Continuously Available Virtual Environments

by

James C. R. Purbrick, BSc (Hons.)

Thesis submitted to the University of Nottingham for the degree of Doctor of Philosophy, October 2001

Abstract

This thesis presents a framework for continuously available persistent collaborative virtual environments which is fundamentally more flexible than current approaches. Whereas existing systems allow the artefacts in the environment and the application behaviours of those artefacts to be changed at run time, they still need to be shut down if the infrastructure mechanisms of the system need to be changed. The framework presented by this thesis pushes run-time extensibility to a lower level allowing previously static infrastructure mechanisms and application level behaviours to be replaced and extended in a uniform way. By associating infrastructure mechanisms with artefacts in the same way that application behaviours are associated, the framework allows multiple alternative infrastructure mechanisms to coexist within the virtual environment system. Rather than applying a single infrastructure mechanism to all artefacts in a virtual environment, mechanisms can be tailored to an artefact's role, optimising the operation of each artefact. This allows a wider range of artefact behaviours and so applications to be supported by a single virtual environment. Infrastructure level behaviours may implement a single infrastructure mechanism or multiple mechanisms, allowing the framework to explicitly present the complex interdependencies which can exist between infrastructure mechanisms such as persistence and consistency. In addition to providing greater run-time flexibility for continuously available persistent virtual environments, the framework allows infrastructure mechanisms to be easily developed, compared, tested and configured, making it a useful test bed for the development of future infrastructure mechanisms.

After reviewing existing virtual environment systems and related systems, the thesis presents an experiment which reveals some of the problems existing with current approaches to persistence in virtual environments. The thesis then describes the framework discussed above and the issues involved in its realisation before evaluating the current prototype. Finally some conclusions are presented and future work discussed.

To Ali and Chris. One gave me the will. One gave me the way.

Acknowledgements

I would like to thank my supervisor, Chris Greenhalgh, who has taught me more than anyone else and has been an inspiration and guide for the last 6 years.

I would also like to thank the members of the Communications Research Group and Mixed Reality Lab, especially Adam Drozd for his help with the video figures.

Finally, I would like to thank my dad, Malcolm Purbrick, and mother in law, Anne Cooper, for their help proof reading this thesis.

Table of Contents

Table of Figures

1 Introduction

"*When* is the virtual environment?"

Something about this question seems strange.

"*Where* is the virtual environment?"

This seems more reasonable. In our modern networked society people often wish to know the network address of a web page or other service. Asking for the network address of a machine which allows access to a virtual environment seems sensible.

"*What* is the virtual environment?"

Again, this is a sensible question. A virtual environment might be a training ground for a simulation or exercise. It might be a fantasy world full of fairy tale creatures or a science fiction environment of revolving space stations. It might just be a place for people to meet and talk.

"*Who* is in the virtual environment?"

Another useful query. Collaborative Virtual Environment systems can now support arbitrary numbers of users, so asking who is present makes perfect sense.

But, "*When* is the virtual environment?"

This question sounds strange as in many cases we assume (virtual) environments are always there. With current virtual environment systems however this is often not the case. Most virtual environment systems require that the environment be designed using off-line tools, started, used and then shut down. In some cases this model is appropriate. Where applications require short tasks to be performed by groups of users together then a time can be agreed between the participants when they will start a virtual world, meet in it and then shut the world down.

“When is the virtual environment?”

“3:30pm”

Users may be distributed around the globe in different time zones. Some may need to work on lengthy tasks. The environment may need to support casual ad hoc meetings between users who happen to be on-line at the same time. In these cases the current model is clearly not good enough. The virtual environment must be constantly available and consequently any changes made to the environment must be made on-line and should persist in the face of partial failures.

There are virtual environment systems designed to support these applications which do provide persistence. They either periodically write the contents of the virtual environment to disk or log changes made to the environment or do some combination of these. The environment can then be made constantly available and changes made can be recovered in the event of failure. The field of database research provides many approaches to recovery which can be utilised to make virtual environments persistent.

While these virtual environment systems are able to make the environment appear constantly available and support the evolution of the content and behaviour of the virtual environment, they are limited in that the environment still has to be shut down for system maintenance. If a new application cannot be supported by the current system or the system needs to be patched, the users are again left asking, “When is the virtual environment?”

This thesis extends the current state of the art in persistent virtual environment systems by developing a framework which supports fundamentally more flexible virtual environments. The work proposes virtual environment systems which allow not only the on-line evolution of the content of an environment, but also of the applications which the system can support, and of the infrastructure of the system itself. The ultimate goal is to create systems which need never be shut down, so that users never need to ask, “When is the virtual environment?”

1.1 Background and Motivation

Collaborative Virtual Environments (CVEs) are computer-generated environments which provide spaces in which geographically separated people can communicate and collaborate. In order for users to communicate with each other they must normally be present in the virtual environment at the same time, making CVEs generally a same time, different place (synchronous) communication technology. The environment contains representations of each user present in the environment, allowing users to be aware of the other users with whom they can currently communicate. When a user joins the environment a representation of their presence is created indicating that they are able to communicate. When a user leaves, their representation is removed from the environment indicating that the user can no longer be contacted.

Given this definition, a wide range of applications can be considered to be collaborative virtual environment systems. Text based communication systems such as IRC (Oikarinen and Reed, 1993), ICQ (ICQ Inc., 2001), AOL Instant Messenger (America Online Inc., 2001) and chat room applications (Bradner et al, 1999) all provide representations of the users present in a virtual space and so are in some senses CVE systems. Yahoo! on-line services (Yahoo!, 2001) also allow users to see when other users are available on-line for synchronous communication. Yahoo! mail allows users to see whether the sender of an e-mail message is on-line when the message arrives – blurring the line between the synchronous (same time, different place) CVE communication and asynchronous (different time, different place) e-mail communication.

There are also many multi-user, networked systems which cannot be considered CVEs. Multi-user database systems (Date, 2000) support concurrent access to a shared corpus of data, but go to great lengths to isolate the effects of one user from another. Rather than representing users who are accessing the same data, multi-user databases appear to be single user systems. Early VRML browsers (Carey et al, 1997) were also not CVE systems as the initial VRML specification provided no facilities for representing other users simultaneously accessing the VRML environment or for allowing communication between those users. Virtual worlds could be downloaded across a network, but once running in a browser they were single user experiences. Although the early VRML worlds provided 3D graphical virtual environments, they were not collaborative virtual environments.

The CVE systems which form the main focus of this work are systems which provide similar 3D graphical virtual environments to VRML, but which also support collaboration. Users appear to each other as 3D models, referred to as *avatars* or *embodiments*. Users view the virtual world through their avatar's eyes and so can view different parts of the virtual world by moving their avatar representation. Communication is achieved either by providing text chat to accompany the 3D view (Vellon et al, 2000) or by recording a user's speech and replaying it to other users in the environment (Greenhalgh and Benford, 1995). Visual cues might be used to integrate the 3D graphics with the communication, for example, by displaying a user's typed text above their avatar's head or by animating an avatar's mouth to indicate an audio source.

The explicit positions provided by graphical CVE systems promote the environment from being simply a container for a list of mutually aware users to a continuous medium, which can drive many aspects of the system. Positions determine the parts of the environment a user can see and might also determine which parts of the virtual world are communicated to a user across the network, or which events in the world are of interest to a particular user. The distances between users can be used to calculate the volume at which users overhear each other. Expressive frameworks such as the spatial model (Benford and Fahlén, 1993) allow the relative positions of users to be combined with their levels of projection or attentiveness to determine the mutual awareness of users. Inattentive users might only notice a user drawing attention to their actions, while attentive users might be aware of every other user in close proximity.

In addition to using explicit positions to drive communication, most 3D graphical CVEs support environments populated with interactive artefacts in addition to user embodiments. These artefacts might be as simple as terrain artefacts, which alter an avatar's height as they are traversed, or walls, which prevent passage, to complex visualisations of data, which can be collaboratively modified and discussed by groups of users. Training or simulation environments might include artefacts simulating real world equipment which permit users to learn how to use the equipment without risk. Environments used for learning or therapy help users to tackle problems at their own pace without distractions present in the real world. Multiplayer games provide fantastic virtual environments populated with potential enemies or artefacts which might form puzzles or allow progression through the game. Interactive environments greatly increase the gamut of applications which CVE systems can support. Instead of being purely graphical backdrops for communication these environments support diverse applications in which communication and collaboration can form a greater or lesser part.

While state of the art CVE systems support high-resolution 3D graphics, real time audio streaming, arbitrary numbers of users and infinitely large interactive virtual environments, many do not support persistent changes to those virtual environments. Users can enter a virtual environment, communicate with other users and change the virtual environment by interacting with artefacts, but in many cases those changes are transient. If the CVE system is closed down, or fails, all of the changes made to the environment are lost. If the environment is started again it will return to an initial state as if nothing had happened. As already argued, this is a major limitation and greatly restricts the applications which many current CVE systems can support.

There are a few CVE systems which solve some of these problems by supporting persistent changes to the virtual environment (Vellon et al, 2000, Curtis, 1997). Typically these systems use standard database techniques such as checkpointing or logging to either periodically write the state of the environment to disk or record each change to the environment as it occurs. In the event of failure the system can be restarted by either reading the latest checkpoint or reapplying the changes in the log to an initial state to restore the changes made before the failure. Technically adding these facilities is relatively easy and allows the content of a virtual environment to evolve with use. Rather than stopping the system and making changes to the environment off-line, the system can keep running, changes can be made and those changes will be durable in the face of failure.

Some CVE systems go a step further allowing artefact behaviours to be added, modified and removed while the system is running. These facilities load, unload and link code into the running system and so allow the applications of a virtual environment to evolve with use. An environment can be started and used to support an initial application and then evolve to support new activities without shutting down. These facilities are particularly useful when virtual environments are used to support communities and so must be elastic enough to support the changing needs of the community. Some MUD, Object Oriented (MOO) systems, such as LambdaMOO (Curtis, 1997), support the run-time configuration of object oriented inheritance hierarchies, which allow new artefact behaviours to be loaded which extend existing behaviours. These extremely flexible code-loading facilities result in many virtual worlds in which run-time evolution is the primary activity. A large proportion of users' time is spent building new artefacts and behaviours or exploring parts of the world built by others.

Even these systems, however, stop short of supporting on-line system evolution. If a behaviour requires facilities which cannot be provided by the current system architecture, even systems like LambdaMOO must be shut down and updated off-line. This work explores the feasibility of building systems which support a notion of system evolution by allowing every aspect of the system architecture to be replaced at run-time. It also examines state of the art approaches to persistence and run-time extensibility to see if any are more appropriate for CVE systems than the pragmatic solutions currently used by commercial persistent CVE systems. The goal is to develop a system which is truly continuously available and persistent.

1.2 Problem Breakdown

The previous section introduced CVEs, explored the spectrum of CVE systems, from text based chat systems to interactive audio-graphical environments, and presented the need for and current lack of facilities for persistence and extensibility. This section focuses on the implications of making virtual environments continuously available. By breaking the problem down into fundamental elements a clearer understanding of the problem can be gained.

1.2.1 Continuous Availability

Continuous availability may be for pragmatic reasons, such as the need to support users distributed across time zones and so requiring access to the environment 24 hours a day or a need to support casual access, which implies a lack of co-ordinated start up and shut down. Alternatively, continuous availability may be needed to maintain the illusion of an alternate reality. This requirement is most likely needed by environments used for gaming or entertainment. The goal in these applications is to present an immersive experience of another world. While high quality graphical and audio rendering and a highly interactive environment contribute to this immersion, without continuous availability the illusion of a parallel world is broken. Although conceptually the environment exists as a parallel reality, if a user can only visit this reality at certain times, or occasionally cannot visit it because it is shut down for maintenance, then the spell is broken. Once the edges of the reality are reached, the game is up. Once Truman finds the door at the edge of the ocean, the Truman Show is over (Paramount Pictures, 1998).

While these applications have the greatest need for continuous availability, there are few applications which would not be enhanced at all by the facility.

Continuous availability removes a barrier to the use of CVEs by removing the need for often complex and time consuming start up procedures to be followed before an environment can be used. The telephone would be far less widely used if it required 15 minutes of configuration to be carried out before a 30 second call could be made. By making all virtual environments continuously available this arbitrary start up time is eliminated. At a time when web sites, FTP services and a plethora of other web services are available around the clock, it seems strangely anachronistic that in this increasingly 24/7 society advanced technologies such as CVE systems are only available when someone turns them on.

1.2.2 On-Line Evolution

The need for continuous availability combined with the need for interactivity produces a basic need for persistence. Where virtual environments are not interactive, continuous availability can be tackled in isolation. The virtual environment is a backdrop for communication and collaboration which can be defined in a static manner. The system is started, the world created and used. In the case of failure the system can be restarted from the same initial definition. Where environments are interactive, that interactivity can change the state of the world. If a button is pressed or an artefact moved, the environment moves to a new state. A persistence service is needed to record this new state so it can be recovered in the event of failure. This evolution of content can be viewed as the simplest of 3 kinds of on-line evolution which would be supported by an ideal CVE system.

1.2.2.1 Content Evolution

The progression of the virtual environment through a series of recorded states can be viewed as content evolution. Although an initial state may be defined for the environment, once interaction alters the state and the new state has been recorded via the persistence mechanism, the content of the virtual world can be seen to have evolved.

In addition to this view of content evolution as a side effect of interaction, CVEs designed to support continuously available virtual environments must specifically support on-line content evolution wherever a virtual environment might need to be changed throughout its existence. In practice this is likely to be every virtual environment. Just as software systems are increasingly designed to evolve with user requirements, so virtual environments must be able to do so. It is unlikely that any continuously available virtual environment will perfectly fulfil its users' needs either initially, or over time. If the needs of the users change and the environment cannot change with them then eventually the environment will not be able to support its users and so stop being used. If a continuously available virtual environment is to evolve it must be able to do so on-line.

Content evolution is relatively easy to achieve technically, as it only requires that facilities exist to update, add and delete artefacts from the virtual world and for the results of those operations to be persistently stored. A more challenging requirement is that facilities need to exist to manage the evolution. Benign changes must be allowed, but virtual vandalism must be impossible or hard or at least traceable.

1.2.2.2 Application Evolution

Rather than dealing with changes to the artefacts in the virtual world, application evolution is concerned with allowing the addition, update and removal of artefact behaviours in the virtual world. It should be possible to add new artefact behaviours which allow artefacts to be interacted with in new ways and so support new applications. It should also be possible to remove or replace old behaviours when they become unsuitable or unneeded. The need for application evolution is an extension of the need for content evolution. It is difficult to predict the exact behaviours required for a virtual environment before it is used and even more difficult to predict the behaviours which might be needed in the future. In order to avoid the system being shut down to alter the set of available behaviours, the CVE system must support application evolution. Technically application evolution is a challenging task. New code must be able to be introduced and removed from the running system and the changes stored persistently in case of failure.

1.2.2.3 System Evolution

A continuously available virtual environment which supports content evolution and application evolution is able to exhibit a great amount of flexibility in the face of changing requirements. Artefacts in the virtual world can be added, changed and deleted along with the application code which defines their behaviours. However, such a system might still need to be shut down in order to patch the system itself. Should the infrastructure be found lacking, for example unable to support new functionality required by the users, then it must be shut down, patched and restarted with the new infrastructure capabilities. Ideally, this should not be the case. Any part of the system should be able to be extended or replaced in the same way that application code can be changed to provide new artefact behaviours. Aspects of the CVE system which are traditionally static parts of the platform – such as distribution, replication or consistency mechanisms – must become as flexible as application behaviours. In addition CVE systems which support system evolution must be able to change the mechanisms which are used to support higher-level evolution mechanisms, such as the access control mechanisms used to control content and application evolution. Clearly, such systems must have carefully designed fixed points to control access to the extension mechanisms without limiting the ability of the system to evolve.

1.3 Challenges and Approach

Only a CVE system which could support content, application and system evolution could remain constantly available in the face of changing needs. However, building such a system is clearly a challenging task. If running a CVE system continuously could be compared to juggling, content evolution would be the equivalent of changing the number and type of the objects being juggled. Application evolution could be compared to changing the order in which the objects are thrown and caught, while system evolution is akin to changing the laws of gravity and Newtonian physics while attempting to keep all of the balls in the air.

Historically, continuously available CVE systems have been built pragmatically in the sense that they have applied simple, standard approaches to persistence and extensibility, either because they have been commercial systems, or have been most concerned with the sociological aspects of on-line collaborative worlds. As such they have concentrated on the low hanging fruit of on-line evolution – content and application evolution. Both can be tackled with relatively simple approaches which nevertheless result in a highly flexible system which can support on-line changes to both the content and behaviour of the virtual world. In addition to looking beyond the simple approaches taken by existing systems to content and application evolution, this work maps out the largely unexplored area of system evolution. In both cases the approach taken is to look outside the field of CVEs, to examine the state of the art in persistence and run-time extensibility and reconfigurability. Where mechanisms exist they are evaluated and where appropriate tailored for use with CVEs. Where mechanisms are not found they are developed. The ultimate goal of this work is to develop a framework for a next generation of continuously available persistent CVE systems which can evolve at run time to support any demand made of them.

1.4 Scope and Organisation of Work

The field of CVE research brings together a wide array of academic disciplines including computer science, psychology, sociology and ethnography. This thesis, however, concentrates on the technical computer science and software engineering issues of continuously available persistent CVE systems.

As has previously been mentioned there is a body of sociological study of persistent CVE systems (Morningstar and Farmer, 1990, Churchill and Bly, 1999, Garton et al, 1997). In contrast, while there is a wealth of research published on other technical aspects of CVE systems, the commercial persistent CVE systems which make useful tools for studying the sociology of on-line communities do not expose technical details for business reasons. Because of this an exploratory study was performed at the beginning of this work to identify the technical challenges facing continuously available persistent CVEs. This experiment is described in Chapter 2.

Chapter 3 is split into two main parts. The first reviews existing CVE systems from a technical perspective focusing on their support for persistence, content management and run-time extensibility. The second part reviews systems and mechanisms from outside the field of CVE research which may be applicable to future continuously available persistent CVE systems.

Chapter 4 proposes a new approach to CVE design, which treats each item differently from an infrastructure perspective and argues that this approach is both more flexible, efficient and increases the gamut of applications which CVE systems can support. It presents a novel framework to support this per-item approach which forms the main original contribution of this thesis. The framework introduces two new concepts. The Distributed Event Filter (DEF) framework is presented as an extremely flexible approach to low level extensibility which allows the run-time modification of mechanisms such as consistency and event distribution, which have historically been part of the static infrastructure of CVE systems. Deep Behaviours are presented as a complimentary higher-level concept which allows the orchestrated addition, configuration and removal of distributed event filters. The chapter presents several example DEF configurations which illustrate how existing CVE infrastructure mechanisms can be realised with the framework and presents novel mechanisms developed as part of this work.

Chapter 5 provides details of the prototype implementation of the DEF/Deep Behaviour framework presented in chapter 4. It provides a UML illustrated description of the design of the prototype along with sequence diagrams and descriptions of the important details of the framework. The implementation of important mechanisms which support the framework are also discussed, such as the store implementation which provides flexible persistence facilities used by many prototype filters.

Chapter 6 evaluates the framework by comparing existing approaches used by current CVE systems to the per-item infrastructure approach proposed by this thesis. Current approaches to both persistence and caching are compared against per-item approaches enabled by the DEF/Deep Behaviour framework and in both cases the per-item approach is shown to be more efficient. A final experiment shows that at least some item roles identified in one application can be applied to other applications.

Chapter 7 identifies the contributions made by this work to the field of CVE research specifically and to computer science in general. It presents a number of areas in which the work can be extended and areas of complementary research which could be performed. Finally it presents some short conclusions which attempt to paint a picture of the future of persistent collaborative virtual environments and the role of this work within it.

2 Exploratory Experiments

While many systems and mechanisms exist to persistently store data beyond the lifetime of any application, few graphical virtual environment systems exist which provide these facilities and no technical analysis has been made of the particular requirements of persistence in collaborative virtual environments or studies made of the persistent data generated by existing systems. For this reason a set of experiments was run early in the course of this work to explore the characteristics of persistent data in collaborative virtual environments and the expectations and experiences of users of persistent collaborative virtual environments. An intentionally simple persistent virtual environment system was implemented to run these experiments with the results and insights gained from the experiments fed into the design of the framework detailed later in this Thesis. In addition to characterising the needs of collaborative virtual environment systems with regards to persistence, these experiments highlight a number of facilities which must be provided by persistent virtual environment systems above and beyond the provision of persistent data.

Section 2.1 identifies activities within persistent virtual environments which are application independent and have the greatest impact on the provision of persistence. These foci are used to develop the experimental scenario outlined in section 2.2. The collaborative virtual environment platform, persistence implementation and client modifications made for the experiments are discussed in sections 2.3 to 2.5. The experimental results are shown in sections 2.7 and 2.6 and some conclusions presented in section 2.9.

2.1 Experimental Design Goals

The task of designing an experiment to analyze the use of continuously persistent virtual environments is a challenging one. Most previous applications of continuously persistent virtual environments describe worlds which existed for months or years at a time. They have supported hundreds or thousands of users who have became familiar with their world or changed it to better support their community. In some senses continuously persistent virtual environments only make sense when used in this way. If a world is only used for hours at a time, the ability for the world to change while it is used becomes less critical. The chances of failure are reduced and so the durable storage of data becomes less essential. The likelihood that some unforeseen and unsupportable activity will need to take place is similarly reduced, as is the ability of the environment to become a familiar space which supports a community. Despite this it is possible to perform worthwhile short-term experiments to analyse the use of persistent virtual environments by focusing on the activities which are critical to persistence. In the same way that crash tests do not involve a great deal of open driving, activities which do not directly affect persistence can be removed to provide a focused analysis of the critical moments (when car hits wall). In order to do this, features expected to be common to the vast majority of continuously persistent virtual environment applications are identified in sections 2.1.1 to 2.1.4 before an experiment designed to test these features as intensely as possible is presented in section 2.2.

2.1.1 Communication is Key

Existing and historic uses of continuously persistent virtual environments focus on the provision of social spaces which are always available and allow the formation of communities within the space. While other applications of continuously persistent virtual environments are likely, these applications will still hope to leverage the facilities for communication which collaborative virtual environments provide. An application providing a persistent environment for collaborative design would still place a great deal of importance on communication between the designers. In some sense the only reason to use a collaborative virtual environment is to enable communication between users. As such communication between users should at least form part of the application.

2.1.2 Online Evolution

One of the biggest challenges in continuously persistent virtual environments is the evolution of the world while the system is used. Cleaning must go on as in a 24/7 burger bar (Capps et. al., 1999). Our experiment should thus combine the activities of social interaction and world evolution – users should be aware of changes being made to the world, but be able to continue communicating. The users changing the world should have to work around others, be aware of their activities and be able to communicate with other users not contributing to the changes. Users should be able to give suggestions to those making changes they can witness, or at least be able to lie in front of the bulldozer (Adams, 1979)

2.1.3 Ad Hoc Modification

Continuously persistent virtual worlds should be able to be changed using standard user clients with a single human viewpoint and standard manipulation tools. This allows the world to appear malleable, to evolve and to allow rapid modification of the environment to suit its use. While large scale changes to an environment might be performed using a CAD style system with multiple viewpoints, or programming tools which allow the modification of behaviour, users with a standard client must be able to perform ad hoc modifications to the world to facilitate activity. A table should be able to be moved out of the way to accommodate people. Without ad hoc modification facilities the world becomes a rigid backdrop to users activity even if they can view changes performed by others. Not being able to change the world with a standard client creates a multimodal approach to use where users switch clients to make a change and then switch back to use a world. This discontinuity is wasteful and discourages rapid interleaving of modification and use. Instead of the world evolving with use, users resort to a "design then use" approach, which is indicative of current virtual environment systems.

2.1.4 Accelerated Aging

While continuously persistent environments should be able to evolve with use, this evolution is likely to be slow, with the environment staying static while it is suitable for the needs of the users and only being changed when it becomes inadequate. However, it is the changes and not the periods of stability which are of most interest to this study of persistent virtual environment systems. For this reason an accelerated aging approach was taken in the experiments – while the interleaving of social interaction and modification was encouraged, the modification of the environment was a primary goal of the experiment. Rather than waiting for a modification to become necessary, modification was an aim.

Experiments intensely focusing on the issues above are complementary to large-scale studies in the same way that crash tests are complementary to statistics on road safety. Laboratory experiments are much easier to control and to gather microscopic detail on the operation of persistent virtual environments, while large-scale studies provide a macroscopic view with larger volumes of statistical information. It will be interesting in future to see if the insights and results gained in these experiments apply to longer trials.

2.2 Experimental Scenario

Given the focus on the issues identified above, it was decided to make the user goal of the experiments the on-line creation of an art gallery using a standard user client with only minor modifications to support the addition and deletion of items from the virtual world.

Over a period of 3 weeks, 20 volunteers split into groups of 2 to 6 were asked to create part of the museum in an initial 30-minute session. In this session they were asked to spend 10 minutes familiarising themselves with the user client, 10 minutes creating a room or adding to another room in the world and then 10 minutes to freely create and modify objects as they saw fit. Subjects were encouraged to talk to each other throughout the task resulting in this session simulating the critical moments in a persistent virtual environment where some users are modifying the world as others use it for communication.

The volunteers return to the world for a second 10-minute session in different groups and are asked to show one other around the museum and to see how the world has changed. This second session was designed to see if the users were able to navigate around the world which had changed; understand how the world had changed in their absence and to see whether the ability to modify the world was useful even when performing the apparently passive task of guiding others around the world. Whereas the first session was designed to provide an accelerated view of world evolution, the second session was designed to see if, having learned how to change the world, subjects found the facility generally useful.

