continue execution, it will take longer before the thread can be rolled back. The thread is still found to have failed and eventually rolled back, so this has no bearing on the system's integrity. It hurts performance as it makes the rollbacks more costly by continuing execution wastefully at a point where it is known that the thread will be rolled back. Had distribution been implemented within DGD itself, this issue could have been completely avoided.

2. If at a point in the code there is a collection of objects that subsequent execution is known to access, then the underlying system should be notified by a call to `require()`.

Even if it had been possible, full transparency would probably not be a good thing. During the construction of the test implementation, the `inventory` command was first given a straightforward implementation. What the command does is to take the list of objects carried by the user, collect a textual description from each of them in turn, and finally display a list of those descriptions to the user. The result was horrible performance when a user executed the command after crossing over from one node to another. As the command accessed the items one by one, the thread would be rolled back every time. If a user carrying a hundred objects would move to another server and type `inventory`, the thread executing his command would be rolled back 100 times. Not only do the rollbacks themselves waste resources, requesting one object at a time rather than all at once can result in a significant delay before the thread is finally able to complete. Abiding by guideline number two above, passing the list of objects to the `require()` function before accessing any of them, easily fixes the problem. In this case the system will request all the necessary objects in one go.

The last example also shows that in a real system it would be extremely useful for programmers to be able to see how many rollbacks are caused by their code. Such a mechanism could be used as a profiler,

to locate performance bottlenecks. Not only are programmers bound to forget the guidelines at times, it may also be difficult to see everything that could cause such “cascading rollbacks”.

While none of this is in any way complex and the details of the underlying mechanisms have been hidden, a shorter list would be desirable. This is one of the downsides to implementing distribution on top of DGD using LPC rather than within DGD itself. While it is more flexible, as everything can easily be modified and extended, even while the system is running, it is impossible to do certain things transparently. Most of the rules are merely annoyances and could have been avoided had DGD been modified. The other downside is that the LPC code will not be as fast as DGD's code which is written in C. However, as the layer of code supporting distribution stays out of the way most of the time, the difference may not amount to much in practice.

Even if it may seem like much at first, the difference from regular DGD programming is not that great. Code is still written the same way, with a few relatively minor changes. Even though servers are working concurrently, there are no concurrency problems to worry about for the programmer. To any given thread, all other executed code appears to have run either before it or after it. Single-threaded semantics have been preserved. This is taken care of by the way threads are rolled back and finally executed in entirety within one node, with exclusive access to relevant objects.

The rules are simple and they are relatively few. Perhaps the greatest danger lies in that breaking rules five and six will not break the code until objects migrate. The result could be bugs that are hard to track down and code that may work well for a long time before it suddenly breaks for no apparent reason. Likewise, the guidelines may be easy to ignore until performance really suffers. It would be safer, and easier, if breaking any of the rules would result in immediate consequences. That may be difficult to accomplish without modifying DGD itself, however.

5.3 End user experience

From the user's point of view, the test implementation appeared for the most part like other similar systems. One of the major objectives was to keep the system quick and responsive as it is intended for interactive use. To a great extent this was achieved except when crossing node boundaries.

No delay is observed when sending a message to a user on another node, but there is clearly a noticable delay when going from one node to another. Both operations involve roughly the same number of object migrations. This implies that the delay experienced when moving across

server nodes is mostly due to having to establish a new connection to another node and log in. Not only must the user be authenticated again, but it can take a significant amount of time to establish another network connection between the client and one of the server nodes. Clearly, the way users are moved from one node to another was not well enough thought through. It does save the server nodes the burden of having to forward data between users and other nodes. Sadly, it also introduces a delay large enough to ruin the illusion of one large persistent world rather than pieces run on different servers.

This could be solved with a "connection manager" which is always connected to the user and to all server nodes. This solves both problems. The connection manager can forward data much more efficiently than the server nodes whose code runs in DGD's virtual machine. Meanwhile, the connection to the user stays up when the user moves between nodes. By keeping connections to all server nodes, no connections need to be torn down or built up between the connection manager and server nodes either. This solution should give the highest possible responsiveness at all times while avoiding unnecessary waste of server resources. Connection managers can also do some work to offload the main servers, such as encryption, when that is wanted, and possibly some pre-processing of user input before it goes to the main servers.