The volunteers were asked to complete questionnaires about the ease of use of the system and their perceptions of the change in the world after each session. The activity in the world was logged during each session allowing detailed analysis of the changes taking place and a comparison between the volunteers' perceptions and the actual events in the world, which in many cases was very revealing.

2.3 MASSIVE-3

The collaborative virtual environment platform adapted for use in the experiments was the MASSIVE-3 system (Greenhalgh et. al, 2000). Sections 2.3.1 and 2.3.2 provide an overview of the system components and section 2.3.3 describes the serialisation mechanisms used as the basis of persistence in the experiments, before section 2.4 details the persistence facilities developed specifically for the experiments.

2.3.1 Environments and Agents

Each MASSIVE-3 application is represented by an "agent", which normally corresponds to a thread of execution. "Environments" in MASSIVE-3 are databases which contain part or all of a virtual world as a hierarchical scene graph. Environments provide an API which applications use to create, destroy or update the world state, or be notified when another agent has updated the state. The two main types of MASSIVE-3 applications are clients and servers. Servers create an agent and an environment and then publish the existence of the environment with a Trader service. Clients create an agent, locate a server by querying the Trader service and then ask the server for a replica of its Environment. Having received the replica the client can query and update its local replica which will generate events which are first sent to the server and from there multicast to other clients.

2.3.2 Items

A MASSIVE-3 environment contains a scene graph made up of "items" of different types. Each artefact or "thing" in the virtual world is represented as a sub-tree in the scene graph. An "entity" item at the root of the sub-tree describes the artefact's position, orientation, scale and extent. The remainder of the sub-tree comprises "geometry" items which reference 3D geometry describing the appearance of the artefact; "attribute" items containing <name, value> tuples which annotate the artefact; further sub-trees which describe sub-parts of the artefact which should be positioned relative to the artefact and "switch" items which can be used to change an item's appearance. The hierarchical nature of the scene graph allows complex composite artefacts to be moved around the environment just by updating the entity at the root of the artefact's sub-tree. In addition to the basic item types, custom behaviours can be given to artefacts by annotating them with "behaviour" attributes. Whenever a behaviour item is replicated it causes the method implementing the custom behaviour to be called by the replicating agent.

2.3.3 Serialisation

In order to replicate MASSIVE-3 environments across network connections and the communication of updates, nearly all MASSIVE-3 classes support serialisation and deserialisation using a similar mechanism to Java 1.0 (Gosling et. al., 1996). A MASSIVE-3 object can be given an ObjectOutputStream and asked to write itself to that stream or an ObjectInputStream and asked to read its state from the stream. The object uses methods implemented by the stream to put or get its primitive members such as floats or strings and passes the stream to its object member's own readObject or writeObject methods. The stream classes are abstract interfaces implemented by a number of concrete implementations in MASSIVE-3 to allow the serialisation of objects to and from files or network streams in ASCII or binary formats.

2.4 Persistence Facilities

The implementation of persistence used in these experiments was based on periodically checkpointing Environments. To support reading and writing checkpoints, a PersistentApplication server was developed which either deserialises an environment from a checkpoint or creates a new locale of a given name. The application then runs the environment as a standard MASSIVE-3 server and periodically serialises the state of the environment to a new checkpoint and registers the command line needed to restore the environment with the Trader. If the environment run by the PersistentApplication is unused for a given period of time, the application terminates. If a client wants to replicate a locale which is not currently being run by a PersistentApplication, the Trader transparently starts a new PersistentApplication using the command line registered by the last application to serialise that environment, before returning the network address of the new application to the client. The termination and on demand creation of application processes allows a server to support a much larger number of environments than if applications had to run all of the environments continuously. If users only use a subset of the environments only those environments are run by server processes. As activity moves around the world, new environments are run and unused applications terminate. This model could be easily extended to support load balancing across multiple servers by the Trader choosing a server to start each new process on from a pool, or by clients connecting a random Trader from a cluster of servers which all have access to the pool of Environment checkpoints.

2.5 Client

The client used in these experiments was based on the standard MASSIVE-3 user client with additions to allow the creation, deletion and modification of objects in the world. Movement in the client is achieved using the left mouse button – dragging up the screen moves forward, down the screen moves backwards, dragging to the left or right turns the avatar. To simplify navigation for novice users, only 2D movement was possible in the experiment. The right mouse button is used to manipulate objects. Right dragging objects moves them in a sphere around the avatar - as if the object was on the end of an arm. Double right clicking on an object picks it up. Once picked up, objects were moved with the avatar and can be rotated, scaled, dropped or deleted from the world using icons in the visor or keys. To make aligning objects easier a grid snap mode could be used to snap manipulated object positions, orientations and sizes to course units. Objects could be added to the world by scrolling through a palette displayed on the visor and the right clicking on iconised representations of available objects, which were then added to the world just in front of the avatar. A screen shot of the interface is shown in Figure 2-1 and a demonstration of the client is shown in Video Figure 1.

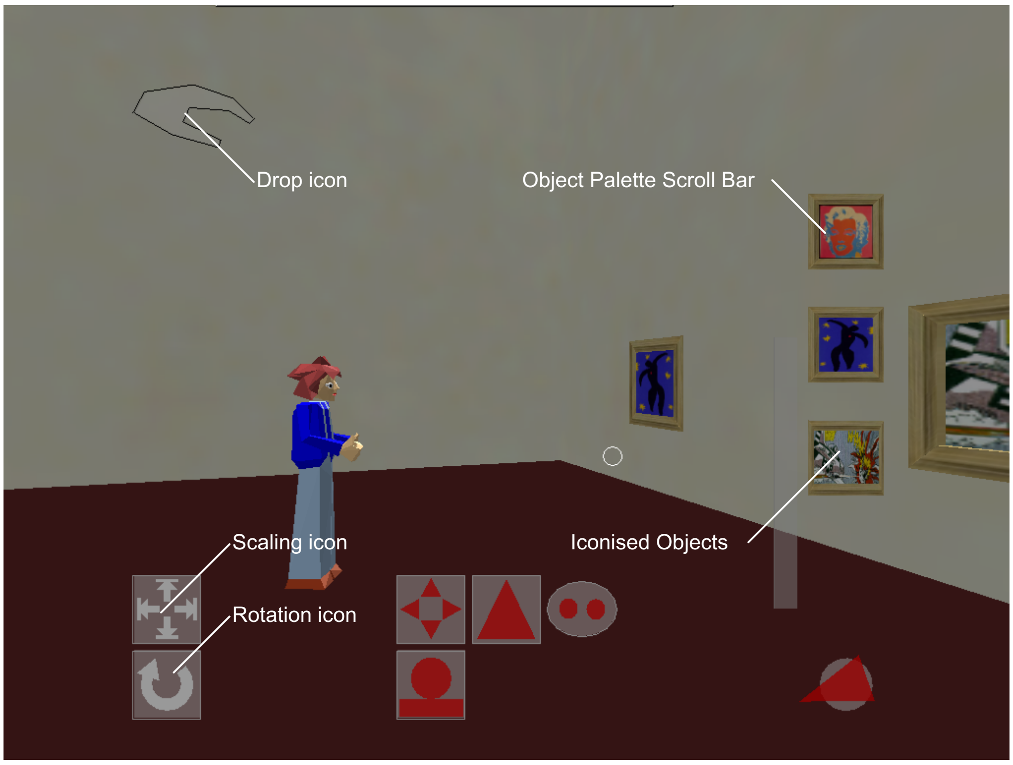


Figure 2-1 Interface to the on-line editor

An early version of the editor used widgets embedded in the world in addition to those in the visor. Resizing walls and boundaries was achieved using handles which appeared on the corners when an avatar approached, however this multi-modal interface, using proximity and widgets in the world and visor, proved too complex. The original version also allowed users to create their own locales and manipulate boundaries between them, but this required users to understand the concepts behind locales and, as locales could overlap, it allowed users to stand in apparently the same location, but be in different locales and so be unable to communicate. A further problem was caused when users ignored the original locale creation facilities performing all of their editing in a single locale, which ultimately became overcrowded and overwhelmed the client machines rendering capabilities.

To address these issues, the locale manipulation facilities were removed from the editor and a torroidal universe made up from a grid of linked locales was used in the experiments. By moving away from cluttered areas users move into new locales and so spread the world content amongst the available locales making the world more scalable. By defining boundaries in advance to ensure locales do not overlap users are always able to communicate with neighbours and manipulate objects that they are close to.

The goals in developing the client were to keep it close to the standard MASSIVE-3 user client, so changes represented the ad hoc modification described above, while making it as easy to use as possible so it coloured the experimental results as little as possible. For this reason several versions of the editor were developed and evaluated even though it was persistence and not on line editing which was the subject of the experiment.

2.6 Quantitative Results

In the detailed analysis of logged activity the goal was to characterise how objects changed in the persistent virtual environment. By knowing how objects change the persistence facilities might be tailored to the specific requirements of CVEs. By finding how much differentiation there was between artefacts the goal was to discover whether a single mechanism for persistence could be applied to all artefacts in the virtual world or whether a range of mechanisms were needed, which, given the wide range of data in a typical CVE, seemed more likely.

There are a number of dimensions along which patterns of activity and differentiation can be examined, including an item's semantic function (added items or embodiment items), its type within the scene graph (3D transforms, geometry, text attributes etc.), its appearance (wall, cube etc.). There are also several variables which can be measured along these dimensions; updates over time, number of items, lifespan, number of updates.

Comparing updates to added (authored) objects with updates to embodiment objects reveals a major differentiation, with 372765 updates made to 38 embodiments compared to 39665 updates to 596 added objects, i.e. there were 10 times as many embodiment updates to 6% of the items.

Furthermore, making embodiments persistent does not make sense as an embodiment represents a running application and so should not exist beyond the application's lifespan. In these experiments not making users' positions persistent would reduce by an order of magnitude the amount of data that needed writing to disk.

As previous research has been carried out on patterns of embodiment behaviour in CVEs (Greenhalgh, 1997), most of the following analysis focuses on the persistent, mutable objects added to the world.

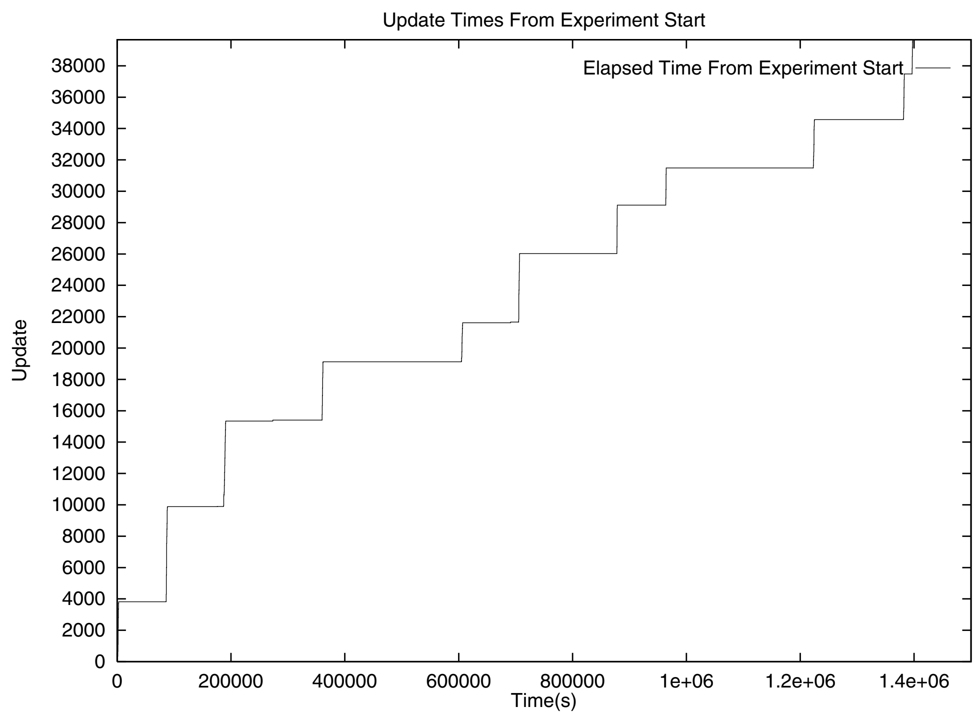


Figure 2-2 Update times from experiment start

Figure 2-2 plots the update times of added objects measured from the experiment start and clearly shows staircasing caused by short sessions interleaved with lengthy idle times. While such pronounced staircasing would be unlikely in a continuously available world accessed from around the globe, there are likely to be idle periods for some locales in large worlds when the server can be stopped to save resources. While fewer updates occur in later sessions, when users where asked to explore rather than create, but the decrease is not pronounced – users updated added items frequently, even though they were “visiting”.

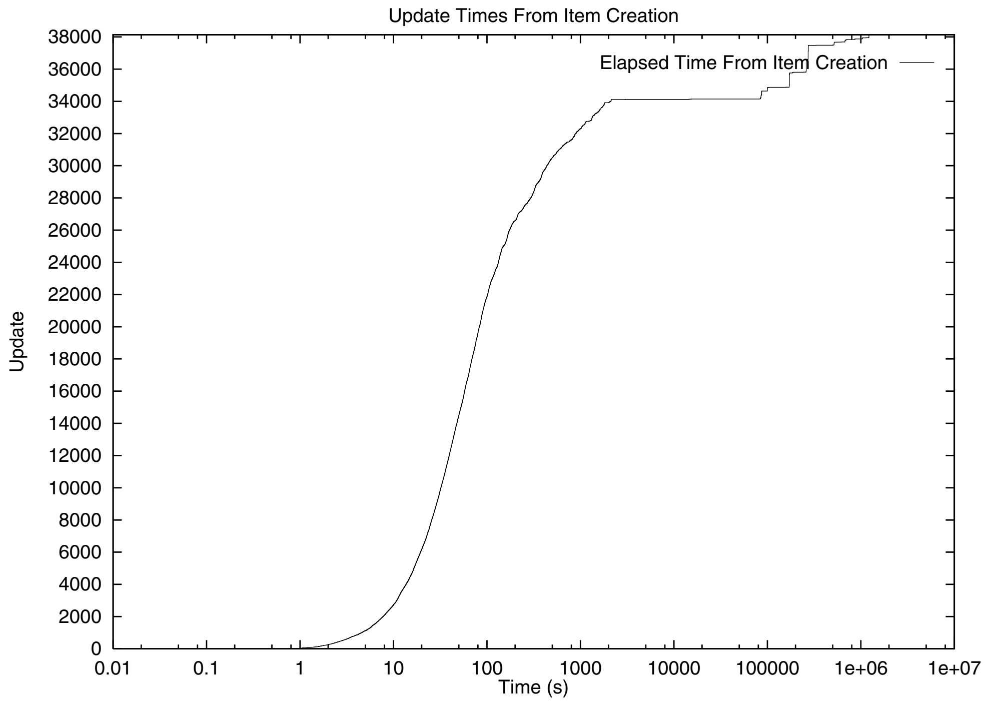


Figure 2-3 Update times from item creation

Figure 2-3 plots the elapsed time from an added item's creation to when an update occurs. 85% of updates occur between 10 seconds and 1800 seconds after creation, with the vast majority occurring within the first 2 minutes. This shows a pattern typical of behaviour in the experiment – items were created, manipulated it into an initial state, then either left, deleted or modified before logging off. The staircasing at the right hand side of the graph shows updates occurring in sessions after the session in which an object was created. These updates account for around 10% of the total, showing that items were returned to but that inter session updating was far less common than initial manipulation after object creation.

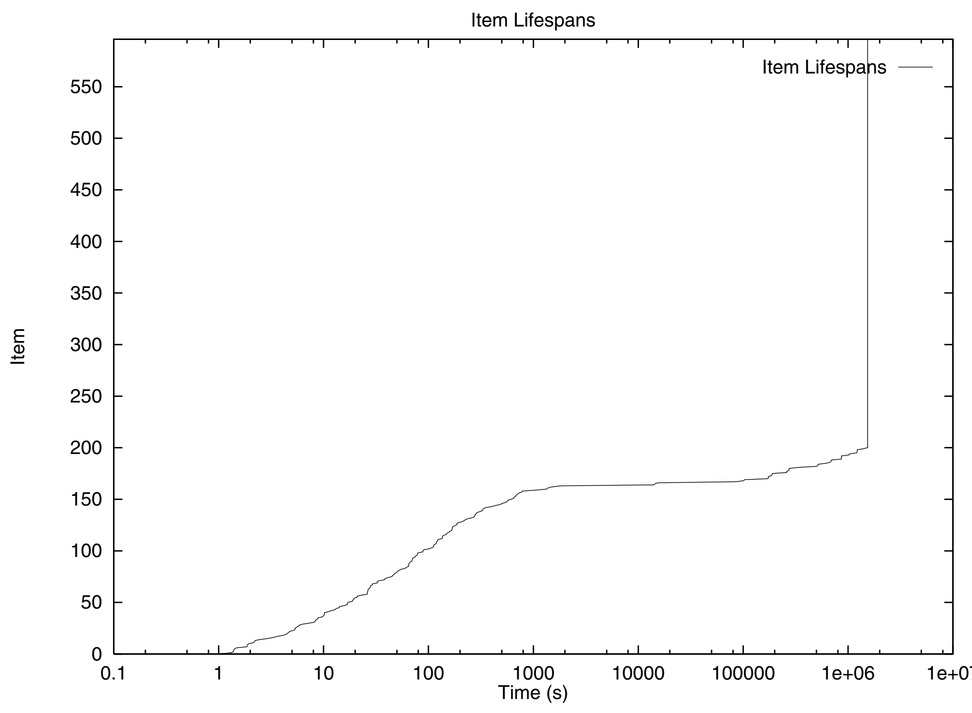


Figure 2-4 Life spans of added items

Figure 2-4 shows the life spans of added items, sorted from short to long. The life spans grow exponentially to between 600 and 1800 seconds, the length of a session, and then a few items are deleted in following sessions. 66% of added items remained at end of experiment, although it was clear that many were discarded while others provided important landmarks. The analysis of updates from item creation and lifespan both show a significant difference between activity during the session in which an object is created and subsequent sessions. This suggests that the moment when a user logs off after creating an item might mark a useful point in the management of the item's persistence.

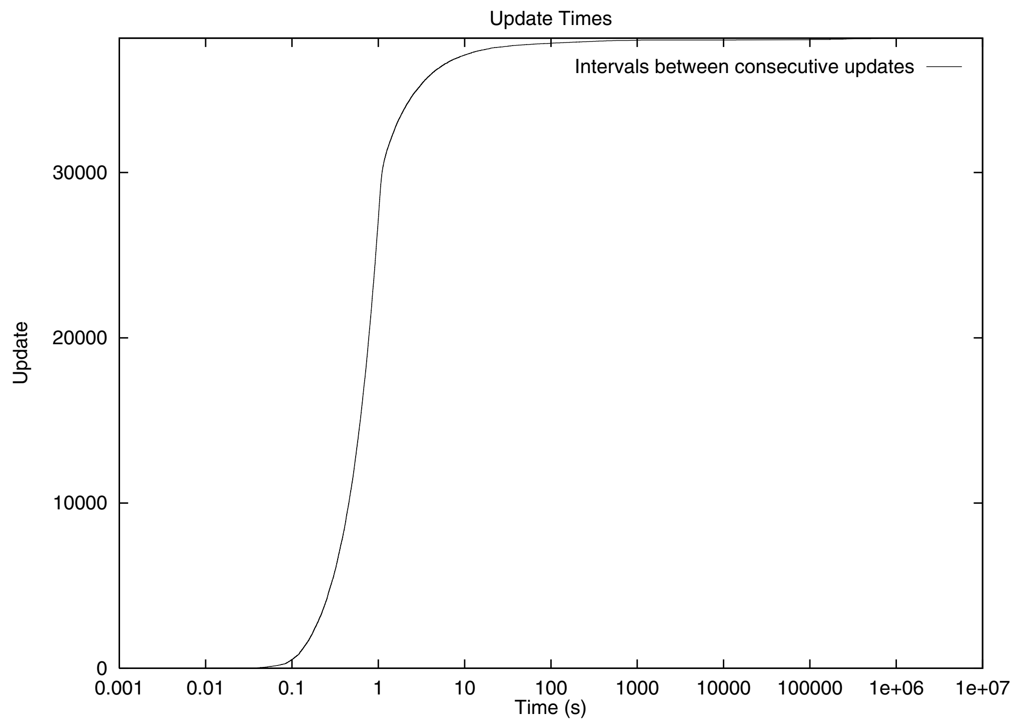


Figure 2-5 Intervals between consecutive updates on added items

Figure 2-5 shows the intervals between consecutive updates made to added items. The shape of this graph is governed mainly by the continuous stream of updates generated by waving artefacts around. In a centralised high latency system it would be unendurable for users to see the durable state of items when modifying them in this way.

Breaking the analysis of added items down by geometry reveals the different ways in which different artefacts were manipulated in the experiments. It is interesting to see that there were differences as only the solid wall was any different in any technical way. The other artefacts differed only in appearance and yet show differentiation in the ways in which they were manipulated.

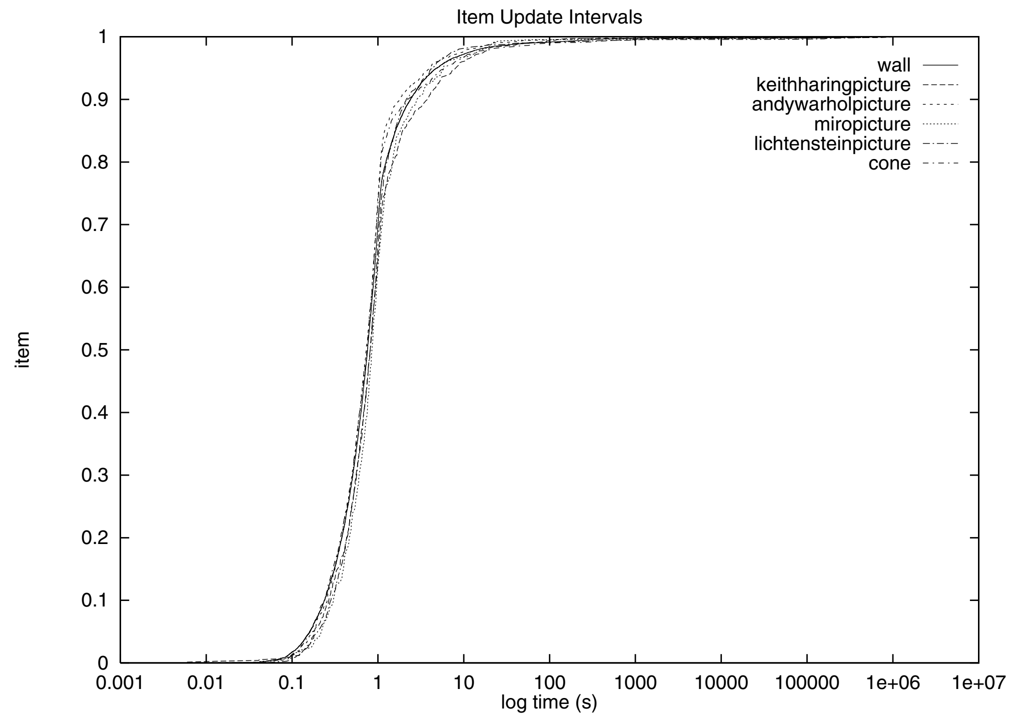


Figure 2-6 Intervals between consecutive updates by geometry

Looking at the intervals between updates by geometry in Figure 2-6 shows little differentiation as, once again, the shape of the graph is governed by the continuous nature of the updates generated as artefacts were moved around.

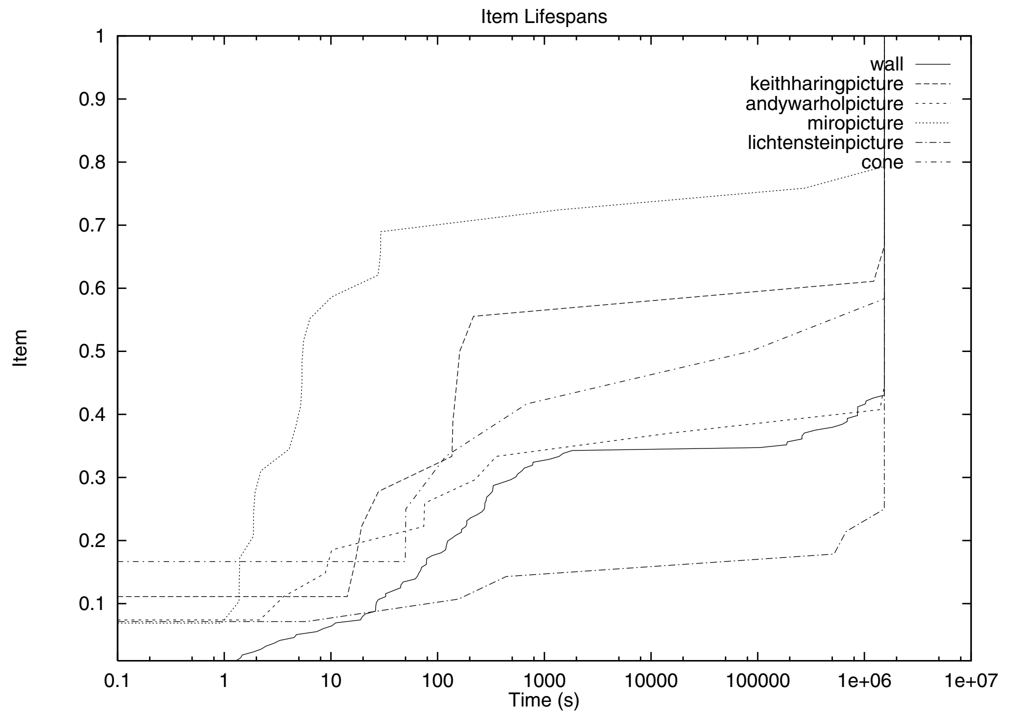


Figure 2-7 Item lifespan by geometry

However, significant differentiation is seen in Figure 2-7, which shows item lifespan by geometry. The popular Lichtenstein picture clings to the bottom of the graph and 70% of these pictures remained at the end of the experiment compared to the less popular Miro picture which tended to be added, evaluated and deleted within a few seconds, with only 25% remaining at the end of the experiment. Although walls were important landmarks which provided structure in the world, the graph shows that around 35% were deleted in the session in which they were created, with many people repeatedly adding, manipulating and deleting walls before they ended up with the desired configuration.