5.4 Thread scheduling

Another requirement with keeping the system responsive is to make sure that all threads will at least eventually get the chance to run. This is not enough to make the system truly responsive, however. After a thread is created, it should be able to complete relatively soon. This requires a fair scheduling of threads. Scheduling of threads here refers to how threads are scheduled in the system as a whole, not on each individual node. A reasonable definition of fair scheduling would be that threads be allowed to finish in the order they are created. Such a strict level of fairness does not seem necessary however, nor easy to do without a costly overhead due to the needed synchronization. It is with this philosophy that the thread priority system was designed, as described in previous chapters.

A simple test was performed to see how the scheduling worked in practice. Two users on separate server nodes would send 100 commands to the system with no delay in-between. The commands sent short textual messages to the other user, on the other node. The same test was also done with three users on three different nodes, with the same results. In the case with three users, they would send messages in a circle, from user A to B, B to C and finally C to A.

To execute the command in question, both the user's own "user object" and that of the message's recipient, is needed. Textual messages for a user are delivered by passing them to that user's user object. These threads are competing for the same objects, therefore they must be scheduled in a strict order, they cannot execute in parallel.

In the beginning, one user would send off several messages without receiving any, because the users did not start at exactly the same time. After that each user would send two messages at a time before letting the next user send messages. In this way they would continue taking turns, executing two commands each, until they had finished their 100 commands.

The priority-based thread system clearly works as intended. No thread can be blocked by other threads for long. It was observed that each node was able to run two threads before the necessary objects were lost to a remote thread with higher priority. This is fine as a strictly fair one by one scheduling was not a goal. In fact it may be interesting to go in the other direction. If each node was able to execute a few more threads each time before losing the necessary objects to another node, it would reduce object migration overhead. This may be difficult to do without unwanted side effects, such as decreased responsiveness.

The case used for testing shows that sending a message from one user to another can be rather inefficient. The recipient's user object migrates to the sender's node to receive the message, then goes back to the first node to actually send the message to the user. Instead, the message could have been passed between the nodes directly and immediately passed on to the user. However, the way it is done guarantees that all users see the same events in the same order. This is important for consistency. However, there may be applications where it makes sense to relax the demand for consistency. The widely used chat system Internet Relay Chat (IRC) works this way. When using IRC, two users do not necessarily see the same messages in the same order. A persistent world with a communication system that allows many users from different nodes to communicate may have a high number of rollbacks unless a different method for passing messages can be employed. As a more general tool, a simple remote procedure call (RPC) mechanism may be useful, allowing a thread on one node to start a thread on another node. If any results from the call were to be returned, it should be done through an RPC call in the opposite direction, since a thread cannot wait for a reply.

The execution of the commands used for testing thread scheduling are not part of the statistics found in the following sections.

5.5 Measured results

Tables 5.1, 5.2 and 5.3 hold the test implementation's recorded statistical numbers as well as calculated numbers derived from those statistics. Again it should be emphasized that the numbers were recorded in a persistent world only 40 rooms large, used by 25 users for a short period of time for no particular reason other than to try the system. It is not possible to draw any substantial conclusions from this material. Yet, it may be sufficient for a few interesting observations. Improving this material is grist for future work, which will be discussed in the following chapter. A discussion of the statistical material follows.

5.5.1 Searches

Table 5.1 shows the statistics related to searching. Regular searches here mean searches using forwarding addresses.

Searches	**Node1**	**Node2**	**Node3**	**Node4**	**Total**
Broadcast searches	29	97	102	107	335
Regular searches	603	318	338	260	1519
Regular per broadcast	20.8	3.3	3.3	2.4	4.5

Table 5.1: Search Statistics

Nodes 2, 3 and 4 are using broadcasts to locate objects much more often than node 1. This is probably because node 1 is where all users begin. Node 1 therefore has the advantage of knowing all the user objects from the beginning whereas the other nodes do not. Perhaps the users also picked up more objects early than they did later in their travel across the nodes. Node 4 is the one using broadcasts the most with only 2.4 regular searches per broadcast. It is also the node furthest away from the beginning. Of course collecting objects from the other nodes and bringing them back to node 1 also causes broadcasts from node 1 when the objects are first accessed there.

A few broadcasts may also be the result of a user on one node sending a message to a user on another node if the recipient has never visited the first node and left a forwarding address. This is an example of an object (the user object) being able to address another object without having a direct object reference to it. A broadcast capability is very convenient for such cases as there is no way of guaranteeing the presence of a forwarding address. Though in this case, a central authority, which object ID is likely known, must be queried to translate the recipient's user name into an object ID. A forwarding address could also be acquired through that central authority.