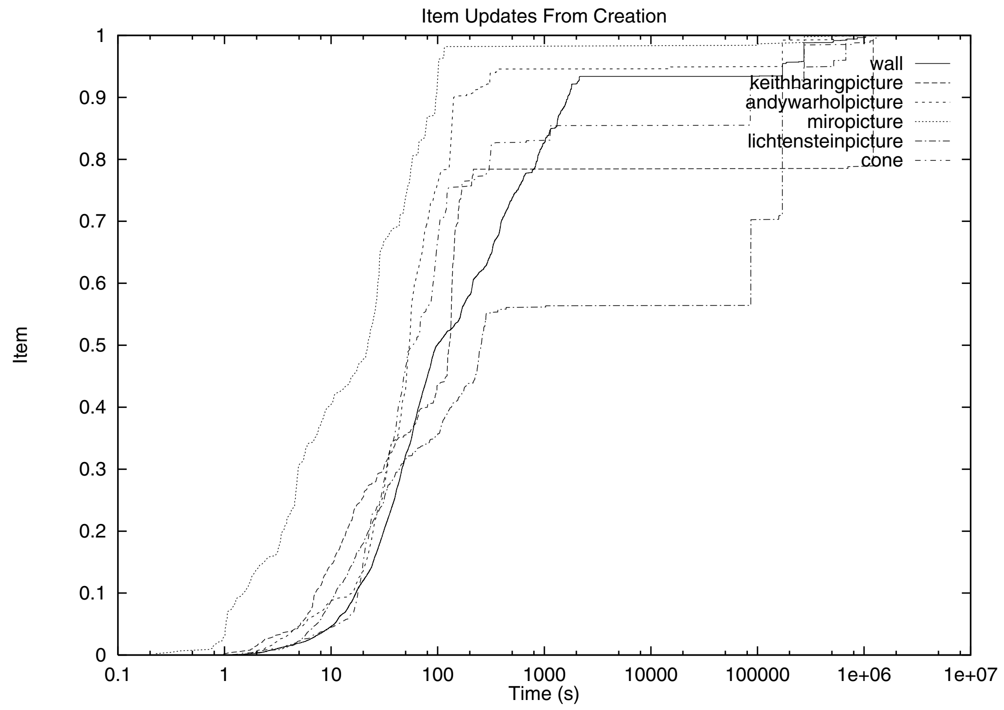


Figure 2-8 Update times from item creation by geometry

A similar pattern is shown in Figure 2-8, which shows item updates from creation by geometry. The Miro pictures were hardly updated at all after the session in which they were created, as most were deleted, while the cone geometries which were left in the world after the session in which they were added continued to be edited in subsequent sessions – with 45% of their updates occurring in later sessions.

As the experiment focused on manipulating items and embodiments, it is unsurprising that 100% of updates were to entities. Other item types were created, deleted and accessed on a read only basis, but only entities were changed during their lifetime. It is easy to imagine other applications may have different, mixed characteristics for example attributes representing virtual bank accounts changing infrequently, but entities representing positions changing often.

2.7 Observations and Qualitative Results

While the detailed analysis of the recorded experiments aimed to reveal the details of change in persistent virtual environments, the questionnaires were designed to reveal the user issues facing persistent virtual environments. In particular user responses to the on-line editing interface are examined in section 2.7.1, the issues involved in providing content management facilities are discussed in section 2.7.2. and the ability of a persistent, plastic environment to increase the user's sense of immersion is examined in section 2.7.3. Example footage of the experiments is shown in Video Figure 2.

2.7.1 Usability

The questionnaire usability results, summarised in Table 2-1, showed that the experiments proved very challenging. Although most users rated the overall ease of use as average, nearly half of the subjects found resizing an artefact hard and a significant proportion also found moving and rotating artefacts hard. Only the addition, selection and deletion of artefacts were easy for the majority of subjects. This is not overly surprising, as working with artefacts in CVEs has been shown to be difficult in the past (Hindmarsh et. al. 2000). This difficulty was increased by the short time the subjects had to learn the system - an unfortunate consequence of the time available and meant that the subjects were still learning to use the system as they carried out the task. Several features were requested by subjects, such as the ability to move objects along the Z-axis by moving the mouse forward and backward and the ability to zoom and pan the viewpoint. While some of these features could have improved a human viewpoint client, many were CAD like features which would have required a client application radically different to a standard CVE user client. The combination of the difficulty experienced by users and the desire for CAD like features suggests that large scale on-line development should probably be carried out using a specialised client. However, by asking our subjects to undertake such a challenging task using standard user clients, a great deal of experience with ad hoc, human perspective manipulation of a persistent virtual world in a short period of time was gained.

Question	**Very Easy**	**Easy**	**Average**	**Hard**	**Very Hard**
How easy was it to add an artefact?	37%	63%			
How easy was it to select an artefact?	16%	37%	37%	10%	
How easy was it to move an artefact?	5%	26%	37%	21%	19%
How easy was it to delete an artefact?	37%	53%	10%		
How easy was it to resize an artefact?	5%	16%	27%	47%	
How easy was it to rotate an artefact?		10%	31%	26%	21%
How easy was it to see what other users were doing?	10%	42%	47%		
Generally how easy to use was the application?		35%	55%	10%	

Table 2-1 Summary of questionnaire usability responses

2.7.2 Content Management

A number of interesting content management issues were highlighted in the experiment as a result of the intentional exclusion of content management facilities from the experimental platform. Because both space and objects were in infinite supply the world tended to get cluttered with objects, which a user had tried to position, but had failed. Several users described the world "getting more cluttered and messy", containing "a lot of random objects which didn't contribute much to the world" and "lots of unfinished projects".

In many cases it was easier to create a new object, which would appear in its normal orientation at ground level, than to move and rotate an existing object on the ground. This was especially noticeable in the case of objects which had been rotated and which many users had trouble rotating back. The discarded objects would simply be left in the world and the users would move on to a new empty space to create new objects. One user commented on this, suggesting that limiting the number of objects a user could add, or the number of operations which could be performed would result in a more ordered world. This problem highlights the importance of mechanisms to manage world evolution rather than just providing facilities for change and persistent storage of changes. Processes should exist which can identify and garbage collect discarded objects.

2.7.3 Immersion

Despite the lack of content management facilities and the difficulty experienced creating the world, some subjects were very enthusiastic about the world progressing from its initial barren state to a highly populated world with pockets of emerging order. The world was described as "evolving slowly", "growing and developing", "beginning to take shape" and "moving from a more disordered state at the beginning to a state where sections were becoming ordered".

Users seemed to be influenced by the state of the world when they entered it. When initially bare, users tried to create an ordered world; when they failed, subsequent users were presented with chaos on entering the world and so made less effort to order the world. After a group of very able users created an ordered network of rooms, subsequent users made much more effort to create something recognizable. The example provided by the rooms showed subsequent users what was possible and provided motivation, whereas users entering a totally chaotic world seemed to think that chaos was all that was possible. One subject picked up on this, saying, "It appears that successive visitors have learnt from previous visitors, by copying or adding to what they constructed."

As there was only a small palette of objects to build the world from, towards the end of the experiments, users seemed to be more inventive with the objects to create something that stood out in the increasingly populated space. The "donut" sculpture shown in Figure 2-9 was created in the final week of the experiments, whereas in the proceeding weeks the creation of recognizable rooms and exhibits was the major challenge. One subject commented on this, noticing "people getting more imaginative and creative with the objects".

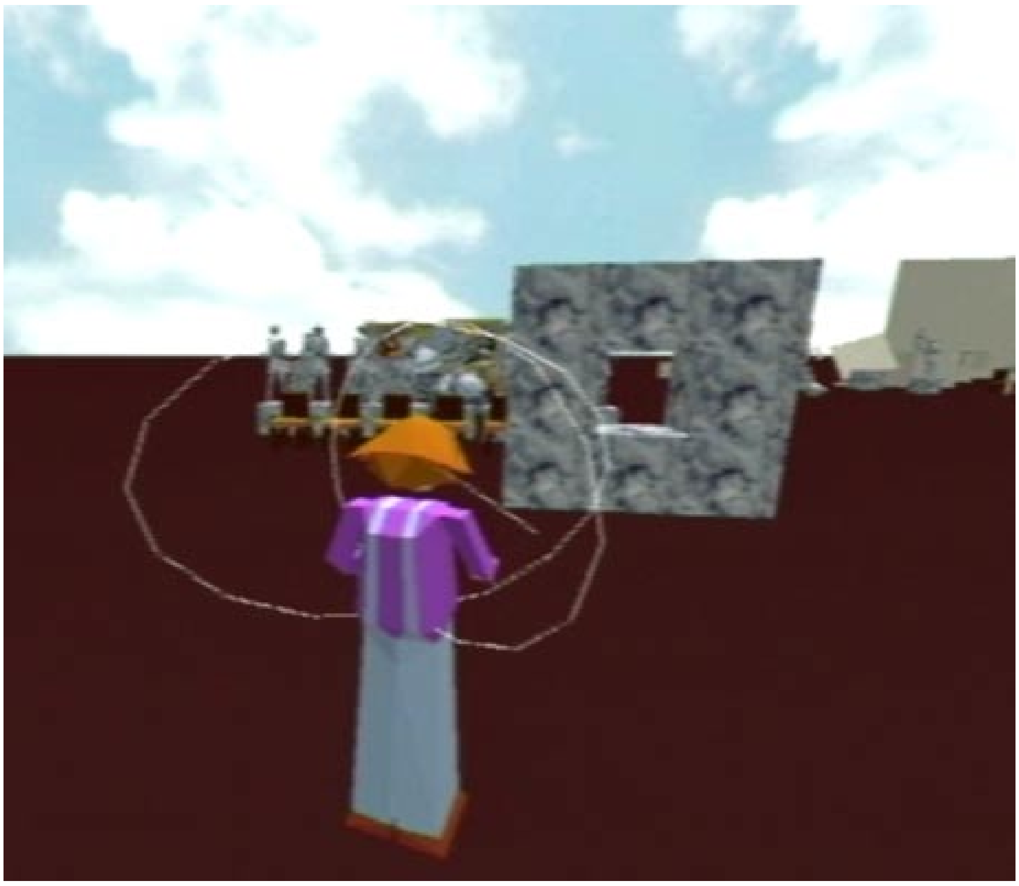


Figure 2-9 The "donut" sculpture created towards the end of the experiments

The existing state of the world provided both an example to subsequent users and a challenge to be bettered. The entire spectrum of collaboration was experienced. Some groups actively disrupted each other by deleting objects the others had created. Some groups ignored each other and worked entirely independently. Some users gave each other tips while working separately, while in other cases one user would help another place objects by watching from a different perspective and giving directions. This was especially useful in the placement of walls where the wall being held could take up most of the view and make it difficult to see where the object was positioned. Spoken directions also helped judge depth, which is normally difficult without stereo displays due to the lack of parallax.

2.7.4 Questionnaire Results

Question	Yes	No
Did you get lost?	33%	66%
Were you aware of other users?	100%	
Did you talk to other users?	90%	10%
Did you work together with other users?	37%	63%
Did you try to finish modifying one object before moving on to another?	83%	17%
Did you try to complete one area before moving on to another?	70%	30%
Did you leave any objects and plan to return to modify them again later?	41%	59%
When you returned to the world was it as you expected?	56%	44%
Was the scenario sufficiently clear?	94%	6%

	Useful	Average	Useless
How useful was sharing the world with other users?	31%	53%	16%

	None	Very Little	Little	Some	A Lot
Previous experience had you had with VR or 1st person games?		7%	7%	47%	39%

Table 2-2 Summary of closed questions

2.8 User Perceptions

The use of both logging and questionnaires to gather results from the experiments allowed the two sets of results to be compared with each other, effectively allowing a comparison between the actual events which took place in the virtual world and the users' perceptions or interpretations of them. In initial pilot trials of the experiments many users thought that others had updated their objects between their two visits when in fact they had simply become lost and could not find their own objects, so in the final experiments users were asked whether they thought they had changed objects they did not create and whether they thought others had changed their objects. The questionnaire responses are compared to analysis of the logs in Table 2-3. 33% of the users contradicted the analysis, with most of the errors perceiving more change than took place. This discrepancy suggests that a mechanism for viewing the changes to the world which have occurred since a user's last visit might be useful.

Actual updates to others	Perceived updates to others	Actual updates by others	Perceived updates by others
0	No	0	No
612	Yes	343	Yes
0	Yes	0	Yes
0	Yes	0	Yes
0	No	0	Yes
272	Yes	2	No
1175	No	20	Yes
0	Yes	0	No
49	Yes	1506	Yes
64	Yes	0	Yes
20	No	106	Yes
31	Yes	39	Yes
688	Yes	132	Yes
4	No	0	No
31	Yes	0	No

Table 2-3 Comparison of actual and perceived updates

2.9 Conclusions

The experiments revealed a number of important technical issues relevant to the provision of continuously persistent virtual environments outlined in chapter 1. These are presented below:

2.9.1 Content Evolution

The experiments highlighted two main challenges facing content evolution. Section 2.9.1.1 discusses the first, user interfaces for on-line editing, while the technical provision of persistence is tackled in section 2.9.1.2.

2.9.1.1 On-Line Editing Interfaces

The difficulty reported by many of the experimental subjects suggests that the development of interfaces for on-line world modification presents a major challenge. It was hypothesised that ad hoc, human scale manipulation was identified as important for rapidly reconfiguring the world for new activities. However, the difficulties experienced by the experimental subjects suggest that using a human scale interface for the complete development of a persistent virtual world is over-ambitious and that specialist tools should be used for the macro engineering of the environment. In the case of specialist tools the provision for communication between designer and other inhabitants of the world and allowing the designer to be aware of events in the virtual world are challenging user interface issues.

2.9.1.2 Persistence

While the provision of persistence itself is relatively easy, the experiments show that the requirements for persistence by different items in the virtual world vary widely. Although it is possible to use a single mechanism such as the checkpointing used in the experiments for all items in the virtual world a single solution is less than ideal. If checkpoints are made frequently enough that updates to important items are not lost, the checkpointing mechanism will waste a lot of time and space making the state of transient objects such as embodiments and manipulated items persistent. Similarly, if a single logging strategy is used to achieve persistence a great deal of transient updates will be logged along with important changes to items. Dealing with these differing requirements for persistence presents a significant technological challenge.

2.9.2 Application Evolution

The experiments did not focus on application evolution directly because of their limited time scale. However, they did highlight the importance of run-time application evolution through the significant modifications made to the virtual world during the “tour guide” sessions. If a standard user client had been used for the second sessions the updates would have been impossible. In addition, many subjects suggested features they would have liked to have seen in the system. In a virtual world used for an extended period of time these features would have to be provided to avoid users becoming frustrated with the system. The experimental system could support system updates by writing the contents of the virtual world to a checkpoint and terminating. The client and server could then be updated off-line and the new system could restore the previous world state by reading the checkpoint when it starts. While this would allow application evolution it would require downtime while the system is updated and so could not be used where constant availability is required.

2.9.3 Constant Availability

The mechanism for automated application termination and spawning used in the experiments provided a simple and elegant technical solution to providing constant availability, but this was largely due to the absence of facilities to provide application evolution. If application evolution is required alongside constant availability the virtual environment system requires facilities for the loading and unloading of code modules at runtime, protocols defining entry points in the loaded code and mechanisms for dealing with different versions of code modules and determining when modules are no longer needed.

2.9.4 Content Management

As well as subjects commenting on the chaos and clutter in some parts of the world during the experiments, an important observation gained from the questionnaire results is that users expect artefacts to behave in different ways. Specifically users thought that a wall should be more difficult to move than a picture on it and that a large landmark should be more difficult to change than a piece of furniture. The first observation suggests that there are different roles for each artefact. As well as the need for a mechanism for content management there should be a way of treating artefacts differently with regard to content management. A wall should be able to be changed in different ways to a picture. The second observation suggests that the roles of artefacts change dynamically so that the different content management policies should be able to change dynamically. An artefact which was initially unimportant should be able to be treated differently once it has become a landmark.

2.9.5 Summary

The conclusions above are summarised in Table 2-4. With the exception of the interfaces for on-line world editing all of the technical issues raised by the experiments relate to the infrastructure of the virtual environment platform – the provision of multiple, dynamic mechanisms for persistence and content management and the provision of mechanisms for dynamic code loading for application evolution in a constantly available environment. Chapter 3 explores the provision of these mechanisms both inside and outside the domain of virtual environment systems. Chapter 4 then presents a novel framework for persistent virtual environment systems which tackles all of the issues presented here.

Goals	**Technical Issues**
Content Evolution	Editing tool interfaces, Differing requirements for item persistence
Application Evolution	Dynamic code loading
Constant Availability	Dynamic code loading
Content Management	Different requirements for item management

Table 2-4 Summary of conclusions

3 Literature Review

The goal of this chapter is to review existing work and systems relevant to the infrastructure technical issues identified in section 2.9. This includes both virtual environment systems which provide any combination of persistence, content management or extensibility facilities, systems which provide these facilities for other applications and mechanisms which provide the facilities themselves. To examine these areas this review is split in to two main sections: systems and mechanisms. The first examines existing collaborative virtual environment platforms, using their support for persistence, world management and extensibility as criteria for evaluation. The second section then looks at mechanisms for persistence, content management and extensibility in turn with each section examining systems which employ these mechanisms.

3.1 Virtual Environment Systems

In the review of virtual environment systems the features identified as important in the introduction are examined. These are:

- Persistence. Does the virtual environment system write its state to stable storage to provide durability and fault tolerance? In some cases these facilities are also used to record activity or allow large environments to be supported.

- Content Management. Does the virtual environment system provide facilities to manage its content? (e.g. to prevent unauthorised tampering with its content)

- Extensibility. Does the virtual environment support run-time extensions both in the form of artefact behaviours or low-level/system extensions?

A system designed to support continuously persistent virtual environments must support all of these features.

3.1.1 LambdaMOO

Multi-User Dungeons or Multi-User Dimensions (MUDs) were the first networked virtual environment systems and also the first continuously persistent virtual environment systems. As they were used around the clock and were freely accessible across the internet, MUDs needed to provide all the facilities required by continuously available virtual environment systems; persistence, world management and runtime extensibility mechanisms. The system examined in particular is the MUD, Object Oriented or MOO system developed at Xerox PARC by Pavel Curtis (Curtis, 1997) simply because it is better documented than most other MUD systems. MUDs use a simple client server architecture. Text clients like Telnet send strings to the server which are parsed and converted into operations carried out on a database, with the textual results returned to the client. By returning descriptions as the results of operations modelled on physical activities and by maintaining a symbolic representation of space in the database, a text based virtual environment is created. The LambdaMOO database contains the objects which make up the virtual reality and the programs which operate on that data (termed verbs). Objects are made up of named properties which may be objects or primitive types and verbs which are associated with an object and implement the operations on an object. Objects also reference parent and children objects which are used to define a generalisation/specification hierarchy similar to a class hierarchy in an object oriented language.

3.1.1.1 Extensibility

The key to LambdaMOOs extensibility mechanism is that properties, verbs, parents and children can be added, removed or changed at run time. The alteration of the parent and children relationship is equivalent to run-time modification of a class hierarchy, which can be achieved in object oriented languages using delegation. This allows new types to be introduced at run-time. A new type of room may be added to the virtual reality which specialises a generic room already existing in the virtual environment. The ability to add and remove verbs and properties at run-time is a facility not normally present in object oriented languages. It allows the behaviour of objects and all their children to be changed while they are being used. If an object in the virtual world is not operating satisfactorily, its behaviour can be changed without the system being stopped. This extensibility allows a LambdaMOO server to be started with minimal contents and new objects and behaviours to be created while the system is running (the reason that world building applications are one of the most popular uses of LambdaMOO compared to earlier MUD systems)

3.1.1.2 Content Management

LambdaMOO uses a permissions system for world management which is similar to the UNIX file permission system (Garfinkel and Spafford, 1996). LamdaMOO defines a number of standard properties on objects which include the owner (who has privileged access to the object) and a set of standard non owner permission properties, which include bits specifying whether the object is publicly readable, writable or “fertile”. Publicly fertile objects can be specified as parent objects by users other than the object’s owner. Properties of objects have a similar set of non-owner permissions, but have a “change ownership in descendants” bit in place of the fertile bit. This flag specifies whether the owner of a child object becomes the owner of inherited properties in that object. This flag is used to ensure the correct operation of verbs which run with the permissions of their author, so if a property and a verb to operate it are added to an object, the author must remain the owner of the property in child objects to ensure the verb works on the child object. Verbs also have a set of UNIX like permissions for non-owners allowing them to be publicly readable (which permits the verb’s code to be viewed) writable (which allows the verb’s code to be changed) and executable (which allows the verb to be run). In addition to this management model based on object ownership with varying degrees of public access, LambdaMOO defines a number of administration roles such as programmers and wizards who have privileged access to objects and facilities in the server. With these mechanisms users of the virtual environment can be given the ability to change and extend parts of the environment, but not others. The objects in an environment and the types of objects can evolve, but important landmarks can be protected from virtual vandalism.

3.1.1.3 Persistence

While the extensibility and management facilities provided by LambdaMOO are highly developed, its persistence mechanism is relatively simple. The contents of the LamdaMOO database are kept in the server's memory while the system runs and are periodically checkpointed to a file on disk while the server is running and when it is shut down. In addition there are standard system calls which can generate checkpoints from anywhere in a verb. The persistence is transparent to users and developers of the system who can simply assume that everything in the database is made persistent. Because of the periodic checkpointing behaviour of the system it is possible for information to be lost if the system fails between checkpoints. While this is sufficient for the mainly recreational uses of LambdaMOO, it may be insufficient for commercial uses of a continuously persistent virtual world.

3.1.2 Habitat

The Habitat system developed by Lucasfilm (Morningstar and Farmer, 1990) was an ambitious attempt to create an open ended, large-scale multi-user virtual environment. The system provided real time animated 2D graphical world made up of 20,000 regions in which a population of 15,000 users could communicate, get married, start businesses, found religions and wage wars. Like many uses of the LambdaMOO system the goal was not to provide a game with fixed objectives, but an evolving community driven by the desires of its members. Architecturally, Habitat resembled LambdaMOO, consisting of a central server containing the persistent world state which was accessed via clients across a network. Unlike LambdaMOO, in Habitat a sub-set of the objects which make up the world are cached at the client which generates frames of animation using the position and appearance they contain. Whereas MUDs generate textual descriptions of the world at the server and send them to the client where they are presented, it is clearly not feasible for the server to generate graphical representations several times a second for every client and transmit them across the network. In addition to saving bandwidth this architecture ameliorates network round trip latency from many interactions. In most cases the client can interact with the local proxy objects and immediately see the results, rather than waiting for the request to propagate to the server and have the results returned – a wait which is acceptable in relatively slow paced text based virtual environment, but not in interactive graphical systems.

3.1.2.1 Extensibility

Although Habitat has a strongly object oriented architecture it does not possess the run-time extensibility features present in LambdaMOO. While world development in Habitat was driven by user desires it was implemented by world designers. This limitation is identified by Morningstar and Farmer who note that “The ability to add new classes of objects over time is crucial if the system is to be able to evolve”.

3.1.2.2 Content Management

Because Habitat does not allow potentially malevolent unknown code to be run within the system, its infrastructure requirements for security and world management are lower than LambdaMOOs – if an important object must not be moved its behaviour can be hard coded to disallow movement. Morningstar and Farmer describe situations where users managed to do something they should not due to unforeseen circumstances or bugs in the system. On these occasions they advocate resolving the situation within the bounds of the reality the system is portraying – when a bug resulted in users becoming extremely rich their money was used to subsidise treasure hunts and when a user managed to get an item they should not it was ransomed back to the system administrators. Both examples display an approach of negotiating world management rather than trying to enforce it at a system level. Habitat’s infrastructure does make sure that it assumes nothing about the user client, however. It validates all communication to ensure that players could not cheat by altering their client application, something that Morningstar and Farmer report was attempted frequently.

3.1.2.3 Persistence

Like LambdaMOO, Habitat implements persistence on its central server both to provide durability in the face of failure and to allow the world to grow beyond the limits of the server's memory. Unused areas of the world are kept on disk until they are required by a user.

3.1.3 VWorlds

VWorlds (Vellon et. al., 2000) is another graphical virtual environment system which can be seen as a direct descendent of MUD systems like LambdaMOO. VWorlds aims to provide a distributed, persistent, secure platform in which end users can develop graphical virtual environments. Like LambdaMOO, VWorlds uses a dynamic object model architecture to enable users to extend and evolve the virtual environment without shutting it down or users needing to use compilers to develop the system. The graphical nature of VWorlds demands a slightly different architecture to LambdaMOO. While it is based on the same single central server and multiple clients, VWorlds clients cache copies of the objects in the world. These proxy objects allow users to interact with objects and change their viewpoint without experiencing high network latencies. While network round trip latencies are acceptable for the low speed interaction of text based virtual environments, they are unacceptable in interactive graphical virtual environments.

3.1.3.1 Extensibility

VWorlds achieves run-time extensibility using the same object inheritance mechanism as LambdaMOO. Basic Thing, Room, Artefact, Avatar and Portal objects are defined from which the contents of the virtual world are derived. When a method or property of an object is referred to, it is first looked for at the object and if not found it is looked for on the object's "exemplar" and then exemplar's exemplar and so on. Adding a property or method to the object is achieved by adding an entry to the objects property or method map which is searched when a property or method is needed. In addition to the inheritance hierarchy, all Vworlds Things have a container object and any number of contained objects which specify the aggregation hierarchy of the world.

3.1.3.2 Content Management

Vworlds' management facilities are also based on LambdaMOO's mechanisms. Methods and objects have owners and public permission settings and methods are run with the permissions of their authors to ensure that methods can only change the objects and properties to which the author has access. Methods on exemplars are able to access any property they create to solve the problem of inherited methods needing to access inherited properties owned by the derived object.

3.1.3.3 Persistence

VWorlds achieves persistence via a combination of logging and object serialisation at the server. While the server is running, all changes to object properties and object structure are written to a sequential file. When this file gets too big VWorlds can checkpoint its state by serialising all of its objects to disk and starting a new log file. While this mechanism provides more durability than the LambdaMOO periodic checkpoint approach to persistence, only server objects are made persistent. This allows users to interact with a proxy object and immediately see the results, but for an error to occur before the update has propagated to the server and then to the server log. When the system is restarted the change which appeared to the user to have taken place would be lost. Some customisation of persistence is provided by the ability to mark properties as transient – an indicator to the system that changes to the object should not be logged.

3.1.4 CAVERNSoft

CAVERNSoft (Leigh et. al. 1996) is a Collaborative Virtual Environment platform designed to enable rapid development of applications on the CAVE Research Network (CAVERN), a network of academic institutions equipped with CAVEs (Cruz-Neira et. al., 1993), ImmersaDesks (Czernuszenko et. al., 1997) and high-performance computing resources. As such CAVERNSoft comes from a 3D scientific visualisation background which assumes high bandwidth and high performance systems rather than the commodity hardware assumptions which MUDs and Habitat make. The architecture of the CAVERNSoft platform is based on a Distributed Shared Memory model. Each client or server using CAVERNSoft creates an Information Request Broker (IRB) which manages network access and a database of storage locations identified by keys. To share data in the system a local key is linked to a remote key. Depending on the properties of the link the two keys are synchronised in some way (the local key becoming a cache of the remote copy) and then updates are propagated between the linked keys to keep the copies synchronised. Because a client may share data with multiple servers, the CAVERNSoft platform is potentially more scalable than MUDs, Habitat or VWorlds which rely on a single server architecture.

3.1.4.1 Extensibility

While CAVERNSoft incorporates no explicit facilities for run-time extension or reconfiguration, the architecture is flexible enough to support a number of network configurations. IRB clients with replicated databases can share updates via a multicast group, connect to a shared central server, support a shared distributed database with peer to peer updates or connect to a collection of servers as required. This flexibility could potentially be used to support different distribution mechanisms for different types of data, while the properties between links provide flexibility in update propagation. By dynamically altering the distribution and link configuration the system could be seen to provide some support for infrastructure evolution and reconfiguration, but only within the set of options hard coded in the platform.

3.1.4.2 Content Management

CAVERNSoft provides no documented infrastructure facilities for security or world management. As a platform developed for use on a network between trusted institutions and a platform with no facilities for run-time extensibility, it is unlikely that such facilities exist.

3.1.4.3 Persistence

Keys in CAVERNSoft may be transient or persistent. Keys are transient by default and their data is discarded when the IRB containing the key is destroyed. A key can be made persistent by performing a commit operation on the data. When the client or server relaunches the data will still be retrievable using the same key identifier. In addition to the ability to make data persistent, CAVERNSoft can make recordings of activity in the virtual world by clients declaring keys which hold recordings of groups of keys. The recording client writes each change seen to the recording key (time stamped with the time the change arrived at the client) along with checkpoints of all the recorded key's state at periodic intervals. The use of the client's time stamp means that the recording is seen from the recording client's point of view, while the periodic checkpoints in the recording allow the recordings to be quickly fast forward or rewound.

3.1.5 DEVA

DEVA (Pettifer et al 2000) is a framework for providing distribution and execution facilities for collaborative virtual environment applications designed to complement the graphics and spatial management facilities provided by the MAVERIK VR kernel (Hubbold et al, forthcoming) to enable the development of complete applications. Logically DEVA uses a client server architecture, although in reality the server is made up of a number of separate server nodes which share the work normally performed by a single server, allowing DEVA to scale further than a traditional client server system. The virtual world in DEVA is made up of a number of environments each containing a number of entities. The scalability and flexibility of DEVA is further enhanced by its architectural support for the distinction between the objective reality of a world and a subjective perception of the world. This is achieved by splitting an entity's processing between an "object" on a server node and a "subject" on each client observing the entity. The object performs processing which must be visible to all observers, such as updating the position of an entity, while subjects perform processing which can be kept locally, such as animating the wings of a moving bird. All observers will see the animation produced by their local subject, but the server does not need to send updates for each frame of animation across the network.

3.1.5.1 Extensibility

DEVA provides rich support for extensibility via the subject-object framework, the ability to develop custom environments and the ability to compose the behaviour of an entity from its own behaviour, inherited behaviour and behaviour enforced and imbued by the environment. This composition of behaviour is achieved using a concept of “components”, a collection of subject and object methods which can be attached or detached from an entity at run-time. Facilities for setting an entity’s position or handling collisions might be provided by an environment component, while the ability to change an entity’s colour might be provided by a component specific to a type of entity. When a method is called on an entity, enforced environment methods are searched first (those which cannot be overridden by the entity), then the entities innate methods and finally the environments “imbued” methods. These facilities allow new environments to be created, potentially inheriting some behaviour from an existing environment, or new entities to be developed, potentially related to other entities, by implementing a single new component. The subject-object framework allows different entities to manage communication between the client and server in different ways allowing per-item infrastructure management.

3.1.5.2 Content Management

DEVA does not provide any specific support for world management or security: however, these facilities could be implemented as components allowing management on the environment or entity scale. An environment component could enforce access control on its contents, or a specific entity could provide authorisation. The subject-object split allows authorisation decisions to take place on the client in trusted networks or on the server in situations where client applications cannot be trusted.

3.1.5.3 Persistence

DEVA supports persistence only in the sense that the server continues to process entities in the absence of clients, but it does not write entity state to stable storage and so is not durable in the face of failures or persistent in the sense that it contains data which outlives any single application.

3.2 Summary

The review of existing virtual environment systems, summarised in Table 3-1, reveals nearly every combination of support for extensibility, persistence and reconfigurability. While this seems initially surprising, it is because only Habitat, LambdaMOO and VWorlds are designed to support continuously persistent virtual environments and so support all the features needed for that application. Other systems focus on one or more of the facilities and so tend to be more developed in those areas. For example, although DEVA does not support persistence or content management its sophisticated extensibility mechanisms might well prove useful in a future system supporting continuously persistent virtual environments.

System	Extensibility / Reconfigurability	Content Management	Persistence
CAVERNSoft	None	None	Persistent arenas of distributed shared memory and recording of changes
Habitat	None	Hard coded object behaviours	Checkpointing and paging of content
LambdaMOO	Run-time addition of objects, properties and verbs	Permissions on property and verb access	Checkpointing
VWorlds	Run-time addition of objects, properties and verbs	Permissions on property and verb access	Checkpointing and logging
DEVA	Entity behaviour composed from parent, environment and subject object components	None	None

Table 3-1 Comparison of extensibility, content management and persistence facilities of collaborative virtual environment systems

3.3 Persistence

As the work of persistence is the storage and retrieval of data it is unsurprising that the majority of research and development into persistence has taken place in the field of databases. Sections 3.3.1 and 3.3.2 examine the two main approaches to databases; relational and object databases. Section 3.3.3 then looks at serialisation as an alternative approach to persistence, before section 3.3.4 discusses orthogonally persistent object systems which seamlessly provide object database like persistence within standard programming languages.

3.3.1 Relational Databases

The goal of databases is to move data from one consistent state to another in the face of concurrent access and failure. Databases use the concept of transactions to describe a sequence of actions which move the database from one consistent state to another. The database cannot allow a transaction to perform some of its actions, but not all. This is avoided by "rolling back" transactions which have not "committed" in the event of failure. There are a number of mechanisms which can be used to achieve these semantics. The database may log all changes made by a transaction but not update the database until the transaction commits. This deferred update approach requires a no-undo/redo recovery strategy - no intermediate results are written to the database and so need to be undone, but a failure between a commit being written to the log and the updates being written to the database requires transactions being redone. If the database uses an immediate update approach - writing to the database during the transaction it must first flush updates to the log allowing the updates to be undone in the event of failure. If all updates are written immediately, an undo/no-redo recovery strategy can be used, otherwise immediate update requires undo/redo recovery.

Durability is not the only consideration for recovery mechanisms as those which do not allow data to be written to disk before a transaction completes ("no-steal" approaches) or demand that data is flushed to disk ("force" approaches) restrict the operation of a database's page cache. Because of this "steal/no-force" approaches are used by most databases as they impose no restrictions on page cache operation (more information on the interaction between database recovery and caching can be found in Date, 2000).

DBMSs which use logging periodically checkpoint their state either every *s* seconds or every *t* committed transactions to allow logs to be emptied. To avoid delays while a checkpoint is written, most databases use fuzzy checkpointing-continuing processing while the checkpoint is written to disk, but relying on the previous checkpoint for recovery until the write completes.

3.3.2 Object Oriented Databases

Object Oriented DataBases (OODB) attempt to combine the persistence, secondary storage management, concurrency, recovery and ad hoc query facilities of database management systems with object oriented facilities such as support for complex objects, object identity, encapsulation, extensibility and computational completeness (Atkinson et. al. 1989). They avoid the 'impedance mismatch' encountered when developing applications using relational databases where data must be continually mapped between relational and object oriented models. Rather than designing an application and a database to contain the application data and then implementing the application to move data to and from the database, an object oriented database automatically generates database schema from application data types, manages the storage of objects promoted to persistence and garbage collects objects no longer needed.

The storage of types and methods within OODB systems allows a single database to support application development and provide application persistence. O_2's OOPE development environment (Borras et. al. 1989) supports this approach. Similarly the support for classes as objects in Smalltalk (Goldberg, and Robson, 1983) allows its distributions to be seen as pre-populated OODBs.

The support for ad hoc queries, which are allowed to break the rules of encapsulation, initially appears to undermine the object oriented nature of OODBs. However, without it there would be no way of accessing the data in an OODB without writing a program. This would be a severe weakness compared to the rich interactive querying possible with relational approaches. As ad hoc querying only relies on the object's structure for the lifetime of the query, the breach of encapsulation is much less serious than if applications were to rely on an object's structure.

Object oriented databases use many of the same mechanisms for storage as relational databases, relying on transactions and strategies to recover from failures during transactions. In most cases OODBs use a layered architecture (Keller, 1998) with upper layers dealing with complex objects which are converted into operations on tuples handled by the lower layers. In some cases the lower layer is implemented by a relational database as in the case of POET (Thelen and Beckert, 1997), or disk management system as with O_2 (Deux et. al. 1990). Static analysis of the application methods in the OODB can allow increased concurrency and reduced locking of objects involved in transactions.

3.3.3 Serialisation

Object serialisation is the process of taking a reference to an object and producing a series of bytes which represent the object and all objects reachable from it. The sequence of bytes can then be transmitted across a network to another machine or written to a file or database Binary Large Object (BLOb). When the serialisation is received across the network, or subsequently read from storage, new objects for each object serialised are constructed. The new objects have the same type and state as the original objects, but a different identity. If an application serialises an object to disk and then reads the object back it ends up with two identical objects. In some cases this can be useful, for example the same mechanism can be used for serialisation and object cloning, but in the case of using serialisation for persistence, the side effects are generally unwanted. If two objects which both reference a third object are serialised and subsequently read, the third object will be included in both serialisations and so a new object equal to the third object will be created when each serialisation is read. Rather than sharing a reference to an object the objects created from the serialisation each have their own copy of the third object which can diverge. The semantics of the system created from the serialisation are different to the semantics of the system which was serialised. Systems using serialisation can also suffer from the “big inhale” problem: if the graph of references between objects is highly connected then a large number of objects are included in any serialisation. An application wishing to load on a small subset of data has to wait until the whole serialisation has been read before any data can be accessed. Despite these problems, serialisation is an attractive mechanism to use for persistence as it is easy to use, can be automatically driven from class definitions in languages like Java (Gosling et. al., 1996) and is also useful for moving objects between nodes, as shown by its use as the foundation of Java RMI.

3.3.4 Orthogonally Persistent Object Systems

Orthogonally Persistent Object Systems (Atkinson and Morrison, 1995) allow the creation and manipulation of data of any type in a way that is independent of its lifetime. Data used as an intermediate result which is discarded as soon as a calculation is complete and data which persists for years in a company database are created in identical ways and manipulated using the same standard programming language constructs. In addition to the principles of persistence independence and data type orthogonality described above, orthogonally persistent systems aim to separate the identification of persistent objects from the type system. This is achieved by following references from persistent roots, objects which are explicitly named and promoted to persistence. All objects which can be reached by following references from these persistent roots, and in some cases the executable code needed to interpret the objects, are promoted to persistence. Conversely, objects which become unreachable from a root are removed from the persistent store via garbage collection. Orthogonal persistence is usually provided by heavily modifying the run-time environment of a standard programming language. The modified run-time is responsible for obtaining the type information from programs and accumulating it as a schema to allow the storage of objects, the storage of any values which future programs may use, identifying which values should be made persistent and which persistent values can be garbage collected and arranging to incrementally load persistent data as needed. Orthogonal persistence greatly simplifies the development of Persistent Application Systems (PASs) compared to the use of relational databases and programming languages. Instead of attempting to maintain consistent database and programming models of a system a single programming model is used with data made persistent as required. Compared to serialisation, orthogonal persistence does not suffer the “big inhale” problem or the loss of object identity. The main problems with orthogonal persistence are its limited commercial availability and the performance overheads imposed by the modified runtime.

3.3.5 Summary

The database field provides many mechanisms for durability which have been used in persistent virtual environment systems or could be used in future systems. However, the central role that transactions play in these mechanisms can make the techniques difficult to use for virtual environment systems. Waiting until updates are committed to stable storage before presenting them to users in a virtual environment system increases latency and greatly impacts interactivity, while rolling back updates makes little sense in a real time system where updates have already been experienced. In general, these techniques are useful in virtual environments when used more optimistically (for example if logs and checkpoints are used for recovery, but the system does not promise to restore all updates experienced by the user).

The dual use of serialisation as a object migration and persistence mechanism makes it a very attractive mechanism for use in virtual environment systems, which can either reuse existing serialisation mechanisms to add persistence, or be designed from the outset with this dual use in mind.

The transparent persistence provided by orthogonally persistent languages is less attractive for collaborative virtual environment systems where the transparency of other infrastructure issues tends to be rare.

3.4 Extensibility/Reconfigurability

Extensibility is often found in systems through the support of plugins which provide alternative implementations for a well defined interface (e.g. web browser plugins providing alternative rendering methods, or graphical editor plugins providing different filter effects). Middleware frameworks go further by providing facilities for applications to define their own interfaces and to discover the interfaces implemented by other components at run-time. Reconfigurable systems also provide facilities for replacing and removing implementations at run-time allowing the system to be effectively upgraded at run-time. This section examines a number of systems which provide mechanisms for extensibility and reconfigurability at both the application and infrastructure level.

3.4.1 Review Criteria

In the review of systems supporting extensibility and/or reconfigurability the important criteria to examine are:

- The level at which the extensibility and/or reconfigurability operates. In particular, does the system support application level run-time extensibility, or infrastructure level extensibility? (for example allowing the introduction of new networking protocols or consistency mechanisms).

- The support for distribution of extensions, for example can the system support extensions which span multiple nodes in the network and cooperate via network communication?

- The generality of extensions, for example does the system only allow tightly defined extensions, such as the replacement of a particular service or mechanism, or is the extension mechanism more general?

The reason these criteria are applied is that a continuously persistent collaborative virtual environment system needs to support all three; general, distributed and infrastructure level extensibility.

3.4.2 Components

Components (Orfali et. al. 1996) are reusable pieces of compiled software which can be combined with other components to produce complete applications. As such, components provide a great deal of reconfigurability – a good component must be able to be used in one application and then reconfigured and combined with different components to produce a different application. Component reconfigurability is supported by visual builder tools and high-level scripting languages designed to make combining and reconfiguring components easy. These tools are in turn supported by standard component functionality such as support for introspection (which allows builder tools to analyse how components work) and support for events (which provide component communication and properties which allow component customisation). Typical reusable components might provide customisable Graphical User Interface (GUI) controls such as sliders or database viewers which can be visually dragged on to an application dialog and their events visually connected to application methods. In addition to supporting rapid application development, components can allow run-time customisation of applications using the same graphical development tools. This thesis document is a component which can embed other components, be viewed in a graphical editor as a component with properties such as a DefaultTabStep size and can be manipulated using scripts. Distributed components are components which communicate with other components across networks using a component infrastructure. The component infrastructure provides an object bus which allows components to call methods on remote components, exchange metadata, discover each other and provides services such as persistence and transaction management. The facilities provided by the infrastructure allow applications to be written as enterprise components which contain just the business logic required for an application and rely on the component infrastructure for implementation services. While components may prove useful to extend or reconfigure virtual environment applications, the overheads imposed by request brokering generally make component architectures unsuitable for real-time communication between nodes in a collaborative virtual environment system.

3.4.2.1 CORBA

The Common Object Request Broker Architecture (CORBA) is an open component infrastructure defined by the Object Management Group (OMG). CORBA components are white box objects in that they support the classical object concepts of inheritance, identity and encapsulation. The components are described using the language independent CORBA interface definition language (IDL) allowing components to be implemented in any language. CORBA Object Request Brokers allow static method invocations which use stubs (similar to those used by Remote Procedure Calls (RPCs)) like or dynamic method invocations which use automatically generated meta-information to allow objects to discover each other at run time. CORBA provides many services including Life Cycle, Naming, Persistence, Versioning and Concurrency Control services and provides mechanisms for different Object Request Brokers (ORBs) to communicate over the internet.

3.4.2.2 TAO

TAO is a CORBA ORB implementation designed for real-time high performance applications (O'Ryan 2000). It supports infrastructure level extensibility and reconfigurability through its support for pluggable protocols for inter-ORB communication. These allow networking protocols more suited to real-time operation than to be used instead of or alongside the Transport Control Protocol (TCP) used by the General Inter-ORB Protocol. In addition to presenting the TAO ORB and the advantages of pluggable protocols O'Ryan presents the problems inherent in allowing pluggable protocols and suggested solutions.

3.4.2.3 COM/DCOM

The Component Object Model (COM) (Horstmann and Kirtland, 1997) is the component architecture at the centre of Microsoft's Windows operating system. Beginning as an evolution of the Windows Object Linking and Embedding mechanisms, COM initially provided an architecture for binary interoperability between components written in different languages and then was extended with facilities for distribution to become Distributed COM (DCOM). Like CORBA, COM is based on components implementing interfaces. Unlike CORBA, COM components are black box components not supporting inheritance. Reuse in COM is achieved by aggregating components within components which support the encapsulated components. Calls to the interfaces are then delegated to the encapsulated components. This mechanism avoids some problems of multiple inheritance, which can be simulated by encapsulating multiple components and potentially provides more flexibility through the run-time configuration of delegates. COM supports run-time discovery of component behaviours through a standard IUnknown interface which can be queried to find other interfaces supported by the component. The same interface provides reference-counting facilities to support garbage collection.

3.4.2.4 JavaBeans/Enterprise JavaBeans

Rather than allowing the binary interworking of components written in different programming languages, JavaBeans are Java components which can be used on any system that can run a Java Virtual Machine (Hamilton, 1997). JavaBeans separate run-time and design-time code such as customisation wizards allowing fast run-time downloading. Enterprise JavaBeans support middleware application components by providing transaction, security and networking services through Enterprise JavaBean servers which provide the component infrastructure.

3.4.3 Bamboo

Bamboo is a component framework that manages the dynamic loading and unloading of language specific plugins (Watsen, 1998). Although originally developed to enable continuously persistent virtual environments, Bamboo is general enough to support any application which needs dynamic code loading facilities. Each plugin is part of a module which contains the plugin and any resources the plugin requires, such as graphics or audio. Bamboo manages dependencies between modules by making sure that dependent modules are loaded before the modules that use them. Once a plugin is loaded an initFunc entry point is called in which the plugin can initialise and a corresponding exitFunc is called before the plugin is unloaded allowing it to free resources. Once initialised the plugin can either create a thread to perform processing, extend objects or classes, extend the process's execution loop by registering callbacks. Each callback incorporates callback handlers before and after the function call allowing further callbacks to be registered with the callback's callback handlers. This recursive behaviour means that a function can always be inserted before or after an existing function. Bamboo uses runtime linking and function calls in a single process to provide maximum efficiency, however this means that a more heavyweight framework like CORBA, DCOM or

Enterprise JavaBeans must be used with Bamboo if remote method calls are required.

3.4.4 Bayou

Bayou is a system which supports applications requiring replicated data in widely distributed or imperfectly connected networks (Terry et. al. 1998). Whereas strongly consistent replicated systems make generally make replication transparent, weakly consistent systems must deal with varying degrees of consistency and conflicting updates which cannot be made transparent. In order to cope with this situation and to acknowledge that different applications have different requirements, Bayou provides flexibility and reconfigurability by giving applications control over replication choices and conflict detection and resolution. To allow conflict resolution, a Bayou write operation consists of a nominal update, a dependency check and expected results and application code. The nominal update is the changes to be made if there are no conflicts. The dependency check is an application defined query which should return the expected results when there are no conflicts. The application code travels with the write and is executed to resolve detected conflicts. For example the dependency check for a calendar application might check that no appointments in the database being updated overlap a meeting being added and the conflict resolution code might move the new meeting to avoid the overlap. Bayou uses an anti-entropy algorithm for update propagation (Petersen et al. 1997). The sending replica obtains a list of updates known by the receiver, then sends any updates that it knows, but the receiver does not. To avoid replicas keeping all updates, a primary replica assigns a sequence to writes which propagates back to other replicas which can then commit and discard these globally ordered writes. This may result in state needing to be transferred if a replica does not have writes, but a sender has committed and discarded them. Bayou allows application policies to decide when to reconcile, with whom, when to truncate the write-log and which server to create a replica from. Reconciliation can be periodic, user triggered or system triggered, when network or CPU bandwidth is available or when logs need to be shortened. In general, frequent updates keep replicas consistent, but use more bandwidth. Partner choices can be made based on reachability, network state, database state, primary status and truncation state. Truncation policies can be co-operative, with some keeping writes allowing others to commit.

3.4.5 Summary

There are a number of trade-offs apparent in the review of systems supporting extensibility, summarised in Table 3-2. The most obvious is that of generality versus level. All component technologies support very general models for extensibility which allow arbitrary new interfaces to be introduced and other components to discover and use those interfaces. However they provide this at an application level. At the infrastructure level extensibility is generally much more limited – systems like Bayou and TAO allow a certain level of customisation or extension of infrastructure mechanisms such as networking protocols or replication strategies, but there are large parts of their infrastructure that must remain static. In some senses, the discussion of infrastructure level extensibility is misleading. If the infrastructure is the framework on which applications rely, it must be static: different applications use the infrastructure in different ways, but the infrastructure is what is leveraged to allow reconfigurability. However, the low level mechanisms which systems like Bayou and TAO make dynamic have historically been part of the infrastructure. In order to make these mechanisms extensible, they must be stripped from the infrastructure, to create micro-infrastructures which contain the minimum of hard-coded bootstrapping knowledge and which load all other mechanisms dynamically. Bamboo is an example of this kind of micro-infrastructure, only containing enough logic to load modules which provide all other services. A potential problem with pushing extensibility so low is that frameworks can become so small and general as to become useless – one of the major attractions of CORBA is that it provides services, discovery and communication facilities to its components. Applications built with Bamboo must implement these services themselves. Virtual environment systems aiming to provide low level extensibility mechanisms must weigh up this trade off carefully. If the framework provides too much as standard, it risks being unable to replace mechanisms at run-time; if it provides too little as standard it risks pushing too much of the burden on to application developers.

System	Extensibility Level	Distributed	Extensibility Generality
CORBA	Application	Yes (ORB Communication)	High (Arbitrary component interfaces)
COM/DCOM	Application	Yes (ORB)	High (Component interfaces)
JavaBeans	Application	Yes	High (Compontent interfaces)
TAO	Infrastructure (Network Protocol)	Yes (Symmetrical protocols)	Low (Network protocols only)
Bayou	Infrastructure (Replication mechanism)	Yes (Mobile reconciliation code)	Medium (Arbitrary reconciliation code, but limited replication choices and only replication/reconcillation decisions extensible)
Bamboo	Infrastructure (All but plugin loading)	No (In process components)	High (All but plugin loading)

Table 3-2 Comparison of extensible systems

3.5 Content Management

Content management is the work of ensuring that only authorised users can view and change information in a system and so is found in nearly every shared or distributed system. In some cases content management is performed by a single, global administrator, while more powerful content management systems allow privileges to be delegated to users in order to spread the work of managing system content. In this review of content management mechanisms, two very different approaches are examined. Section 3.5.1 discusses access control, a conservative approach favoured by databases, while section 3.5.2 looks at a more optimistic approach – update reconciliation – which may be more appropriate to collaborative virtual environments.

3.5.1 Access Control

Access control aims to restrict access to system content to authorised users. Users may be able to perform different actions on different parts of the system content, but the approach is to make all access control decisions in advance of any operations being performed.