Had the system been running for a longer period of time without introducing new objects, the regular per broadcast ratio for searches would increase greatly. This would happen as nodes acquire forwarding addresses for more objects, making broadcasts less often necessary. Still, not creating new objects dynamically is unrealistic for most persistent world scenarios.

The ratio of regular searches to broadcast searches is not promising. However, they could be reduced drastically by including a forwarding address with every object reference passed from one node to another. This is what Emerald [7, 6] does. As the implementation used rather anonymous arrays to represent object references, this was not easy to do as it made it difficult for the system to recognize a reference. It could perhaps have been done by including some form of signature in every object reference. This should work even if it is not a clean nor entirely robust solution. It should be mentioned that, after the implementation was written, DGD got a new feature called "light weight objects". Using light weight objects as references and passing forwarding addresses along with references could be done in an elegant and efficient manner.

If forwarding addresses would be passed along with every object reference in a migrating object, broadcasts would only occur when accessing a reference that was received by a node in some other way. One example of this is the central daemon object whose ID is known to all nodes. It results in one broadcast from each node except the one where it was created. Another way a node might receive an object reference is if a programmer typed in an object ID manually, perhaps during debugging. All in all, these are special cases and result in few broadcasts. The proposed change would almost remove the need for broadcasts. It would improve performance in systems with many server nodes that would otherwise have to process many broadcast messages.

5.5.2 Threads

Statistics were also recorded to see how often threads were rolled back. These numbers can be found in table 5.2.

Threads	**Node1**	**Node2**	**Node3**	**Node4**	**Total**
Threads executed	13568	2049	2095	1808	19520
Rollbacks	567	387	425	320	1699
Threads per rollback	23.9	5.3	4.9	5.7	11.5

Table 5.2: Thread Statistics

One could say that, systemwide, for every 11.5 threads, a thread is rolled back once. This is not accurate, however, as some threads are

rolled back more than once. It would have been interesting to measure what share of threads is rolled back once, what share is rolled back twice, and so on. Such numbers were unfortunately not tracked by the implementation. They may be even more interesting in a production system because the result can be improved by code changes. By requesting more objects at once rather than one by one, where possible, the number of cascading rollbacks are reduced. As discussed in section 5.2, this fact is easy to ignore, so giving the programmer some feedback of this form would be helpful.

Most, if not all, of these threads were initiated by user input – one command corresponding to one thread. Roughly 70% of the threads were executed on node 1 meaning most user activity took place there. A similar tendency can probably be observed in most persistent worlds; users flock to certain places or activities that are more popular than others. In this case users would talk to eachother in the cabin that was the starting point. The rest of the world was rather barren. One could walk around and pick up inanimate objects. Enough to try the distributed properties of the system, but that was it. Users also tended to return to the cabin, after having visited the other nodes.

Compared to the number of threads it executed, node 1 also had much fewer rollbacks than the other nodes. This fact adds to the image of node 1 being the one that saw the most local activity while other nodes had people enter, take a few objects, and move on. Numbers like these would hopefully not be seen in a real system. However, it shows how important it is to spread out popular places or activities on different servers. Otherwise the workload quickly becomes badly skewed between server nodes.

The amount of rollbacks seems disappointingly high. Considering that talking would not have caused rollbacks, other activity must have created a very high number of rollbacks. At first sight, this does not look good. The setting would have influenced these numbers greatly, though. Users did things that cause rollbacks because they wanted to see how the system works. Also, the world is so small that users cannot move far before they cross a server boundary. Finally, there was nothing to motivate activity local to one server node, except perhaps for the fact that everyone started out at the same location. A more realistic scenario would be completely different. Therefore one could probably expect the "threads per rollback" statistics seen here to improve greatly.

5.5.3 Efficiency

Perhaps one of the most interesting things to measure was how much processing power was wasted on rollbacks. DGD has its own unit for measuring amounts of work performed in its virtual machine, called

"ticks". DGD can count ticks while code is running, so this property was easy to measure. Table 5.3 shows the recorded numbers. "Completion ticks" are the ticks spent when threads completed successfully, running from start to finish, shown in thousands (k) of ticks. The number is the total for all threads that were executed on that node. "Rollback ticks" are ticks lost to rolling back threads, also shown in thousands. "Ticks/Completion" is the average number of ticks spent on a complete thread. "Ticks/Rollback" is the average number of ticks lost in a rollback. "Work wasted" is the share of rolled back ticks out of the total number of ticks (completion ticks + rollback ticks). It should be mentioned that only the threads themselves were counted. There is some additional overhead in storing information about a thread after it has been rolled back and starting the thread again later. However, this overhead should not be large enough to make a big difference.