3.5.1.1 Relational Databases

Database content management is based on the concept of user accounts which have permission to perform operations and consists of discretionary and mandatory mechanisms (Date, 2000). Discretionary mechanisms are used to grant privileges to users allowing them to perform certain operations on certain data. Mandatory security mechanisms which classify data and users into security classes as a basis for security, for example only allowing users to see information classified at their level of classification or lower. User accounts are created by the database administrator with a password which is kept in an encrypted table in the database and a users identity is proven by providing the password whenever database access is needed. The granting and revoking of privileges follows an access matrix model where subjects, which may be users or programs, make up the rows and objects such as relations or records make up the columns. Each position (*i*,*j*) in the matrix contains the read, write or update privileges which the subject *i* holds on the object *j*. A relations owner is responsible for granting privileges to other users and can optionally allow those users to grant the same privileges – allowing privileges to propagate without requiring the owner to grant every privilege. The system log is annotated with the account responsible for operations allowing the log to be used as an audit trail in the event of tampering.

3.5.1.2 UNIX Permissions

The UNIX operating system (Garfinkel and Spafford, 1996) makes pervasive use of the file metaphor to represent devices, network connections, processes, directories as well as executable and data files. This makes the file system central to the UNIX operating system and the permissions which control access to files central to UNIX content management and security. Permissions on a file consist of 3 groups of 3 bits which indicate whether read, write and execute permissions have been given to the file's owner, group or other users. Read permissions allow users to read the contents of a file, write permission allows the file to be overwritten or its contents modified and execute permissions allow users to run a program or view the contents of a directory. The hierarchical nature of most file systems allows coarse grained or fine grained access control – access to large portions of the file system can be controlled by changing the permissions of directories near the root of the hierarchy, while individual file permissions allow fine grained control.

3.5.1.3 Access Control Lists

Some versions of UNIX also provide access control lists to augment the basic file permission mechanism. The list contains entries which can completely specify the permissions for certain users or groups, deny actions for certain users which they could otherwise perform, or permit certain users to perform actions which the standard permissions would not allow. Using access control lists, access for files can be specified as completely as with relational databases, with each entry in the list specifying the contents of an access matrix cell.

3.5.1.4 SPACE

SPACE is a spatial access control model for virtual environments (Bullock and Benford, 1999). Rather than associating access rights with users or resources, SPACE associates access rights with spatial boundaries within a virtual world. The process of restricting access to an object or information in the model corresponds to moving the object to a space which allows the restricted access. Users wishing to cross a boundary into a restricted area must meet some criteria such as a clearance classification value, match a name or status value or be in possession of a token such as a key or password. SPACE also presents several mechanisms for group access to regions. The combination of group access and the regioning approach allows the collaborative modification of the restricted information, unlike database or UNIX access control where multiple privileged users must make updates sequentially. The configuration of an environment can be represented as an access graph in which regions are the nodes and boundaries the arcs. These access graphs allow the application of standard mathematical techniques to reason about access rights across the entire space, for example the calculation of the minimum clearance needed to move between nodes, or the relative classification of a particular route between two nodes. While the SPACE model provides a very natural metaphor for access control – exploiting users' spatial reasoning – it becomes impractical for fine-grained access control which requires many very small regions and for very large environments in which complete access graphs cannot be constructed.

3.5.1.5 Summary

Table 3-3 compares the approached to access control. While they vary in the granularity in which collections of data or groups of users can be specified, all of the approaches bar SPACE restrict collaboration by serialising access to data. While the problems of waiting for access to locked data can be reduced in a collaborative virtual environment by allowing very fine data granularity, an approach which allowed truly simultaneous access to data is more desirable. This affordance is one of the benefits of update reconciliation, discussed in section 3.5.2.

System	Data Granularity	User Granularity	Collaborative
Relational Databases	Medium to Fine (Relations to Fields)	Fine (Individual Users)	No (Locking and Transactions)
UNIX Permissions	Coarse to Medium (Root directories to Files)	Medium (Owner, Single Group and Other)	No (Sequential Access)
Access Control Lists	Coarse to Medium (Root directories to Files)	Fine (Arbitrary users or groups)	No (Sequential Access)
SPACE	Medium (Regions)	Course (Clearance)	Yes (Group access to a region then collaborative updates)

Table 3-3 Comparison of access control systems

3.5.2 Update Reconciliation

Update reconciliation differs from access control in that it defers the management of changes to content until after the updates have been performed. Users are able to make updates to system content freely and the updates are stored in the system as variants. Updates can subsequently be merged, accepted or discarded in a subsequent reconciliation stage. As update reconciliation deals solely with the creation and merging of variants, it cannot be used to control users' viewing content, only changing it. As such it is only usable where all users can view all content, but only a subset can change the world. Update reconciliation provides a more optimistic approach to managing content in that updates by unknown users can be judged on their merit rather than restricting updates to known users.

3.5.2.1 Versioning

Software versioning systems use the idea of variants to allow multiple inconsistent versions of data to co-exist and later be reconciled. Versions of a file are either revisions, intended to supersede a previous version, or variants intended to co-exist with other versions. Branches resulting from variants in version spaces can be merged by analysing the differences between the versions. Where changes are disjoint, versions can be merged automatically, but user intervention is required where changes overlap. Revision Control System (RCS) (Tichy, 1985) generates a marked up version of the merged data, presenting alternatives in areas where changes overlap. Variants can also be used to manage divergence between views of data manipulated simultaneously by multiple users. Rather than disallowing inconsistency the inconsistent copies can be allowed to become variants.

3.5.2.2 Prospero

The prospero system uses this model of multiple users causing divergence which can later be reconciled (Dourish, 1996). Dourish views data management as the continual divergence and synchronisation of views. Frequent synchronisation results in synchronous style interaction, infrequent synchronisation results in asynchronous applications. Prospero uses an optimistic divergence/synchronisation strategy based on a guarantee/promise model. The system guarantees a certain type of consistency if the client promises to limit actions in some way. For example if a client promises to only add data during updates the system can allow read access during updates and guarantee consistency. The client can break promise, but then the system may not be able to guarantee consistency. When promises are broken the system falls back to syntactic consistency, maintaining both inconsistent views of the world as variants which must be manually reconciled at a later date. Prospero provides both an optimistic approach to content management and a mechanism for infrastructure customisation through different applications making different promises and accepting different guarantees from the system.

3.5.2.3 Summary

Where changes to the content of a virtual world can be made by all users, but must be checked for acceptability, update reconciliation is an attractive option. Rather than ensuring that valid users update items sequentially, update reconciliation allows all users to potentially modify an item concurrently and for the updates to be reconciled later to produce a new shared state. This model fits naturally with the architecture of most virtual environment systems, which maintain a local replica of world state, which is operated on by users becoming momentarily inconsistent with other replicas before being synchronised by sharing updates. By using update reconciliation incompatible updates are allowed, which cause replicas to diverge further to become variants before being eventually merged. Update reconciliation provides an optimistic model for updates which promotes interactivity by reducing the need to wait for locks. This model of content management allows a merit driven approach to world updates in that all users are allowed to modify a world with good updates being accepted regardless of user. By combining update reconciliation with access control, bad updates could result in update privileges being revoked. In this way an optimistic, interactive approach to world evolution can be provided which is nevertheless policed to avoid vandalism. In an application where many users will be unknown, such as in a freely accessible continuously persistent virtual environment this is an important advantage.

3.6 Conclusion

The examination of the persistence, management and extensibility mechanisms provided by collaborative virtual environment systems alongside systems which specialise in such mechanisms show virtual environment systems to be underdeveloped by comparison. Most virtual environment systems have historically been developed with audio-graphical facilities, interactivity or scalability as a focus with support for continuously persistent applications a secondary consideration at most. Those systems which have focused on supporting persistent environments have been mainly commercial systems which have pragmatically developed persistence, management and extensibility facilities as needed to support a particular application.

Against this background there is clearly room for research which looks at providing more general and powerful facilities for persistence in collaborative virtual environments which borrows concepts and techniques from the fields of databases, components, orthogonal persistence and other fields included in this review.

Looking beyond general and powerful facilities for persistent virtual environments towards the vision of a virtual environment system which provides dynamic infrastructure mechanisms shows even more scope for original research. The Bamboo system falls closest to this vision, being developed originally with the idea of continuously persistent virtual environments in mind. However, Bamboo concentrates on extensibility and reconfigurability at the expense of all other considerations. The framework is general and minimal to the point of having no concept of virtual environments, persistence, consistency or anything else other than plugins and loading.

An ideal framework supporting dynamic infrastructure mechanisms alongside traditional virtual environment platform support seems to fall between Bamboo's determined minimalism and the heavyweight infrastructures of component frameworks. As such it could potentially be constructed by building on top of a system like Bamboo, or stripping down a heavyweight framework. The next chapter presents a model which occupies this middle ground – supporting low-level extensibility while providing application developers a metaphor which includes concepts of communication and items which are frequently found in more static virtual environment platforms.

4 Model

This chapter presents the distributed event filter framework which is the main contribution of this thesis. The goals of this framework are to provide customisable, per-item infrastructure mechanisms, which the previous chapters have shown as being desirable in complex virtual environments containing many heterogeneous types of information. The framework also provides facilities for low-level extensibility and reconfigurability of infrastructure mechanisms like consistency and persistence which are needed in environments which must be continually available and so upgraded and extended at runtime. The framework provides a mechanism for managing the content, the per-item processing and the extendible infrastructure in a uniform way, so that the same techniques which are used to define the users allowed to modify or view an object in the virtual world are used to define the persistence and consistency mechanisms which operate on that object.

After outlining the goals of the framework in section 4.1 the chapter goes on to describe the two main components of the framework; "distributed event filters" in section 4.2 and "deep behaviours" in section 4.3. The frameworks potential for optimising the performance of a virtual environment system is discussed in section 4.4 and its scalability is considered in section 4.5. Examples which demonstrate the flexibility of the framework are presented in 4.6 before some conclusions are presented in 4.7.

4.1 Goals

There are a number of desirable characteristics which the framework should exhibit if it is to provide a useful basis for continuously persistent virtual environment systems which support dynamic per-item infrastructure mechanisms.

- Minimal: The framework must be minimal. That is, the static portions of the framework which the dynamic portions rely on must be kept as small as possible.

- Simple: The framework must be conceptually simple. The distributed nature of the infrastructure mechanisms the framework must support are inherently complex, and the dynamism the framework allows increases the complexity. The framework must attempt to reduce this complexity by providing facilities to automate as much as possible and provide a simple conceptual model which is easily understandable.

- General: The framework must be as generally applicable as possible to maximise its usefulness and avoid prohibiting changes which were initially unforeseen. In the case of the thesis, the framework should be able to support at least existing virtual environment architectures and infrastructure mechanisms.

- Scalable: The framework should not reduce the scalability of any system employing it.

- Efficient: The framework should impose as little processing overhead as possible.

While these goals are all desirable, there are clearly several trade-offs present in this cluster of goals. For example presenting users of the framework a conceptually simple model which automates many of the details might prevent optimisations possible if users were presented with the details. Similarly a simple framework which hides details might not be a minimal framework. Where these trade-offs exist a decision must be made on the relative importance of the goals.

While the framework is designed to tackle the issues of extensibility, content management and persistence identified in chapter 1, the most fundamental of these is extensibility. Only a highly extensible and reconfigurable system allows the mechanisms for persistence and content management to be varied dynamically on a per-item basis. It is this per-item reconfigurability which allows the system to support the varied item roles identified in chapter 2. As a result, the majority of this chapter focuses on the provision of extensibility, content management discussed in section 4.3 and persistence in 4.6.

4.2 Distributed Event Filters

The conceptual model chosen for the framework is that of a network of pipes and filters which exist conceptually between and inside the applications which make up the distributed virtual environment system. The events communicated between applications and processed inside applications are explicitly represented as objects which are generated in the methods which make up the Application Programming Interface (API) of the system and are added to event pipes. Each event has a type describing the function of the event and can specify one or more items in the virtual environment system on which they will operate. The event pipe uses these parameters to decide which filters should process the event and then passes the event to each filter in turn. By selecting filters based on their target items the Distributed Event Filter (DEF) framework enables customisable, per-object infrastructure mechanisms. Figure 4-1 shows the main components in the model.

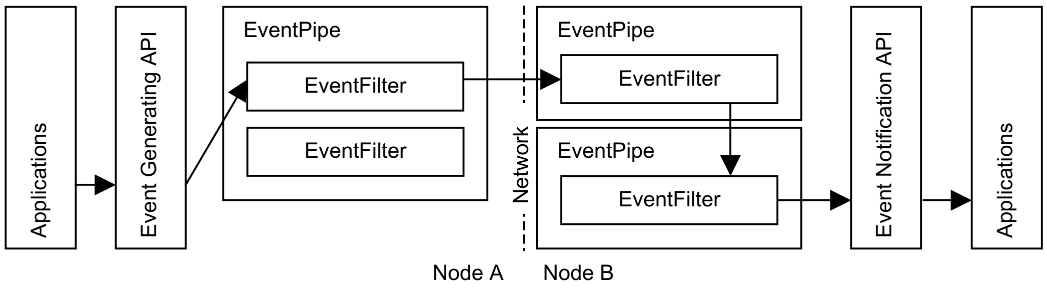


Figure 4-1 Main components of the model

The framework provides simplicity through its simple metaphor and powerful facilities to support the distributed processing of events, and generality and minimalism through assuming only that events must be communicated around the system and processed at specific points.

While many system architectures use a model of pipes and filters (for example UNIX pipes (Ritchie, 1984) or Composition Filters (L. Bergmans and M. Aksit, 2001)), within the field of collaborative virtual environment platforms, the use of a pipes and filters metaphor is novel. There are also a number of important innovations brought to the pipes and filters architecture, which make up the original contribution of this thesis and are discussed in sections 4.2.1 to 4.2.9.

4.2.1 Distributed Infrastructure

The model of events propagating through a network of pipes being processed by a sequence of filters a very natural model for the infrastructure processing which occurs in virtual environment systems. One characteristic of this infrastructure processing is that it crosscuts individual applications and is distributed between applications in the network. The infrastructure is the platform which supports individual virtual environment applications, facilitating communication, persistence, state replication and consistency between replicas. While processes are of their nature centralised, the infrastructure is of its nature distributed. Some processing must be carried out at a client, more at a node in the network and more still at a server, yet the distributed processing units can logically be seen to provide a service through their orchestrated operation.

A network of pipes and filters supports this paradigm very naturally. The implementation of an infrastructure mechanism consists of the implementation of each distributed processing unit as a filter and the insertion of those processing units at appropriate points in the network, within several applications. The mechanism is both part of many applications and part of none, existing separately to the application as a set of filters which can be added to a running system, removed or replaced with a new implementation as required.

While some infrastructure processing does not benefit from being in a sequence of filters and could equally well be implemented via call backs, positioning this processing in an event pipe allows other filters to be placed in specific positions relative to the processing.

4.2.2 Dynamic Routing

It is in its support for dynamic routing that the DEF framework achieves greater flexibility than many other extensible virtual environment systems. Although some MUDs provide verbs on the server (Curtis, 1997), there is no run-time mechanism for extending or changing the way messages reach the server or propagate through it. These previous systems follow the Factory Method pattern (Gamma et al, 1995), providing points at which processing can take place before routing events to the next stage. The object oriented mechanism of delegation using agents (Muller, 1997) provides a way to decide on message flow at runtime using standard object oriented languages. However, this mechanism requires every client to rely on an agent to decide on message flow – the client calls a method on the agent, which chooses which server to forward the message to. Figure 4-2 shows the general model for dynamic routing in object oriented systems.

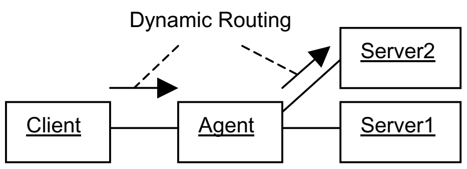


Figure 4-2 Object oriented dynamic routing

In situations where dynamic routing may be required in the future, the agent must still be present to allow for the future variability. In a continuously available virtual environment where few assumptions can be made about future requirements, agents would need to be employed at many locations to proof the system against future changes. However, this would lead to hugely increased complexity and degraded performance as each message call passes through layers of currently wasted indirection as agents just forward messages.

Event pipes avoid this overuse of agents. The initial system is developed using the message flow which is initially required – clients specify servers and messages are passed to them. If, at a later stage, a different message routing is required, a filter is added in the event pipe which routes the message to a new destination. Figure 4-3 shows how dynamic routing is achieved using event pipes. Events normally flow directly to Server 1, but by inserting a routing filter they can be routed to Server 2 without changing the client.

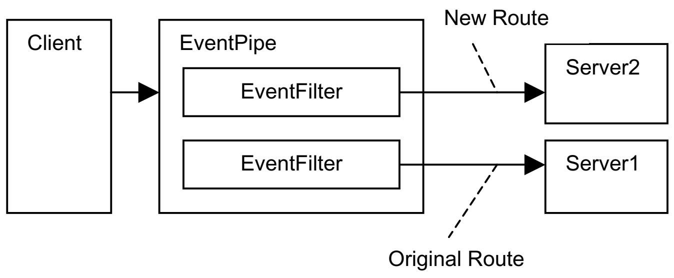


Figure 4-3 Dynamic routing with the DEF framework

In this way a client-server system can evolve to become a 3-tier or *n*-tier system naturally, rather than agents and indirection being specified initially to provide future proofing. If the system ultimately does not need additional routing it remains simple.

The event pipe metaphor also provides a natural way to introduce multiple changes to message passing at once – multiple filters are added or removed to or from event pipes. It is clear by inspecting the event pipes before and after what has changed, however if multiple agents are reconfigured, the changes are difficult to see – the agents which existed before still exist, but now perform different processing. The multiple event filters encapsulate the processing needed for a distributed mechanism more naturally than an agent approach where the mechanism is implicitly expressed through the settings of multiple agents.

4.2.3 Reusable Filters

Implementing infrastructure mechanisms as a network of filters promotes reuse. In particular many consistency mechanisms differ primarily in the order in which updates are distributed among the many application nodes in the system and the order in which updates are carried out. By implementing a generic routing filter and a generic state update filter, we can provide many consistency mechanisms just by configuring the order of these generic filters. Similarly, we can finely control a persistence mechanism by changing the point at which an update is made persistent by being written to stable storage. By changing its relation to update and routing filters, we change the persistence mechanisms semantics by altering whether or not a user is able to see an update before it is made durable.

4.2.4 Reusable Mechanisms

The previous example demonstrates the interdependencies which may exist between conceptually different and independent infrastructure mechanisms. The consistency mechanism may exist without the persistence mechanism and vice versa, but if they operate together the order of the interleaved filters makes a big difference to the semantics of the composite mechanisms. Again this is another opportunity for reuse – two infrastructure mechanisms, which may themselves be composed of reusable filters, may be configured to co-operate in different ways to achieve different system semantics.

4.2.5 Abstraction

While the distributed event filter model achieves much of its flexibility by operating at the low level of events propagating around a distributed network of applications, rather than at the object level used by previous systems (Pavel, 1997, Vellon et. al., 2000) it provides a certain level of abstraction by allowing developers to view the system as an interconnected network of pipes and filters rather than a collection of applications. Event pipes may route events inside a single process or route events across a network link, but appear identical to developers. While the network is clearly not transparent to the infrastructure developer, the uniform abstraction is simpler to work with than the use of an in-process middleware such as Bamboo (Watsen and Zyda, 1998) alongside a network middleware such as CORBA.

4.2.6 Position

As discussed above, the relative positions of filters in an event pipe are very important, often dramatically changing the infrastructure semantics and sometimes being the only difference between two infrastructure mechanisms. For this reason, rich support for specifying positions and dependencies between filters must be provided by the framework. The framework permits filters to specify constraints. These specify which filters, if they exist in the event pipe, must come before or after the filter. Similarly, filters can specify requirements. These specify which filters *must* exist in the event pipe before or after the filter. Specifications of filters which must come before a filter are termed prefix constraints or prefix requirements, while specifications of filters which must come after a filter are termed suffix constraints or suffix requirements. This system of constraints and requirements provides a simple yet powerful way of determining the relative positions of filters and the dependencies between them. To set up an event pipe in a certain configuration the required filters are created, prefix and suffix constraints and requirements are added to the filters and then they are added to the event pipe. The event pipe implementation then attempts to satisfy the constraints and requirements for each filter. If the problem can be solved, the filter is added to the pipe, otherwise the failure is indicated and appropriate action can be taken, either changing the requirements, aborting the initiation of the infrastructure mechanism, or halting system execution as appropriate.

Filter requirements are also used to ensure that the removal of filters from an event pipe does not break any dependencies. The event pipe attempts to satisfy the requirements of all filters without the existence of the filter or filters being removed. If all requirements can be satisfied the filter can be removed, otherwise failure is returned. These semantics for addition and removal of filters ensure that the event pipe remains in a valid state at all times and disallows any operation which compromises its correct functioning. The filters in an event pipe may perform extremely complex processing and involve many dependencies for correct operation. However, this complexity is hidden from the user of the event pipe.

4.2.7 Identification

In order to specify constraints and requirements filters must be able to reference other filters. In order for the framework to allow the replacement of filters and the extension and evolution of the infrastructure these references must be as general as possible, but for the expression of specific dependencies between particular versions of filters the references must also be able to pinpoint a specific instance of a filter. To allow for this the framework supports a hierarchical naming scheme which allows the general and specific identity of a filter to be discovered. The name is made up of the form <Function>.<Version>.<Identity> where function is a sequence of strings describing the filter's function in increasingly specific terms and version is a sequence of strings describing the filter's version in increasingly specific terms. Identity is a single integer which is assigned sequentially to the filters created in an application allowing the precise specification of an individual filter. Using this scheme a filter may specify that it must be positioned before or after a general class of filters, before or after a specific filter of a certain version, or before or after a particular filter. Requirements should be made as general as possible, but no more, to allow the reconfiguration of the event pipe, but to maintain critical dependencies between filters. The model uses the Java definition of versioning (Gosling et. al., 1996) where 1.2 maintains backwards compatibility with 1.1 whereas 2.1 breaks this backwards compatibility, so that sub-string version requirements or constraints will match all compatible versions. Version 2 of a *function* filter which supports the interface of version 1 should be named *function*.1.1 so that a filter with a prefix or suffix requirement *function*.1 will remain satisfied, whereas if the new version behaves differently it should be named *function*.2 to signal to the event pipe that any existing requirements can no longer be satisfied.

4.2.8 Execution

Having configured the event pipe with a collection of filters, the work of the event pipe is to pump each event through the correct set of filters in order to process the event by the appropriate infrastructure mechanisms. When an event is added to an event pipe it must build a list of all of the filters which are applicable to the event and then pass the event to each filter's event processing method in turn.

In fact this description is over-simplified as in order to allow the filters to be as general as possible no assumptions can be made across event processing calls. The simplified description assumes that the initial list of filters which apply to an event, will continue to apply to the event until its processing is complete. If all filters were passive, performing some processing based on the information in an event, but not changing the event then this assumption would be valid. However, filters which modify, delete or create events should also be allowed along with filters which add, remove or replace filters inside their processing methods.

4.2.8.1 List Processing

In order to allow filters to easily delete or synthesise events, a list is passed to the filters processing method containing the single event to be processed. If the filter wants to stop the event being processed further, it removes the event from the list and returns the empty list. If a filter wants to create new events, they can be added to the returned list, while filters needing to change events can either rewrite the event in the list or remove it and replace it with a new event. The events returned to the event pipe are marked as having been processed by the filter and are then processed by the next applicable filter, thus this mechanism is suitable for filters such as interpolators – generating new intermediate events which must not be interpolated themselves (to avoid infinite loops of events being generated). However, there are other filters which generate events directly or indirectly through API calls (rather than in the returned event list): these should be fully processed by the event pipe.

4.2.8.2 Prioritisation

Allowing filters to generate events which must be fully processed by the event pipe requires a choice to be made about the semantics of the event pipe: either the addition of a new event to the pipe causes recursive processing of the new event to completion, or the event pipe could prioritise events which are further along the pipe, so that if a filter called generated an event, the new event would only be processed when the original event had moved completely through the pipe. The latter semantics are more appropriate for a number of reasons. Firstly they minimise the latency caused by event processing as the pipe will always attempt to process the event which needs the least processing to move completely through the pipe. Where event pipes are connected to buffers which may be inspected the latter semantics also ensure that as many events as possible are available to filters for inspection. If a filter generates an event as a result of processing an initial event, prioritising the initial event ensures that it is available for inspection in the buffer before the newly generated event is processed. Finally, the latter semantics are more efficient as fewer events remain partially processed at any one time, consuming memory rather than being completely processed and then deleted. To implement these semantics the event pipe must maintain a buffer of events being processed, must always pick the event furthest down the event pipe for further processing and must continue processing until no events remain in the buffer. Because adding an event to an event pipe causes it to process events until no events remain in its buffer, if an event is added to the event pipe when the buffer is not empty the event pipe must already be processing events. In these cases the event can simply be added to the buffer and left to be processed along with the other events in the buffer.

4.2.8.3 Filter List Caching

In order to accommodate potential changes to its buffer of events to process, and to the filters available to process those events, in principle the event pipe must re-evaluate the list of applicable filters after any filter performs processing. While this allows complete flexibility in the processing which a filter can perform (as the event pipe makes no assumptions about that processing) it makes the event pipe execution very inefficient. A list of applicable filters must be constructed and sorted and the next applicable filter found after each filter performs its processing. If most filters perform little processing themselves then the majority of processing is performed by the framework rather than the mechanisms it supports. In order to maintain flexible filtering with more optimised performance, the event pipe can cache the last used filter list, the next filter to be applied in the list and the event parameters used to construct the list. If the next event to be processed has matching parameters then the cached filter list can be used. If most filters are passive and do not change events, this cache will normally avoid the re-evaluation of applicable filters during the course of a single event's processing. Simple event processing can be performed with a minimum overhead, while the flexibility of arbitrary processing is available at a cost.

The cached filters can also be used between events, so that if consecutive events share parameters, as is often the case in collaborative virtual environments where streams of updates to an object are often generated, event processing becomes more efficient still. Depending on the relative costs of evaluating cached filter sets, generating filter sets and the likelihood of small clusters of event parameters being processed, the cache of filter sets can be expanded arbitrarily to further optimise performance. This may be especially useful where an event pipe in a client application is processing updates to the user's avatar – a large proportion of the events being processed may be updates to various parts of the avatar. By maintaining a cache as large as the number of avatar items the event pipe would vary rarely need to generate a filter set.

4.2.9 Routing

No support for routing is provided by event pipes; this is left to individual filters, allowing events applicable to different items to be routed to different locations, routed out of the pipe at different locations or copied to other pipes while continuing to be further processed in the current pipe. To route an event out of the pipe a filter simply removes the event from the list passed to its event processing method. The event can then be added to another event pipe or buffer, communicated across the network. The ability for routing to take place at any point in the event pipe provides some interesting opportunities – filters may be added which route an event out of a pipe to avoid some standard processing performed by a later filter, or which route an event to another event pipe which performs some additional processing before the event rejoins the standard route through the infrastructure. If during processing there are no further filters which can be applied to an event and the event has not been fully processed and deleted, or routed to another location by a routing filter, the event pipe deletes the event and signals that an error has occurred. This behaviour ensures that if an event pipe is configured incorrectly and events are not routed correctly or disposed of then the virtual environment system will not crash or consume ever increasing resources as events build up in the system. The unprocessed events are lost, but the system will continue functioning for those events which are processed correctly. If different failure semantics are required then a custom filter which processes all events can be added to the end of the pipe to handle unprocessed events differently – writing them to disk for example.

4.3 Deep Behaviours

The Distributed Event Filter framework provides an elegant, natural and flexible framework for virtual environments requiring low-level extensibility, reconfigurability and per-item infrastructure mechanisms. One of the strengths of the model is that multiple filters can be added to event pipes in different applications in order to achieve a fundamentally distributed mechanism through their orchestrated operation. However, in terms of the configuration of the virtual world and the management of its content, it is the mechanism implemented by the multiple filters which is important and not the operation of each individual filter. In order to allow users and virtual world administrators to think on this more abstract level we introduce the concept of the "deep behaviour", another key innovation presented in this thesis.

Deep behaviours are so called because they operate in a similar way to the behaviour mechanisms used in virtual environment systems, but at a lower level. Rather than providing an item in the virtual world with functionality which is directly experienced by the users (e.g. terrain following or collision detection in WorldUp (Sense8 Corp)) a deep behaviour provides the item with infrastructure functionality (for example making the item persistent or subject to transactions). Rather than manipulating the artefact's position to make it follow terrain, a deep behaviour manipulates the filters that process the events describing the artefact's position, for example controlling the way that the position is propagated through the network. The terminology is also consistent with Ivan Vaghi's work on visualising a virtual environments infrastructure using *deeper metaphors* (Vaghi, 2001). Deeper metaphors show users of the virtual world the infrastructure of the system, while deep behaviours provide a mechanism for controlling that infrastructure.

A deep behaviour is loosely analogous to a macro in that it automates the addition, removal and configuration of a group of filters and to a semantic annotation in that it instructs the system how to treat the item and informs the system of how the item is likely to behave.

Deep behaviours provide an interface to infrastructure configuration which allows application developers to think in terms of mechanisms applied to objects, rather than filters processing events. A single behaviour must be created and then manipulated, rather than many filters distributed around the network.

4.3.1 Annotation

As a deep behaviour conceptually applies to a single item in the virtual world in a similar way to traditional behaviours, a deep behaviour implementation requires a way to associate deep behaviours with items in the virtual world. When the association is created the deep behaviour creates appropriate filters to implement the desired low level behaviour, configuring the filters so they apply only to the associated item. If the deep behaviour is changed these changes are translated into reconfiguration of the event filters. Finally, when the mechanism is no longer required the association between deep behaviour and virtual world item is destroyed and the deep behaviour removes the filters from the various event pipes.

In the prototype implementation discussed in chapter 5 deep behaviours are implemented as annotations in the scene graph; this allows deep behaviours to be distributed around the system using the normal data distribution and awareness management mechanisms and for application developers to use the same APIs to manipulate deep behaviours as they use to manipulate the virtual world.

4.3.2 Meta-annotations

Deep behaviours can become an even more powerful tool if they can be associated with other deep behaviours, so that the deep behaviours of deep behaviours themselves can be specified. These associations allow a potentially infinite number of levels of meta-meta-information and a rich syntax for composing complex, parameterised deep behaviours from combinations of simple behaviours.

For example if changes to a deep behaviour might have potentially hazardous effects on the continued running of a virtual environment system, an access control deep behaviour might be used to annotate it. The access control behaviour could restrict access to the deep behaviour item in exactly the same way as it would restrict access to any other item. Without changing either deep behaviour, the combination of behaviours provides new and useful functionality. If the access control mechanism was later replaced with another mechanism, the meta-annotation could be replaced and the original behaviour could take advantage of the new access control facilities without any change.

This potentially powerful form of reuse is the main reason that the prototype implementation discussed in chapter 5 represents deep behaviours as first class items in the scene graph. By doing so the annotation of deep behaviours is automatic – a deep behaviour appears just like any other item to a deep behaviour annotating it.

4.3.3 Fix Points

There are situations where the annotation of deep behaviours can lead to infinite regressions and so some deep behaviours are required as fix points. In the example above there initially seems to be no problem in annotating the access control deep behaviour with another access control deep behaviour – the second access control behaviour specifies the users able to change the users able to change the root behaviour. However, in this model a further access control behaviour would be required to specify access to the leaf behaviour. In these situations carefully limited deep behaviours are required. While the access control behaviour which can be annotated with access control behaviours is potentially useful, an access control behaviour which can only ever be changed by its creator and not annotated with access control behaviours is also required to be the leaf behaviour. If the creator of the leaf behaviour wanted to delegate the responsibility of changing access control they would then add the original access control behaviour between the leaf behaviour and the item whose access is being specified.

4.4 Optimisation

In addition to providing a simple and flexible way to specify infrastructure mechanisms in a virtual environment system, deep behaviours can also be used to optimise the operation of the virtual system. Deep behaviours specify that certain mechanisms must operate on an item, but can also be more generally exploited as meta-information which allows the system to distinguish between items and so optimise other infrastructure mechanisms.

Many of the deep behaviours used to manage the mutability of the environment enforce limits on the mutability, for example requiring co-operation or authorisation to change an artefact which is a well known landmark in the virtual world suggests that the landmark is less likely to change than an artefact which is free of annotation and so able to be freely modified. Similarly an object which is annotated as being persistent and subject to total-ordering through the server providing persistence is likely to be an important object in the world compared to a transient un-annotated object. The system can use these hints to optimise the performance of infrastructure mechanisms which are not specified by the deep behaviour, for example caching items which are unlikely to change due to deep behaviour restrictions. In this way deep behaviours become contracts between the users and the system – the users annotate objects in order to manage the virtual world and the system must provide the mechanisms corresponding to the annotations, but is also free to utilise the semantics of the annotations to optimise its performance. Without the explicit specification of an item’s deep behaviour the system cannot distinguish between items to perform this optimisation.

Deep behaviours which prohibit access to items for a period of time are particularly useful for the purpose of optimising performance as clients caching such items do not need to check the validity of the cached copy for the duration of the period. In addition the client can present a partial view of the world which it knows is correct during periods of disconnection or at start up while waiting for potentially changed parts of the world to be spooled across the network.

The bottom up desire for the system to be able to exploit deep behaviours for optimisation and the top down desire for deep behaviours which manage the world, can be traded off dependent on the application. A virtual environment accessed across a wide area network might make all items subject to 3 month rolling leases by default to ensure that a large amount of the virtual world's content can be cached and used during periods of disconnection. Another virtual world used in a high bandwidth local area network might default to having no constraints on the mutability of objects and only limit changes to a few important landmark objects. These examples fall at either end of a continuum of trade-offs between the optimisation and management driven use of deep behaviours along which virtual environments can move. If the local area network virtual environment needed to be opened up to visitors accessing it across a wide area connection the deep behaviours on the items in the environment could be changed to allow the Wide Area Network (WAN) visitor to cache more of the content of the environment.

4.5 Scalability

For simplicity's sake the above discussion deep behaviours have been described as annotating a single item in the virtual world and as creating multiple filters in the application's event pipe network. Clearly, this requires a great deal of meta-information. In cases where deep behaviours annotate simple artefacts there could be more meta-information needed to annotate the item than information contained in the annotated artefact. Where deep behaviours must annotate many items which employ the same mechanisms a great deal of redundant meta-information is also introduced. In cases where deep behaviours annotate other behaviours to create the composite behaviours described above, or where behaviours create multiple filters the system becomes unscalable – many filters and annotating behaviour items must be created for each item in the virtual world. The large number of filters requires more processing to find the correct filters to apply to an event while the many annotations consume memory. As these overheads grow faster than the information they annotate, more resources are required for system management rather than the actual operation of the system. However, this does not mean that the model is unworkable, just that items in the virtual world should share deep behaviours and filters where possible. New behaviours should only be introduced where genuinely new behaviour is required. A number of ways to group items in the virtual world so that they can share deep behaviours and filters are discussed in sections 4.5.1 to 4.5.4.

4.5.1 Sharing Meta-information Using Aggregation

It is tempting to use the "part-of" hierarchy present in most virtual environments for deep behaviour grouping, so that child items in the hierarchy share the infrastructure specification of their parent unless they have their own overriding deep behaviour annotation. The part-of hierarchy groups items which are part of the same virtual artefact and so intuitively require similar infrastructure processing. A football artefact might be interactive, a complex building site containing many items might be persistent.

This part-of grouping makes sense, but meta-information can only be shared between parents and children. There will still be much duplication of meta-information when items have exactly the same infrastructure requirements, but do not share parents in the hierarchy or are in different environments or locales.

4.5.2 Sharing Meta-information Via References

An alternative is to add references to shared meta-information, rather than annotate items with the meta-information, so an item or event might point to semantic mutability information which can be shared.

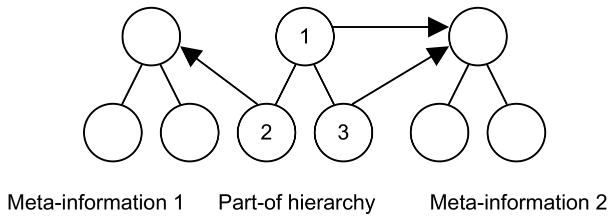


Figure 4-4 Wasteful sharing via references

In Figure 4-4, different items of the part-of hierarchy point to different meta-information graphs. In the example no data is saved as item 1 could be annotated with information 2, item 3 could inherit the information and item 2 could be annotated with information 1. In this case memory and processing is wasted by the references to separate meta-information, which could be more efficiently added to the part-of hierarchy.

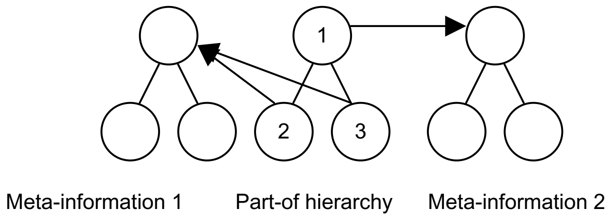


Figure 4-5 Efficient sharing via references

In Figure 4-5, keeping the meta-information separate saves information. If the part-of hierarchy was annotated, items 2 and 3 would have to duplicate information 1, but instead they can reference it. If the meta-information is large this is a considerable saving. In this case inheriting meta-information from the parent in the part-of hierarchy is no help – the part-of hierarchy relationship and meta-information relationship are orthogonal.

4.5.3 Trade-offs

If the meta-information graph is close to the part-of hierarchy – if items often have the same meta-information as their parents – then annotating the part-of hierarchy and moving up the part-of hierarchy to find meta-information where no annotations exist on the current item makes sense and could save memory. If a child in the part-of hierarchy often requires different meta-information than its parent, then references save duplicating meta-information between siblings as in Figure 4-5. Also, if a chunk of meta-information is large and there are relatively few possible combinations of meta-information, then references are better as many references can be redundant (i.e. point to the same information as their parent in the part-of hierarchy) yet still be better overall if they stop a few duplicate meta-information chunks.

4.5.4 Summary

In practice there are likely to be relatively few deep behaviour combinations compared to the number of items in the virtual environment. As the deep behaviour meta-information is relatively big, items reference a shared deep behaviour. The shared deep behaviour just creates one set of filters and those filters process all events to an item which points to that behaviour. The deep behaviour reference is embedded in events to make identification of filters easier.

In a system without deep behaviour sharing, containing *n* items and *o* behaviours each needing *p* filters, $n \times p$ filters are created. These filters must be searched and sorted to process each event. In a reference sharing implementation $o \times p$ filters are needed. These filters can be organised into only *o* filter groups which must be searched to find the correct filters to apply to an event. In practice there will be orders of magnitude more items than behaviour types, so reference sharing cuts down on the number of filters dramatically saving memory and search time and if behaviours often require more than one filter, filter grouping reduces search time further still.

Items can only share deep behaviours, filters and filter groups if they share the parameters for the behaviour between them, so *o* in the example above is actually $q \times r$ where *q* is the number of behaviours and r is the number of configurations, or parameter sets needed for each behaviour. This implies that if the reference sharing mechanism is used, the number of used configurations of behaviours should be kept to a minimum, but in practice this will probably be the case – by quantising parameters so that meta-information can be shared and the number of filters can be kept low. Behaviours can be designed to promote sharing, for example a behaviour applying updates intended for one item to another could use a systematic transform to find the destination item's identity from the source item's identity. This avoids parameterising the behaviour with a target ID, which would require a different configuration of the behaviour for each item which exhibits it.

4.6 Examples

The examples in sections 4.6.1 to 4.6.5 demonstrate the use of the DEF and Deep Behaviour framework. All of the examples have been realised in the prototype implementation and are either novel mechanisms (examples 4.6.4 and 4.6.5) or mechanisms which have previously not been used by collaborative virtual environment systems (examples 4.6.1 to 4.6.3). As such they not only demonstrate the flexibility of the framework, but highlight its potential as a rapid development platform for virtual environment infrastructure mechanisms. The examples are demonstrated in Video Figure 3.

The examples assume an architecture which partially replicates the virtual world in client applications (as is the case with the vast majority of current collaborative virtual environment platforms). The default behaviour of the world is to carry out all updates locally before passing them on to the server where they are carried out on the master environment and then propagated to all other client applications where they are applied to the client replicas. The system has a default event pipe network shown in Figure 4-6.

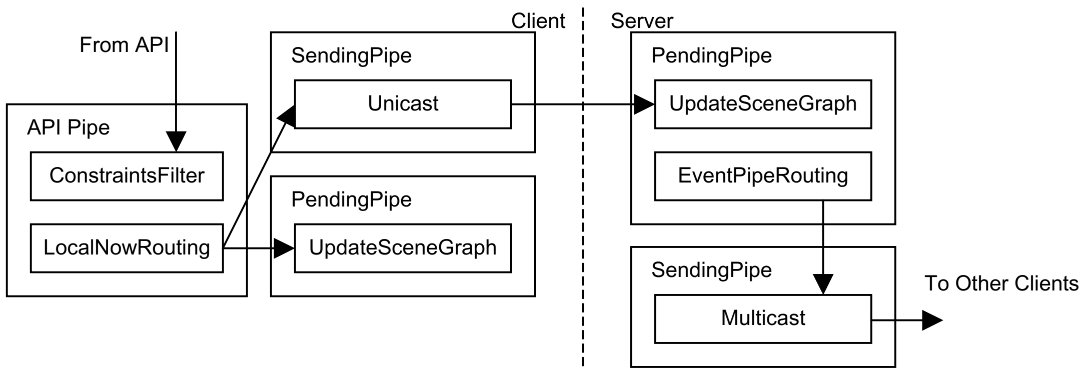


Figure 4-6 Default event pipe network

4.6.1 Trusted Persistence

This mechanism is one of the most important examples of the framework as it demonstrates the interdependence of infrastructure mechanisms. The mechanism is made up of a total ordering consistency mechanism and server based persistence. The motivation is to provide server-side persistence for important items in the virtual world. This could be achieved simply enough by inserting a filter which writes every event processed by it to storage. However, users would be unaware when the important items were persistent – the user would make an update and immediately see its results. Only at some arbitrary time later would their update become durable, and the user would have no idea when that was. As Dourish identifies, users trust immediate events the most, delayed events less and predicted events least (Dourish, 1996). The simple persistence mechanism effectively provides the user with a view of the predicted persistent state of the item through the immediate update of the local replica. If a failure occurs before the update is written to the server store then the update will be lost, the prediction false, and the user's distrust of it justified. In cases where the knowledge of the durable state of the world is more important than local interaction times, for example when updating a virtual bank account, the system must route updates to the server, make them persistent and then return the update to the client where it is applied to the local replica. This mechanism ensures consistency between the persistent state and the client's view of the world. The client can then trust the local state of the item as it is no longer a prediction of durability. These semantics are closer to those of a database than the traditionally optimistic mechanisms of virtual environment systems, but they would be useful for some items in some virtual worlds – by providing per-item semantics the gamut of applications which can be implemented by virtual environment systems is increased.

In order to implement these semantics, the TrustedPersistence deep behaviour adds a routing filter to the client application's API pipe before the standard routing event filter. Instead of copying the event to the pending and sending pipes, the filter just adds it to the sending pipe. The deep behaviour also adds a filter to the server's pending pipe which makes the update persistent just before applying it to the server replica and a second filter just after it which sends the event back to the client. This configuration is shown in Figure 4-7.

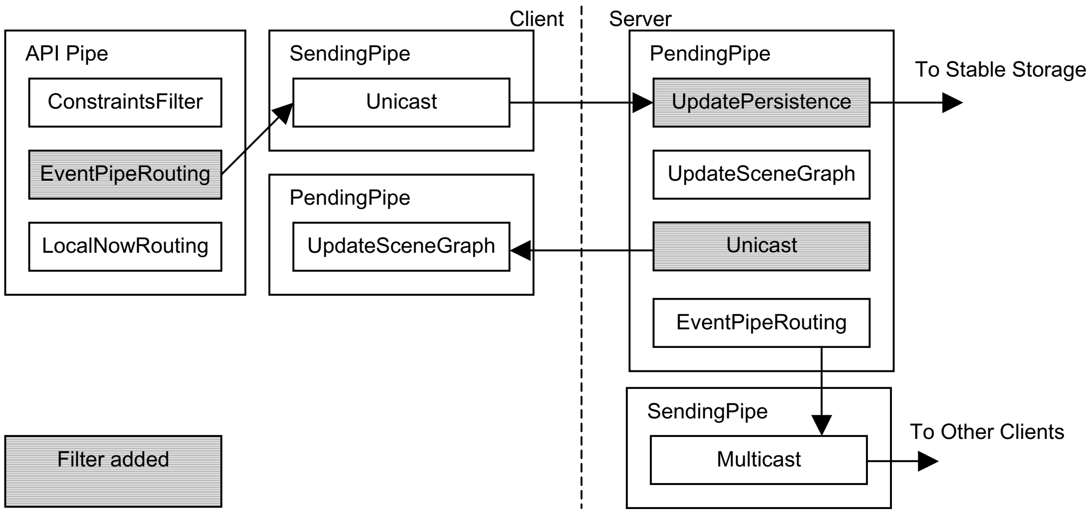


Figure 4-7 Configuration for trusted persistence behaviour

4.6.2 Variant

The Variant deep behaviour demonstrates the flexibility of the framework by providing facilities not usually provided by virtual environment systems. Rather than allowing arbitrary updates to items, updates to items tagged with the variant behaviour create proxy items related to the original item by a syntactic consistency mechanism (Terry et. al., 1998). Other clients viewing the item see its original state and can themselves create related proxy items representing their desired changes to the state of the item. The actual mechanism for creating these subjective views and relating the proxy to the original item will depend on the awareness management facilities of the virtual environment system, but the prototype implementation used aspects (Greenhalgh et. al., 2000) to create overlay environments for each variant. The awareness management facilities can then be manipulated by an administrator to view the different versions of the item and authorise some or all of the updates. This behaviour is useful in situations where user evolution of a virtual world is desirable, but control over the velocity of change and protection against virtual vandalism is required. Instead of updating the shared state of the item, users create desired versions of items which must be approved before they become shared. To implement this mechanism, the Variant deep behaviour first creates a subjective proxy item and then inserts a rewrite filter in the client's API pipe which processes updates to the original item by rewriting the target of the update to be the variant item. The filter configuration is shown in Figure 4-8.

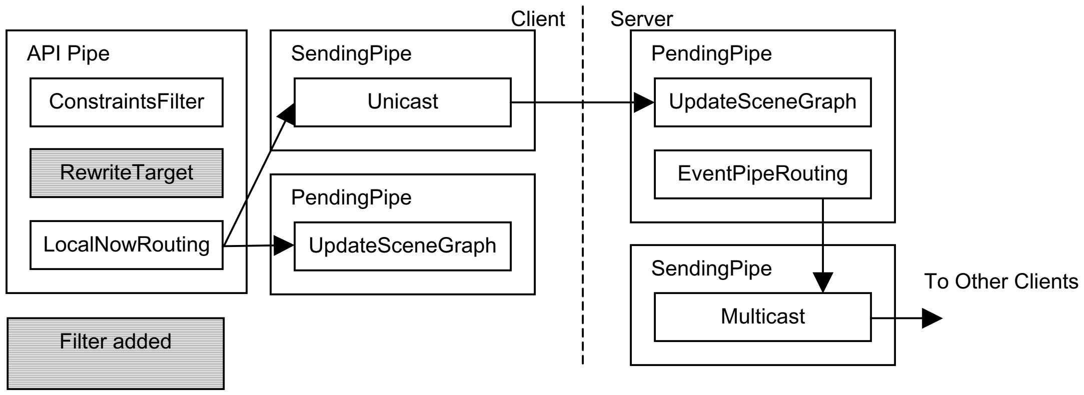


Figure 4-8 Configuration for variant behaviour

One potential scalability problem with the variant behaviour is that if the proxy item is an arbitrary item, then the mapping of original item to target item needs to be stored as a member of the deep behaviour for each item subject to the deep behaviour, which can lead to a large amount of meta information when many items are subject to the behaviour. The solution to this problem used in the prototype implementation was to use a functional relationship between the original item id and the proxy item id – in this case the item id of the original item was the same as the id of the proxy item in the variant aspect.

4.6.3 Leases

The Lease deep behaviour is used to provide a time limited guarantee of immutability for items. The main motivations are to fix parts of the virtual world without committing to a permanently static state and to allow reasoning about the validity of the world for disconnected operation and intelligent caching of the world's state. Like Jini leases (Waldo, 1999) the semantics of the lease deep behaviour are to declare the information annotated by the lease as valid for at least the duration of the lease. Whereas Jini leases guarantee the validity of a service for a time, the lease behaviour guarantees the validity of the state of an item. Like Jini leases, the lease behaviour can also be extended.

This is useful for defining parts of a virtual world as static for the foreseeable future, where the foreseeable future is the length of the lease. If at the end of the lease period the item should remain static, the lease can be renewed and clients can continue caching and using the item for disconnected operation. If during the lease period it is decided that the item should be changed then the lease can be allowed to expire and then the item changed. These “never say never” semantics provide a useful middle ground between declaring an item permanently static as in VRML (Carey et al, 1997) or always transient as in MASSIVE (Greenhalgh et al, 2000).

These semantics are implemented by a simple NullFilter which removes all updates to an item which is inserted by the Lease deep behaviour on creation and removed on expiry. For most efficiency, the NullFilter is inserted as close to the source of updates as possible – at the front of the API pipe.

4.6.4 Triggers

Where leases guarantee the immutability of an item for a certain period of time, trigger behaviours indicate a scheduled change to the item they annotate and provide a mechanism for that change. Like leases, triggers have an expiry time and can be renewed. When the trigger expires it carries out an action by injecting arbitrary events into arbitrary event pipes. This mechanism allows triggers to be as general as possible as they rely only on the existence of event pipes and events, yet can perform any action the system API can perform by the arbitrary sequencing of events. The motivation for triggers are the results of the experiments described in section 3 – many items were created, heavily modified and then discarded in a short period of time, while items which survived this initial period tended to exist for a much longer period of time. By annotating new items with a trigger expiring after this initial “hot” period of manipulation and setting the trigger to add a persistence behaviour to the item, the system can be significantly optimised – of the many updates made to items after their creation, only one state need be written to storage for each item which survives its turbulent youth. More generally triggers provide a mechanism for managing the lifetime of objects by updating, adding or removing other deep behaviours applying to an item based on time or events applied to an item. In this sense triggers are mainly used as a meta-deep behaviour which co-ordinates changes to other behaviours allowing the behaviour of objects to vary dynamically through its life.

4.6.5 Batch Updates

The batch updates behaviour is an example of a bottom up deep behaviour motivated by the desire to optimise the operation of the virtual environment system by restricting the way the environment can change. When a batch update behaviour applies to an item, any update to that item is delayed to the next batch period. Effectively, the batch update behaviour quantises the times at which an item can change. If the deep behaviour framework makes the batch times available in the virtual world (as in the prototype implementation) the limits on when changes can occur can be used to drive caching and disconnected operation. As the system knows that the item cannot change until the next batch period its state can be cached without checking for cache consistency and can be presented to the user as valid during periods of disconnection. In addition, early updates to items buffered until the next update point can be discarded completely if new updates to the item are delivered before the batch point. Given a batch period of *n* seconds, a stream of updates is effectively rate limited to 1 update per *n* seconds. Where many items share a batch update deep behaviour as described in the discussion on scalability the effect of the update behaviour is to created large batches of updates which are applied to large numbers of items in the environment simultaneously. Given sufficient behaviour sharing and sufficiently long batch periods the batch update behaviour can be used to facilitate applications which physically mail out periodic updates on CD – the behaviour ensures that the environment will remain static and so need only be downloaded once, then when the CD arrives updates can be applied en masse without the need to download them. This model is very attractive to applications presenting large, rich environments accessed over low bandwidth connections. An obvious potential problem with the batch updates behaviour is that the new state is not immediately seen by the user performing the update, but this can be solved using the proxy item techniques mentioned in the discussion on the variant behaviour above.