Workload	**Node1**	**Node2**	**Node3**	**Node4**	**Total**
Completion ticks	93045k	17314k	16390k	13921k	140670k
Rollback ticks	1988k	1437k	1535k	1078k	6038k
Ticks/completion	6858	8450	7823	7700	7206
Ticks/rollback	3506	3713	3612	3369	3554
Work wasted	2.1%	7.7%	8.6%	7.2%	4.1%

Table 5.3: Efficiency–related Statistics

The amount of work wasted is below 10% on all nodes. On average, 4.1% of all the work done is wasted on rollbacks. This is subject to opinion, but these numbers are probably acceptable for many uses. As mentioned in the previous system, a realistic scenario would have much fewer rollbacks. As the size of the persistent world in each node grows there is likely more activity completely local to each node as opposed to activity that needs access to remote objects. Fewer rollbacks of course lead to less wasted work. The effect is clearly visible here. Node 1 wasted 2.1% of its work on rollbacks, which is much less than the other nodes. Node 1 was also the one were users talked, which is a local activity. Other nodes saw users often entering from other nodes and soon leaving again. Those activities cause rollbacks. A larger more realistic system would likely see much lower shares of wasted work as long as it does not promote constant travel all around its world. Below 1% seems likely. Of course this is highly dependent on the specifics of the persistent world and the actual usage patterns exhibited by its users.

The table shows that the average number of ticks wasted on a rollback is roughly half the number of ticks spent on an average complete

thread ("Ticks/Rollback" is roughly half of "Ticks/Completion"). This could probably occur in a real system as well. It shows the importance of keeping the number of rollbacks to a minimum. If every thread was rolled back once, half of the server's processing power would be wasted executing code and then removing its effects directly afterwards. Placing related objects on the same node is extremely important for high efficiency. If this is done correctly in a real system, less waste than shown here is likely to be experienced. That is because a real system would not have such small areas that lead to frequent crossing of node boundaries.

What this shows is that the wasteful effect of executing the same code more than once due to rollbacks can be negligible. Assuming correct basic design of the persistent world and an intelligent distribution of its objects between the server nodes, this waste should be minimal. However, that is not the full picture. There are of course costs associated with having DGD execute atomic code. While the actual rollbacks are most expensive, there is always a cost due to having to back up data modified by atomic code in case a rollback should be necessary. Normally this overhead is low, but it can be high in some cases. Since this is highly dependent on the code being executed, it is hard to measure in a generally useful fashion.

It is difficult to provide exact numbers for the overhead involved with rollbacks. The overhead will vary from one system to another. The general indication seems to be that the overhead is most likely acceptable in most cases. Still, it is highly system dependent. It is especially important to keep the number of rollbacks down by grouping objects that interact with each other on the same node.

Finally table 5.3 shows that node 1 had to carry about two thirds of the total workload, leaving a very light load on the other nodes in comparison. Real load balancing was not among the objectives. It does however make an interesting case for further work.

5.6 Summary

In some ways, more transparency for the programmer could have been wished for. Most of those changes would have to be applied to DGD itself, though. On the other hand the system may get too transparent at one point, making a serious performance penalty easy to overlook. For actual use, some kind of feedback needs to be implemented. A programmer or administrator will need to know how well a piece of code performs in accessing remote objects. At least there should be a warning when badly behaved code triggers a large number of rollbacks.

To the end user the distributed properties of the system are mostly

non-obtrusive, keeping commands responsive. The exception is when crossing server node boundaries. The delay observed by a user when moving to another node is more than noticeable, but could be reduced by the use of connection managers. Thread scheduling works as intended, making sure no thread is kept from executing for long.

The system works as intended, for the most part. Assuming the introduction of connection managers, users would likely not be able to tell that the server is actually a set of separate server nodes. While there are some issues a programmer needs to be aware of, the worst complexities of distributed programming are kept out of the way by giving the programmer an environment that still maintains single-threaded semantics.

Efficiency seems good as far as this limited test can tell. The impact of rolling back threads is small. The number of broadcast messages seems excessive, however. This could easily be rectified by including a forwarding address when an object reference moves from one node to another.

Chapter 6

Future Work

There are issues not covered in this paper that would make interesting future work. This chapter looks into three major areas: fault tolerance, load balancing and properties specific to graphical systems. Additionally, it would be very interesting to see how the methods described in this paper actually perform in a realistic large scale system. The importance of realistic testing has been extensively discussed in the previous chapter, so it will not be repeated here.

The following sections discuss fault tolerance and load balancing; these are issues directly related to the two most important reasons for using a distributed server. One reason is that redundancy makes fault tolerance possible, yielding a more stable service. The other reason is sharing the workload between several computers, making a much larger system possible. Finally, graphical systems are discussed, as they present slightly different requirements than text-based systems.

6.1 Fault tolerance

The different types of persistent worlds discussed in this paper have many things in common. One of them is that downtime is highly unwanted. Ideally, some of these systems could run for years uninterrupted. However, hardware or software faults are bound to occur at some point. One of the advantages of a distributed system is that there is redundancy in hardware. If this is taken advantage of by redundancy in software, the crash of one server node does not have to imply loss of vital data, nor the service going offline.

One of the weaknesses of the system presented in this paper is its inability to survive the failure of even a single server node. With two or three nodes this may not seem like a serious shortcoming. But the picture changes if one imagines a system using a hundred nodes. Each additional node is an additional source of failure and adds to the prob-

ability of a crash. It also seems wrong that one failing node should be allowed to bring down 99 other nodes that were functioning just fine. With redundancy it should be possible to avoid loss of data and keep the system running.

There are many levels of fault tolerance. At one extreme one can imagine a fault that allows a node to stay operational, but with erratic behavior. It may not be realistic to cover every type of failure and it might impose too much overhead to do so. The most fruitful approach is probably to investigate the most common type of failure: the complete halt of a node. This may happen because a hardware or software problem has disabled the computer completely or crashed just the DGD process.

Network faults may also be interesting to consider. A single node could become isolated. The network could become segmented, splitting the nodes in two groups that can communicate within the group but not with any node in the other group. Such network errors are indistinguishable from complete node failures. Inability to communicate with a node could mean either type of failure. Perhaps something better can be done than what the implementation described in this paper does, which is a complete system shutdown. Network faults may not be very common with this type of persistent world server however, as all the nodes are likely to be on the same local area network (LAN).

By keeping a replica of every object, one can prevent data loss when a node goes down. That is assuming the replica and original are not both present at the crashing node and that they are not lost while being sent across the network. If such an approach is taken, interesting research lies in finding ways to keep replicas updated and, perhaps more difficult, to keep replicas synchronized in such a way that the recovery from a failure does not introduce inconsistencies. These things seem very difficult to do without seriously degrading efficiency and responsiveness.

6.2 Load balancing

If one goes to all the trouble of creating a distributed server but ends up having one node doing all the work, the whole thing seems a little pointless. This paper has discussed a crude way to share work between nodes. However, as witnessed by the statistics presented in the previous chapter, the resulting workload was in no way balanced.

It is possible to achieve some level of load balancing with the implementation described in this paper in the way that an administrator can change rooms' home nodes. This is cumbersome, however, and it is extremely difficult to adapt to sudden changes in the workload. The

task of load balancing is well suited for automation. There is one exception, though, as the computer must look at past or current behavior as a means for adapting for the future. The administrator on the other hand may know of future events that will place a burden on the system in a particular way. A proper division between automation and human control should be found.

6.2.1 Automatic load balancing

There are surely many possible solutions for an automatic load balancing system. One approach would be to keep the geographically based distribution scheme and the concept of "home nodes". The load balancing system would then change the home node of objects to even out the load between nodes. Obviously, nodes with a high load need to drop some objects while nodes with a low load can take more. However, if objects are jumbled around thoughtlessly, performance will suffer badly. As discussed in the previous chapter, the number of rollbacks must be kept low. That means keeping the frequency of remote object accesses down.

The problem is that two goals need to be achieved. The load must be balanced while efficiency must be kept high. Objects must move, but objects that access each other frequently must be kept together. One can imagine all the objects in the system interconnected in a large web, where the connections symbolize objects that access each other. The boundaries between server nodes will place great costs on those connections in the web that they cross. The optimal solution would be to place those boundaries across as few connections as possible, choosing the ones least frequently used. Finding ways to dynamically shift the node boundaries without destroying performance is certainly a challenge. Finally, moving areas between server nodes is not so interesting in itself, it is moving areas with high activity that helps the most to even the load.