4.7 Conclusion

This chapter has presented the DEF framework for flexible extensible infrastructure processing in distributed systems in general and virtual environment systems in particular. It has compared the framework to the facilities for extensibility provided by existing virtual environment systems and shown that the dynamic routing possible with the DEF framework is not possible with existing systems and is useful for virtual environment systems especially for the reconfiguration of consistency mechanisms. The dynamic routing is then compared against traditional object oriented mechanisms and is shown to be a more general case of the *n*-tier model. Having established that the model is an appropriate solution for the problem of building continuously persistent virtual environments the chapter presented the novel innovations of the model explaining their operation and why they are needed. The chapter then presented the deep behaviour framework for managing event pipes and filters, explaining the advantages of making the behaviours first class items in the virtual environment in terms of providing a unified interface for application programmers, allowing behaviours to annotate other behaviours and allowing the system to reason about behaviours to optimise performance. Finally, a number of example behaviours are presented which illustrate the spectrum of potential uses of the framework and explain their operation in terms of the filters used and their configuration. This chapter has explained the framework in an implementation independent manner in order to present it to readers not familiar with virtual environment systems and to make it possible for developers of existing and future systems to implement the framework in as wide a range of situations as possible. The next chapter presents the prototype implementation of the framework at a lower level which provides examples of how the features described here can be implemented.

5 Implementation

This chapter details the prototype implementation of the framework presented in chapter 4. In particular it describes how the innovative additions to the pipes' and filters' architecture such as constraint satisfaction can be realised and how the framework can be realised by modifying an existing collaborative virtual environment platform such as MASSIVE-3.

Section 5.1 provides background information on the original MASSIVE-3 platform before sections 5.2 to 5.4 describe the workings of the core event pipe implementation. The deep behaviour implementation is described in section 5.5 and the persistent store which was developed is discussed in section 5.6. Finally some conclusions regarding the implementation are presented in section 5.7.

5.1 MASSIVE-3

The starting point for the prototype implementation was the MASSIVE-3 collaborative virtual environment system (Greenhalgh et. al., 2000) which was developed from an initial implementation to a mature system over the course of this research. An overview of the MASSIVE-3 system is given in chapter 2, so this section only describes the details of the system's architecture which are important to the realisation of the DEF/Deep Behaviour framework.

5.1.1 Events

The explicit representation of events in MASSIVE-3 was designed to allow future mechanisms to adapt the system by using reflection to introspect it – tailoring system performance based on the events being generated or processed. In fact the infrastructure of MASSIVE-3 can be viewed as a single, hard coded event pipe configuration – events are generated at the API, and processed through a sequence of methods. The DEF framework takes this architecture and allows the sequence of methods to be changed, the route of events through methods to be changed and for different methods to be configured to process different events at run-time.

5.1.2 Serialisation

The prototype implementation of persistence uses MASSIVE-3's support for serialisation in a similar way as the initial implementation of persistence described in Chapter 2, but it is complicated by the need to write parts of objects to a persistent store at different times. Whereas MASSIVE-3 assumes that an entire composite object is written to a single location at once, a persistent data store needs to split the serialisation into separate records connected by references. This allows a child object to be subsequently written without the parent being serialised again. For example an entire environment should not be serialised each time an item in that environment needs to be made persistent. This problem is overcome by implementing a more intelligent stream class which is aware of when aggregate objects are being written and can write them to new records. This process can be made transparent to the object serialising itself, so that the original MASSIVE-3 serialisation mechanism can still be leveraged. The details of this mechanism can be found in section 5.6.2.

5.1.3 Summary

Many features of MASSIVE-3 provided inspiration for the DEF framework, such as the explicit representation of events and the use of event filters for rate limiting and interpolation. However, other features obstructed the prototype, in particular the reliance on single transferable ownership for consistency. Many of the more exotic prototype deep behaviours, such as the variant behaviour, allow looser consistency with only syntactic consistency being maintained between variants which are later merged. In these cases MASSIVE-3 sometimes had to be fooled, or its processing bypassed, to allow these mechanisms. It is conceivable that similar problems would have been encountered whatever virtual environment system had been used as a basis for the implementation. The process of shoehorning a new infrastructure framework into an existing system is always likely to meet some resistance. The alternative would have been to construct a minimal virtual environment system from scratch to demonstrate the framework. This would have been easier and resulted in a pure implementation, but would have required a lot of subsequent development of supporting facilities to enable the experimental use of the framework. In fact, the prototype implementation of the framework is the best of both worlds as it is a modified MASSIVE-3 and so can draw on many of the facilities provided by MASSIVE-3, but the core framework classes are independent of MASSIVE-3 and so could be used as the basis of a future pure implementation.

5.2 Event Pipes

The implementation of the DEF framework in MASSIVE-3 was largely as described in the Model chapter. The Event class is the only MASSIVE-3 class which the EventPipe and EventFilter classes rely on. Figure 5-1 shows the main classes which implement the DEF framework and their relationships.

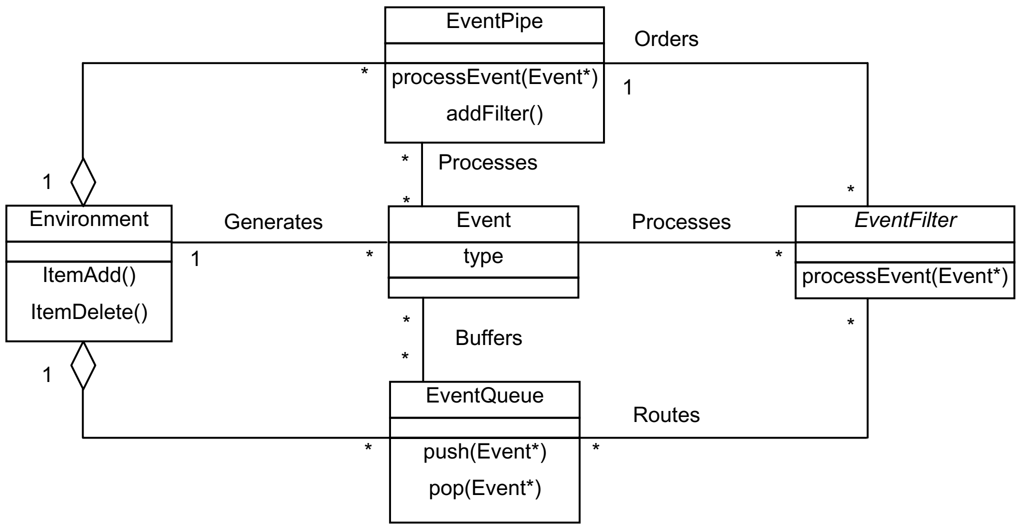


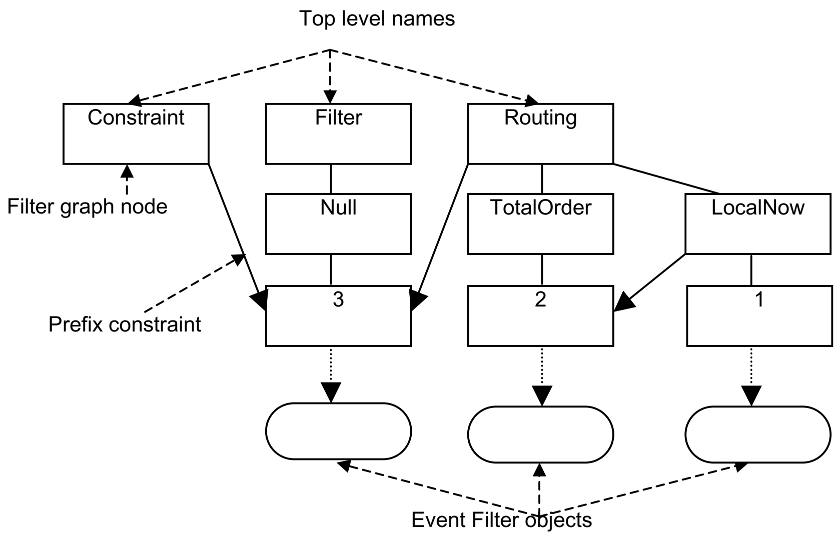
Figure 5-1 Class diagram of the main classes in the DEF framework

From an aggregation point of view, an Environment may contain a number of EventQueues and a number of EventPipes which, in turn, may contain a number of EventFilters. The Environment API is called by applications and these API methods generate event objects which are passed to EventPipes which process the events through each EventFilter they contain in sequence. The EventFilters may route the events to further EventPipes or EventQueues, generate new Events or delete fully processed Events. Either periodically, or as a result of some action, the Environment pumps Events waiting in its EventQueues through its EventPipes to complete their processing.

5.2.1 Constraint Satisfaction

One of the main tasks of EventPipe objects is the correct ordering of the filters they contain based on the constraints which the EventFilter objects specify. Whenever a filter is added to an EventPipe the EventPipe must find a total ordering from the partial ordering defined by all the constraints and requirements specified by the filters in the pipe and the new filter. Whenever a filter is removed the EventPipe must make sure the requirements of all filters are still met. If a total ordering cannot be produced, the filter cannot be added or removed. In order to find a valid ordering, the EventPipe maintains a filter graph which is a hierarchy of known filter names. Each filter has a name of the form <substring>.<substring>.<substring> as described in Chapter 4. A filter name may contain an arbitrary number of substrings, each substring specifying the operation of the filter more precisely until the final substring, which is a unique identifier for the filter. Each node in the filter graph represents one of the substrings in the name of one or more filters. A path from the root of the graph to a leaf node corresponds to the name of a filter. Constraints are represented by directed arcs between any two nodes in the tree – producing a constraint graph over the name tree. A reference to a node near the root of the tree corresponds to a general constraint which refers to a class of filters, while a reference to a leaf node corresponds to a specific constraint on a single filter.

Figure 5-2 shows an example filter graph corresponding to a event pipe containing 3 filters named “Routing.LocalNow.1”, “Routing.TotalOrder.2” and “Filter.Null.3”. “Routing.TotalOrder.2” has a suffix constraint specifying it must come before “Routing.LocalNow” filters and “Filter.Null.3” has two suffix constraints specifying that it must come before “Routing” filters and “Constraint” filters. Note that although filters can specify both suffix constraints (filters which must come after the filter) and prefix constraints (filters which must come before the filter), these are all converted into prefix constraints in the filter graph to drive the process of ordering the graph. Although no “Constraint” filters exist in the event pipe, a “Constraint” node exists in the filter graph as it is referenced by a constraint. If a filter with a name beginning with “Constraint” is subsequently added to the event pipe, it will be positioned under the "Constraint" node in the graph, causing the null



filter's constraint to apply to the filter without further change.

Figure 5-2 An example filter graph

Having produced the filter graph, the EventPipe finds a total ordering. A path from a root node to a leaf node is found such that no nodes in the path have prefix constraints. In the example shown in Figure 5-2 only the path "Filter", "Null", "3" satisfies this requirement. The leaf node at the end of the path is removed from the graph and nominated as the first node. This satisfies any prefix constraints which target the removed leaf node and so these constraints are removed from the graph. In the example this removes the prefix constraints from the "Constraint" and "Routing" nodes. The process of finding paths with no prefix constraints, removing the leaf node and appending it to the total order and removing the prefix constraints which reference the removed node continues until no nodes remain in the graph or until no nodes can be removed. If nodes remain, conflicting constraints exist and the process fails. If an order is found, the filters referenced by the nodes are sequentially numbered to allow fast sorting during event processing.

The classes collaborating during constraint satisfaction are shown in the class diagram below. Note that the FilterGraphNode does not explicitly reference the targets of its constraints, but rather maintains a count of how many prefix constraints remain unsatisfied and references to the nodes, which have a prefix constraint which target the node. This optimises the process of ordering the graph as finding a node to remove just requires testing the nPrefixNodes member of each FilterGraphNode and removing constraints just requires decrementing the nPrefixNodes member of the nodes referenced by the suffixNodes list of the removed node.

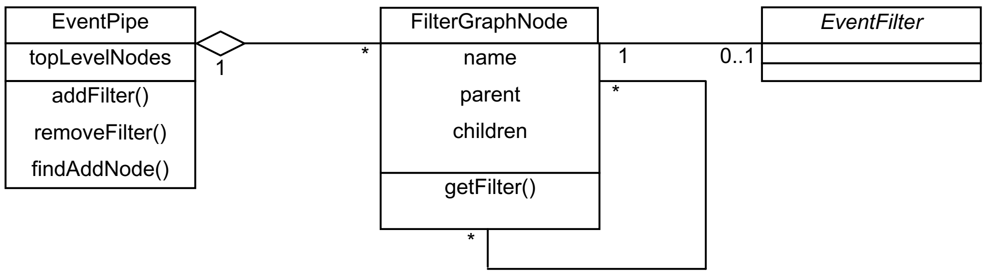


Figure 5-3 Classes used in constraint satisfaction

The process of adding a filter, showing the construction of the filter graph and creation of a total order is illustrated in Figure 5-4. An EventFilter, filter, is named “routing.type1” and has one prefix constraint which specifies it must be after all “persistence” filters. The filter is added to an initially empty EventPipe, pipe, by calling pipe.addFilter() with the filter as the argument. The EventPipe gets the filter’s name by calling filter.getName() and then finding the corresponding FilterGraphNode by calling findAddNode(“routing.type1”). As the EventPipe was initially empty this call creates a root “routing” node and adds a leaf “type1” node to it as a child, which it returns as filterNode. The EventPipe calls filterNode.setFilter(filter) to associate the node with the filter before getting the filter’s prefix constraints and looking for the node which corresponds to the constraint by calling findAddNode(“persistence”). Once again a node does not exist in the filter graph, so findAddNode() creates a “persistence” root node and returns it as targetNode. pipe then increments filterNode’s nPrefixNodes member and adds filterNode as a suffixNode to targetNode. With the filter graph constructed pipe then attempts to create a total order from the filters. It calls findFirstNode() which performs a depth first traversal of the filter name tree looking for paths from root to leaf with no prefix constraints. It finds the only path, which leads to the “persistence” node and returns it. The EventPipe removes this node from the filter graph, appends it to the total order and gets its list of prefix nodes. This list contains only filterNode, which has its nPrefixNodes count decremented. With all of its prefix constraints satisfied the next call to findFirstNode returns filterNode, which is appended to the total order. A final findFirstNode call returns the “routing” node, which like the filterNode has no suffixNodes. With a valid total ordering found, the EventPipe sequences all the filters. As filterNode is the only node which references a filter, it is positioned first in the total order, although if any filters with names beginning with “persistence” were subsequently added, they would be positioned before the routing filter.

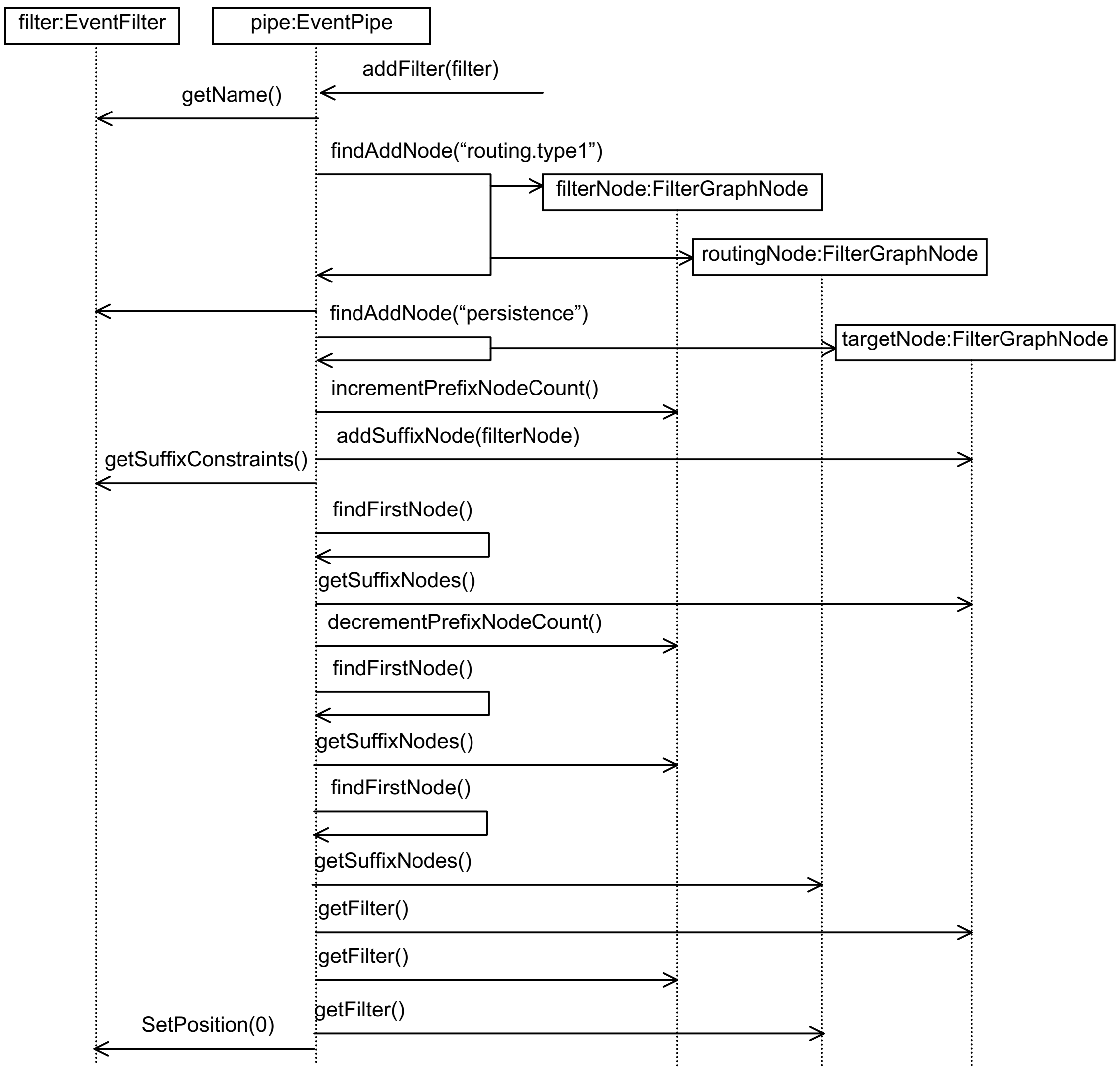


Figure 5-4 The process of adding a filter to an event pipe

5.3 EventFilters

While the core classes were kept largely independent of MASSIVE-3 a large portion of MASSIVE-3 had to be refactored into a number of EventFilter classes. MASSIVE-3 effectively became a standard configuration of EventPipes and EventFilters into which new filters could be introduced to customise Event processing at any point.

Prior to this refactoring API methods generated explicit Event objects which were passed through a number of method calls. In many cases each method could be moved to a separate filter, or where multiple flows of control existed in a method, each flow could be implemented as a filter. This reduced the time required to process an event as, instead of testing which flow to take each time the method was called, the test could be made once when the environment was created and then the appropriate filter inserted in the event pipe, which did not need to perform the test per event. Examples of this were the sets of filters constructed for total order or local now operation. In MASSIVE-3 a total order consistency mechanism can be specified when an item is created. If this option is specified, a flag is set which is tested when processing the constraints and routing for each Event. By moving the constraints processing to TotalOrderConstraints and LocalNowConstraints and using configurable generic EventQueueRouting filters these tests could be avoided. Instead of setting a flag and then repeatedly testing it, the Environment's event pipes could be populated with appropriate filters when the Environment is created. Figure 5-5 shows some of the filters implemented in the prototype.

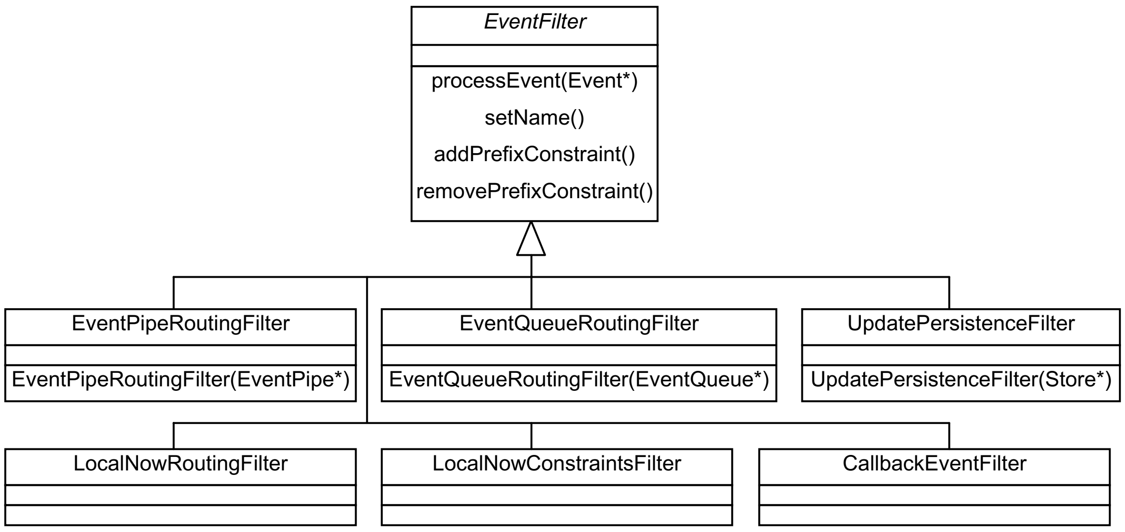


Figure 5-5 The EventFilter class hierarchy

The EventFilter class itself defines the abstract processEvent factory method (Gamma et. al., 1995) and implements mechanisms for naming and constraints. The setName() method takes a filter name incorporating a hierarchical description of the filter's function and optionally a version number and appends a unique identifier to the name allowing the specification of a particular filter. The add and remove constraint functions annotate a filter with the names of filters which must come before (prefix constraints) or after (suffix constraints) if they exist in the same EventPipe. The constraints mechanism allows a large number of consistency and persistence mechanisms to be constructed just using the EventFilters shown above. Using constraints to position an UpdatePersistenceFilter relative to filters updating the local scene graph or routing events can provide a number of persistence semantics. These range from all updates seen by the user being durable, to all updates being seen by the user before they are written to stable storage. The bottom row of filters show some of the filters which make up the refactored MASSIVE-3 implementation. The LocalNowRoutingFilter implements optimistic consistency by processing locally generated events before sending them to the server running the environment. The more conservative total order consistency can be built using standard EventPipeRoutingFilters. By moving constraints processing into a filter, other filters which remove events from the EventPipe can be positioned before it to avoid wasted processing.

5.4 Legacy Filters

Prior to the prototype implementation, MASSIVE-3 supported a primitive type of event filter which was used to implement rate limiting and interpolation of Events. The Event was passed to a C++ callback function which could modify or reject the Event. The synthesis and addition of new Events, which was needed for the interpolation filter was provided for by event injection methods which effectively broke the encapsulation of the Environment API. To accommodate these legacy filters and allow new EventFilters to be positioned around the rate limiters or interpolators the CallbackEventFilter, shown in the diagram above, was developed which wrapped the legacy event filters. Its implementation calls the callback method of the legacy event filter and then checks to see if the legacy event filter used the event injection methods to add Events to the pending EventQueue. If so, they are removed from the pending queue and added to the list returned by the CallbackEventFilter. The CallbackEventFilter wrapper makes the new EventPipe framework look like the legacy MASSIVE-3 implementation to the legacy filter, but makes the legacy filter look like a standard list manipulating EventFilter to the EventPipe framework.

5.5 Behaviours

The behaviour mechanism used to annotate the contents of the virtual environment in the prototype implementation is an extended version of the existing MASSIVE-3 behaviour mechanism. BehaviourData items in a MASSIVE-3 scene graph have a name, a configuration string and a context in which they will run. When an Environment replica creates a BehaviourData item, it compares its role with the context in which the behaviour should run to decide whether to instantiate a Behaviour sub-class corresponding to the name in the BehaviourData item. If the Behaviour should be run it is created and given its position and the configuration string as parameters. The Behaviour constructor acts as an entry point in which the Behaviour can create callbacks in which to do the work of the behaviour.

This model could have been used without extension to provide an entry point in which to instantiate and configure filters. However, the mechanism was really too primitive to support complex deep behaviours. In particular, the configuration string made reading and updating parameters of behaviours very difficult. For a single parameter to be read the string had to be parsed in an undefined way and the desired parameter extracted. The configuration string could be arbitrarily updated with data which could not be understood by the target behaviour. Where behaviours contained arbitrary child objects they would have to be serialised to the configuration string.