There are also many tuning issues. For instance, how fast the system should attempt to adapt to changes in the workload. Quickly adjusting to temporary changes makes little sense as it causes the system to spend resources on load balancing with little gain. Adjusting too slowly to more permanent changes may cause unecessary periods with overloaded server nodes. Load balancing may need to be tuned for each individual system. Perhaps even this tuning can be automated.

6.2.2 Distribution scheme

It would also be interesting to investigate completely different distribution schemes. For a persistent world, a geographical distribution is

likely a very efficient one, as geographically close objects are likely to be accessed together. This results in few object accesses across server nodes, so there is little overhead of that type. On the other hand, it has a serious shortcoming, as described in 3.7.2 on page 29. Server load becomes extremely unbalanced as a large crowd of users gather in the same geographical area. It no longer helps that overhead is kept to a minimum, when all the work must be done by a single node.

Finding schemes that allow the system to survive such cases of user behavior is very interesting. The price of a higher overhead during normal operation may be acceptable if it allows smooth operation during extreme user behavior.

6.3 Graphical systems

Text systems similar to the test implementation discussed in this paper consist of isolated actions. However, a graphical system would most likely have continuous visual updates. Perhaps a useful analogy would be databases and their transactions versus servers that stream audio and video to their users. One problem occurs if a user can see across server node boundaries, possibly watching several objects that move and change continuously. The user may be watching objects on both nodes, possibly reacting to eachother and moving across the node boundary. It is probably hopeless to migrate these objects back and forth between nodes, many times per second, to provide users on both nodes with visual updates. Even if it would work, it is certainly not efficient.

A simple solution employed by some graphical games today is simply not to allow users to see across nodes (or "zones"). It would probably be beneficial to make node boundaries completely transparent to users, so that the experience of one large virtual world is not weakened.

One possible solution might lie in the use of read-only proxy objects. Nodes that need updates on a certain object could keep a read–only copy of it as a proxy instead of constantly migrating the actual object back and forth. Stale information in a proxy object could lead to inconsistencies, though. It is very important that the user sees events in the correct order, especially events that are dependent upon one another.

Another solution might be to build on the connection manager idea discussed in the previous chapter. A connection manager sits between the user and server nodes with a connection to each node. It could subscribe to events — from several servers if necessary. Events, such as visual updates, from several servers would be combined into a single stream of events passed on to the user. This way the user gets visual and audio information from more than one server at the same time. The problem, again, is synchronization. Users should not see the result of an

action before the action itself when they take place on separate servers. Perhaps the timestamps that the threads already have could be used, tagging each event with the timestamp of the thread that generated it. It may be necessary to allow other objects in the system to subscribe to events, not only connection managers. Computer controlled agents may need the updates, as well as other objects.

There are many other possibilities. The core difficulty lies in preserving consistency by presenting events in correct order while keeping the system highly responsive and efficient.

6.4 Extensive testing

Last but not least, it would be very interesting to test the implementation properly. For reasons discussed in the previous chapter the results were not extensive enough for substantial conclusions.

Real statistics would provide a basis for more interesting observations and conclusions. The ideal target for examination would be a large system built for a real purpose, with thousands of active users. Constructing such a system is unfortunately likely to be a large task even for a sizable team. However, it is hard to see how a system designed for thousands of users can be tested properly without having that many users actually use the system, and use it the way they would “in the real world.”

Bibliography

[1] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[2] Michael R. Blair, Natalya Cohen, David M. LaMacchia, and Brian K. Zuzga. MIT SchMUSE: Class-Based Remote Delegation in a Capricious Distributed Environment. In *1995 Lisp Users and Vendors Conference*, page 12. Association of Lisp Users (ALU), August 1995.

[3] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[4] Felix A. Croes. DGD. `http://www.dworkin.nl/dgd/`.

[5] Robert J. Fowler. The complexity of using forwarding addresses for decentralized object finding. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, pages 108–120, Calgary, Alberta, Canada, 11–13 August 1986.

[6] Eric Jul. Object mobility in a distributed object-oriented system. Technical Report 88-12-06, Computer Science Department, University of Washington, 1988.

[7] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

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

[9] Lars Pensjö. LPmud, a programmable multi-user game, 1991. (original LPmud documentation).

[10] Alexander Weidt. Dome, a distributed object oriented execution environment. Technical report, Technical University of Berlin, 1995. `http://autos.cs.tu-berlin.de/~demos/dome.html`.