To address these problems, the behaviour mechanism was extended to become a lightweight object model. The goal was for a deep behaviour to appear to applications using the Environment API as a normal sub-tree in the scene graph and to appear to the Behaviour as a composite object. This meant that instead of having a configuration string, the behaviours had named members which could be read and updated. The members were implemented in the scene graph using an AttributeData object for strings and numeric values and a new StateData scene graph item for arbitrary scene graph objects. These appeared in the scene graph as children of the BehaviourData item. The Environment API was extended with new updateBehaviourState methods and the Behaviour class was extended with new getState and setState methods. These extensions allowed behaviours to publish their state in the scene graph where it could be read and updated by applications and written to Stores to make behaviours and deep behaviours persistent. The introduction of the StateData item solved the problem of behaviours with parameters of arbitrary types. Splitting the members up into individual ItemData objects solved the problem of extracting parameters from a single string. The problem of applications being able to write invalid data to a parameter was solved by using MASSIVE-3's control lock mode of updating (Greenhalgh et al, 2000). When behaviours are instantiated, they publish default values for each of their members to the scene graph as control locked items. When an application wants to change a parameter it calls the updateBehaviourState Environment API methods which annotate the control locked member with an update request. This triggers a callback in the Behaviour class which passes the member name and requested value for the member to a virtual updateRequest member. Concrete behaviour sub-classes can examine the new value and if it is a valid type and value, can return a value to the Behaviour callback indicating the member can be updated. While this subverts the scene graph into providing an object model as well as a frame of reference hierarchy, the extended system provides much richer support for the complex behaviours required for the framework than the initial configuration string implementation.

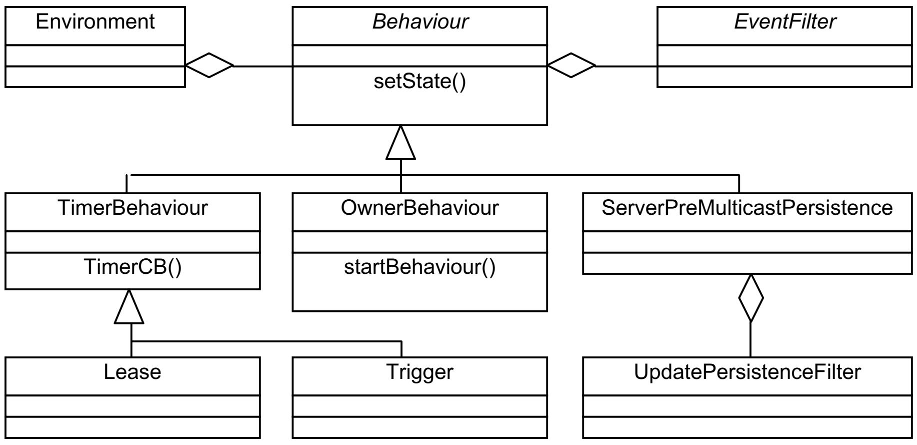


Figure 5-6 Behaviour classes

Figure 5-6 shows part of the Behaviour hierarchy which implements the deep behaviours available in the prototype. Environments create Behaviour sub-classes in response to BehaviourData items being added to the scene graph, and the Behaviour object creates and configures EventFilters for that Environment dependent on the node in which it is instantiated. If a behaviour item is added to the scene graph specifying a ServerPreMulticastPersistence deep behaviour, the ServerPreMulticastPersistence object instantiated on the server creates an UpdatePersistenceFilter. This is positioned in the server's event pipes so that all updates made to the behaviour item's parent are written to stable storage before they are multicast to clients. The TimerBehaviour and OwnerBehaviour classes add facilities to the Behaviour class which are used by several concrete Behaviour sub-classes. OwnerBehaviour monitors the ownership of the item the behaviour is annotating. It calls virtual startBehaviour and stopBehaviour methods when the ownership is gained or lost by the node in which the OwnerBehaviour is instantiated. This allows, processing to be carried out on the node in which events are generated to avoid transmitting Events across the network which will be subsequently discarded. A TimerBehaviour maintains an expires property and provides a TimerCB() virtual method which is overridden by its sub-classes to perform processing when the timer expires. The Lease deep behaviour makes the item it annotates immutable for the duration of the lease. The expiry time of the Lease is visible as a child of the BehaviourData item in the scene graph allowing applications or users to extend the lease by updating the scene graph. The system can reason about the correctness of the annotated item in order to optimise caching and disconnected operation. When a Lease behaviour is asked to validate an update to its expiry time it only allows updates which increase the expiry time.

5.6 Persistence

As the prototype implementation was intended to explore the possibilities provided by the framework, the goal in the implementation of persistence was to make it as general and flexible as possible, so it could be used by many deep behaviours to provide persistence in different ways. This general service for persistence was provided by an abstract store interface which provided a mechanism for filters to store arbitrary objects and then retrieve them by name. The design of the persistence subsystem closely resembles the "two layer persistency subsystem" pattern described by Keller (Keller 1998). This is used by most object-oriented databases and object/relational access layers. In terms of the pattern language Keller describes, the subsystem uses the "multilayer class" pattern (Coldeway and Keller, 1996) to move data between an object layer and storage layer via the Store class. The system uses the "foreign key aggregation" pattern (Keller, 1998) to enable incremental reading and writing of composite objects, using the unique identifiers defined by MASSIVE-3 as the foreign keys. The system uses the "objects in BLObs" pattern (Keller, 1997) to map objects to the relational database. While this means that objects in the database cannot be queried on their attributes, it allows objects to be stored without the generation of schema from the type system. As long as objects implement serialisation, which is required for replication anyway, current and future types will be able to be stored and recovered. The "objects in BLObs" approach means that the system is not limited to using relational databases for storage – any storage mechanism capable of associating arbitrary data with a name, such as file systems, can be used. The classes which make up the persistence implementation and their relationships are shown in Figure 5-7.

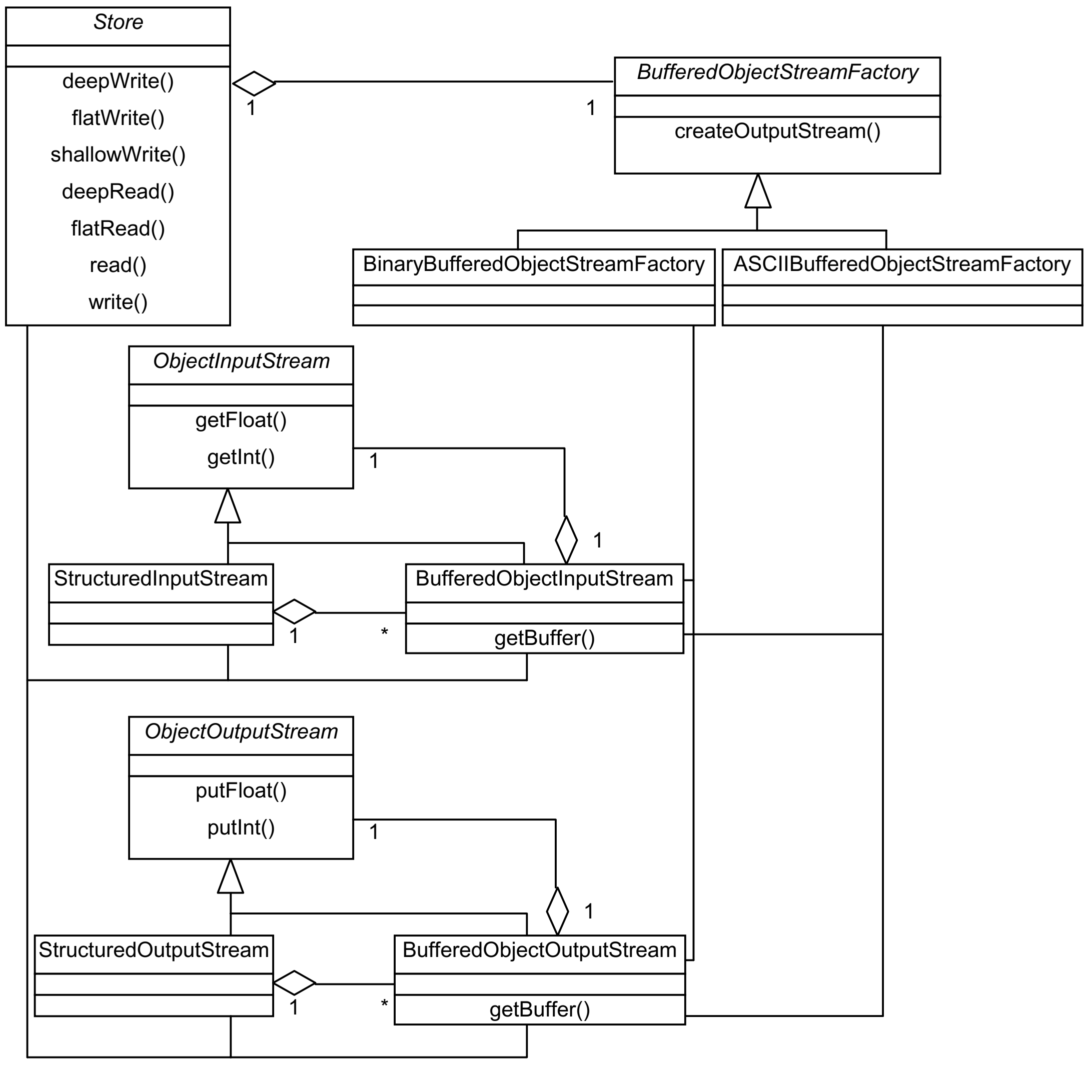


Figure 5-7 Persistent store classes

5.6.1 Store

As noted above, the Store class corresponds to the "multilayer class pattern" (Coldeway and Keller, 1996). The clients of the Store (that is the EventFilters and applications which write or read persistent data) use the high level API which consists of the deepWrite, flatWrite, shallowWrite, deepRead and flatRead methods. These methods take an object and either read or write its state to or from storage. The concrete implementations of the Store interface implement the remaining methods: read(), write() and remove(). These methods just require that the Store implementation is able to store data and associate that data with a key which can later be used to recover the data. This allows Stores to be implemented using file stores, relational databases, object bases or other storage mechanisms. Although not shown on the diagram above, ODBCStore and FileStore implementations of the Store interface have been developed for the prototype. The small interface which must be implemented by concrete subclasses makes the development of future higher performance stores easy.

5.6.1.1 FileStore

The FileStore implementation of the Store interface stores each record in a separate file, using the record name as the file name. When a record must be read, the file whose name matches the requested record name is read and its contents returned. There are a number of problems with this implementation of the store – it is not high performance as no database style indexing is used to find files when they are requested. There are also limitations on allowable file names in most file stores, both in terms of legal characters and maximum length. The FileStore implementation substitutes illegal characters in record names to produce a file name when a record is created and then repeats this translation when the file is read. The advantages of the FileStore implementation are that it is platform independent, and was quick to develop, allowing the Store architecture to be quickly prototyped.

5.6.1.2 ODBCStore

The ODBCStore implementation of the Store interface uses Microsoft's Open Data Base Connectivity API to store records in relational databases. The open nature of the API means that once an ODBC data source has been set up on a computer, the API can find, connect and use it. The ODBC data source can be any DBMS which provides an ODBC driver, such as Microsoft Access or SQLServer or Oracle. Although ODBC is a Microsoft technology, ODBC has been implemented by third party vendors to operate on many operating systems and hardware platforms like Linux and other flavours of UNIX. The ODBCStore implementation does not take full advantage of the relation mechanisms provided by ODBC, as it just stores records in named fields within a single table. However, the optimised indexing provided by relational databases makes the implementation much higher performance than the FileStore implementation. The access control mechanisms provided by ODBC provide limited security for the data stored in an ODBCStore.

5.6.2 Structured Streams

The main work performed by the Store class is the mapping of objects passed to the Store for storage to flat records which can be written by the concrete Store implementations to stable storage. The simplest way to perform this mapping would have been to extend the existing ObjectOutputStream and ObjectInputStream classes to serialise objects to, or read objects from, memory buffers which could be written or read from a store record. The problem with this approach is that it would have resulted in composite objects being stored in a single record which would have to be read to recover the state of member objects. An ObjectOutputStream was developed which identified when a composite object was writing a member object to a new buffer. This allowed allow member objects to be written to their own records and so be read or updated independently of their parent. The stream then identifies when the member object had finished serialising itself and writes the member buffer to the Store and writes a reference to the store record in the parent buffer, before continuing to serialise the parent to its buffer. A complementary ObjectInputStream was developed which can read a record from a Store and transparently read further records when a reference to another record is found. To an object serialising or deserialising itself to or from a stream, the structured stream appears to be simply a source or sink for integers, floating point values or strings. Transparently, the stream may be creating, updating or reading dozens of records in a Store.

5.6.2.1 StructuredObjectOutputStream

In order for the Structured Stream classed to identify when a new object is being written, the StructuredObjectOutputStream must keep track of the calls being made to it by the object serialising itself. Fortunately, the stream of calls is not without structure. Before an object is written its class name is written to the stream and then an object start marker is written. The state of the object is written and finally a object end marker is written. Any member objects follow the same pattern, so an ASCII stream using curly braces for the object start and object end markers for an object of class A containing an object of class B where both classes contain an integer, looks like this:

```
A { B { 1 } 1 }
```

The sequence of calls the method made to the StructuredOutputStream is:

```
putString("A");
putObjectStart();
putString("B");
putObjectStart();
putInteger(1);
putObjectEnd();
putInteger(1);
putObjectEnd();
```

Note that the StructuredOutputStream is unaware that these calls come from 2 different objects, but it is easy to identify the nested object from the nested putObjectStart() and putObjectEnd() calls. One problem is the class name "B" should also be part of the nested record, but it can only be identified as the class name once the following object start marker has been written. To solve this problem the serialisation mechanism was changed to call a special putObjectHeader() method to write the class name. This was possible as the Serialisable bass class wrote the class names, rather than every MASSIVE-3 class. The first four calls now looked like this:

```
putObjectHeader("A");
putObjectStart();
putObjectHeader("B");
putObjectStart();
```

It was then easy for the StructuredOutputStream to identify the start and end of objects which could be written to separate records, which would currently look like: “A { 1 }” and “B { 1 }”. Note that there is no reference between the two records, so no way of finding the record for B when A’s record is read. To solve this, the putObjectHeader() method was extended to provide both class name and object name information. The object name information was used to name the record containing the object, so that when StructuredOutputStream::putObjectHeader() is called it creates a new record named by the object name and embeds a reference to the record in the parent record. If the objects were called A1 and B1, record A1 in the store would contain the data “A { B1 1 }” and record B1 would contain the same data as before.

The biggest problem encountered in the parsing of these serialisation calls was that objects derived from other Serialisable classes called their parents writeObject() method and then wrote their data between object start and end markers, so an ASCII serialised object of class C, which derives from B and contains a floating point, looks like this: “C { 1 }{ 1.0 }”. The problem is that the StructuredObjectOutputStream does not know anything about the class hierarchy, so that whenever the end of an object comes, it does not know if a new object start marker will be written and a derived class’s data serialised. This is really a problem with MASSIVE-3s serialisation mechanism, but it is not a problem which can be fixed without changing every class’s serialisation code, so instead the problem was worked around. When an object could have ended – its object end marker is written to the stream – only an object start marker will continue that object. Any other token indicates that the object is finished and the parent object is writing itself to the stream again. The StructuredOutputStream uses this information to identify the ends of objects. When object start and end markers are written to the stream the StructuredOutputStream keeps track of the depth of nested markers. When something other than an object start marker is written to a record at depth 0 the stream knows the object has finished. The record is written to the Store and the token written to the parent record. For root records the stream just keeps the record open until a new root object is written to the stream, or the stream is closed at which point the record can be safely written to the Store.

5.6.2.2 StructuredObjectInputStream

Having broken complex objects down into separate records for each of their member objects, the process of reading them back in is relatively simple. Given an initial object name, the StructuredObjectInputStream reads the first record in and starts returning tokens from the record in response to the objects get*() calls. When the object requests that an object header be read, the stream reads a class name and object name. The class name is returned to the object being deserialised and the object name is compared with a null string. If it matches the StructuredObjectInputStream continues reading from the current record, if it is non-null, the stream reads the record from the database matching that object name and starts returning tokens from that record. The process is transparent to the object deserialising itself.

5.6.3 Store Format Independence

When designing the Store and stream implementation, it was desirable to make the Store implementation independent from the data format written in each record. MASSIVE-3 supports both binary and ASCII serialisation and due to the wide range of projects it is used for, there is always the potential need for other formats such as XML. The move to another data format in the future should not require any changes to code dealing with storing and retrieving records from an ODBC source, conversely the addition of a new Store implementation should not require developing classes to store all legacy data formats in that Store. If each store-format combination required the development of a new class, the combinatorial explosion would soon become unmanageable. Ideally, adding a new Store would require developing a class to deal with reading and writing from the store and adding a new data format should require only the development of a class to read and write that format.

To achieve this, the Structured Stream classes support the ObjectInputStream and ObjectOutputStream interface and manage the breaking up and reconstitution of composite objects, but defer the work of reading and writing to and from a buffer to BufferedObjectInputStream and BufferedObjectOutputStream classes, which themselves defer reading and writing tokens to an ObjectInputStream or ObjectOutputStream, which could be an ASCII, binary, XML or other format reader or writer.

To specify the combination of Store and data format which should be used, a StreamFactory (see the Abstract Factory pattern in Gamma et. al. 1995) is passed to a Store. When the Store and the Structured Streams it creates need to read and write records they use the StreamFactory to create format specific streams to read and write the data. The relationships between the Store, factories and stream classes are shown the class diagram above.

As a result of this architecture, the implementation of a Store requires only a Store sub-class to be implemented and the addition of a file format requires only the implementation of an ObjectOutputStream and ObjectInputStream for that format and a StreamFactory to create the format specific streams. The new Store can then read and write any format and the new format can be written to or read from any Store.

5.6.4 Deep, Flat and Shallow Reads and Writes

The discussion above has described the case when a composite object is separated into many records or reconstituted from many records. This method of reading and writing is termed *deep* reading or writing after *deep copying*, however, there are many cases when the record for a parent object must be updated, but its member objects may have not changed. Deep writing the parent object would be a waste as the records for its member objects would be rewritten when they haven't changed. To avoid this waste, the Store class supports *shallow* writes which update the record of the parent object without rewriting the member object records. To achieve this it uses a sub-class of StructuredOutputStream – ShallowOutputStream. ShallowOutputStream just overrides the StructuredOutputStream::writeRecord() method and tests if the record being written is the root record. If it is the record is written to the store. If not, the method returns without writing anything to the store. Another potential problem with deep writing is caused by objects referencing members which are different objects, but which return the same object name and so would be stored in the same record if the two composite objects are deep written. If the composite objects were written to the store and subsequently read, both member objects would have the state of the second member object to be written, which is clearly an error. This problem arises most often in the event pipe framework when multiple update events are written to a store. Each update represents a different state of an item, but they all refer to an item which returns its item id as its object name. If a sequence of update events are deep written to a store all item states are lost except the last one to be written. To avoid this problem, the events are *flat* written to a store, which means instead of being broken into primitive objects and written to multiple records, they are written to a single record. Instead of creating a StructuredStream and writing the object to it, the Store uses its stream factory to create a data format specific ObjectOutputStream and writes the object to a single record. The deep, flat and shallow reading and writing options are implemented by the Store class itself, concrete sub classes need only implement methods for writing, reading and removing named records.

5.6.4.1 Null Streams

One potential problem with shallow writing objects, is that the member objects may not exist in the Store when the shallow written object is subsequently deep read. This problem is encountered in the prototype implementation when an item becomes persistent. The item's state must be written to a Store and the reference to the object in the environment must be written to the Store. To achieve this, the item is deep written and the environment is shallow written, to ensure that the environment reference is persistent, but to avoid writing all the environments items to the Store, some of which should not be persistent. When the environment is subsequently deep read, some of the referenced items will not exist in the Store, but this should not be an error – the non existent items were not persistent. To deal with this, the StructuredInputStream creates a NullObjectInputStream whenever a record cannot be read from the Store. The NullObjectInputStream returns default values for every get*() request the object requests. When an invalid class name is returned for the object Serialisable notices the error and informs the parent object. The parent object can then decide what to do. If the member object was essential for continued operation it might exit with an error state, if it can continue without the member it may continue its deserialisation. To solve the problem with Environments referencing objects for which no persistent record exists, the List readObject implementation was modified to discard any list elements which could not be read, but continue reading further elements. When the environment attempts to read its list of items it will thus correctly read all persistent objects in, but not restore transient objects which do not exist in the Store. In addition to solving the problem of transient objects, the NullStream behaviour of StructuredInputStream makes MASSIVE generally more resilient to serialisation errors. It gracefully degrades, just losing the erroneous items rather than failing completely.

5.6.5 Summary

While the Store implementation and particularly the structured streams implementation may appear complex, they encapsulate that complexity to provide useful facilities via very simple interfaces. Complexity is moved into the framework. From the point of view of the event filters and applications, Stores provide a very easy way to read and write objects to a variety of locations and to subsequently read or update part or all of composite objects. From the point of view of future Store implementers, only a very small associative store interface needs to be implemented, the work of breaking complex objects into primitive objects is already done. Finally the many Serialisable classes in MASSIVE-3 are not even aware of Stores. A store looks just like another ObjectOutputStream or ObjectInputStream. This last point was very important in my implementation of a prototype system, although the task of separately storing the members of composite objects using the existing MASSIVE-3 serialisation mechanism was complex, having to change or replace the serialisation code of the dozens of MASSIVE-3 classes would have taken much longer.

5.7 Conclusions

In parts the prototype implementation is a clean implementation of the framework presented in chapter 4 and in parts it is obfuscated by the need to build on the existing MASSIVE-3 platform. The implementation of the core EventPipe and EventFilter class hierarchies present a clear and simple example of how the complex filter ordering, constraints and processing optimisations described in Chapter 4 can be implemented. Although the wrappers for legacy filters are needed solely to provide backward compatibility with MASSIVE-3 they also present an elegant solution to the problem of refactoring the existing functionality of a virtual environment platform to function in a EventPipe architecture.

Most of the unnecessary complexity in the implementation comes from the desire to reuse MASSIVE-3's serialisation mechanism to drive persistence. It was a very attractive implementation decision as the abstract ObjectInputStream and ObjectOutputStream interfaces invited new implementations which serialised objects to new places, such as persistent stores. It was only on closer inspection that although MASSIVE-3's serialisation architecture foresaw the desire to serialise data to new destinations, the need to break serialisations into smaller pieces was not seen. In some senses this is understandable as a schema mapping is more often used where objects are stored in a structured way. The prototype implementation stretched MASSIVE-3's serialisation architecture almost to breaking point and this introduced a lot of complexity, but the implementation did allow hundreds of lines of existing MASSIVE-3 serialisation code to be reused. If the serialisation code in MASSIVE-3 was automatically generated from class definitions (as is the case in the MASSIVE-2 (Greenhalgh and Benford, 1997) and EQUIP platforms) then the generation script could have been easily changed to accommodate the need for structured storage. Unfortunately, the manual implementation of serialisation in MASSIVE-3 made such changes very labour intensive. Nevertheless, the challenge of subverting serialisation mechanisms to is likely to be encountered if persistence is added to many collaborative virtual environment platforms and so the lessons learned here and techniques employed may be an aid to further implementers in the future.

The major advantage of modifying the MASSIVE-3 system to prototype the framework proposed by this thesis is that the facilities of the original

MASSIVE-3 platform could be used to evaluate the framework. The next Chapter uses MASSIVE-3's record and replay facilities (Greenhalgh et. al., 2000) to rapidly evaluate many of the infrastructure configurations made possible by the framework.

6 Evaluation

This chapter presents several experiments which aim to demonstrate that the prototype implementation presented in Chapter 5 can be used to realise the hypothetical optimisations presented in Chapter 2 and Chapter 4. Having designed a model to support per-item infrastructure management and implemented a prototype platform which realises the model, the most important questions are: Can the predicted savings be made? Does the fundamentally different treatment of items in a virtual world make a difference and if so how much difference?

Section 6.1 describes the important issues and challenges which must be faced in the evaluation, and presents the experimental platform developed. Section 6.2 then presents the first experiment which aims to show that deep behaviours can be used to directly optimise the performance of a CVE system. The experiment takes the hypothetical optimisations identified in Chapter 2 and explores the extent to which they can be realised using the prototype implementation. Section 6.3 presents an experiment which repeats the approach taken in section 6.2, but applies it to a different application to see if the optimisations identified in Chapter 2 are more generally applicable. Section 6.4 describes a third experiment which aims to realise the indirect optimisations presented in Chapter 4 by using deep behaviours not to directly drive the infrastructure but as hints to inform a caching process. Finally section 6.5 draws some conclusions from the evaluation.

6.1 Background

The task of evaluating the deep behaviour framework is challenging for a number of reasons:

- The different item roles in a virtual environment application must be identified. In the case of the exploratory experiments described in Chapter 2 this activity was very time consuming and could only be done retrospectively once activity had been logged.

- For each identified role an optimal (or at least good) set of infrastructure mechanisms must be formulated along with the optimal parameter settings for those mechanisms. The many configurations possible with the framework means that combinatorial explosion makes the problem space in which to search for an optimal configuration very large.

- A set of criteria must be identified which can provide measurements to compare the optimised per-item infrastructure mechanisms with a uniform treatment of all items in the virtual environment.

- In order to make a fair comparison the measurements should be taken over a long period of use to alleviate the possibility that freak activity taking place during the measuring could colour the results.

- The infrastructure mechanisms should be the only differences between the conditions used to measure the performance of each approach. Over a long period of measurement it might be hoped that the activity would tend to a common, so over all the test conditions would be comparable. However, ideally each infrastructure mechanism should experience exactly the same events at the same intervals.

While these requirements appear challenging when viewed separately, a number of them are also potentially conflicting: identifying the roles in the virtual environment and the optimal set of infrastructure mechanisms and parameters implies an interactive process of experimentation and modification, whereas the desire for long periods of use implies that an extended period of testing must take place before any mechanism can be judged optimal.

6.1.1 Approaches

In light of these challenging requirements, several approaches for comparison were considered.

The initial intention was to evaluate the framework by running a persistent virtual environment for an extended period of time initially using uniform infrastructure mechanisms, but progressively identifying roles and optimising infrastructure mechanisms. After reviewing the requirements presented above it was clear that this approach was impractical. The experiment could take far too long or result in optimal mechanism configurations not being found. In addition the activity experienced by different infrastructure configurations would vary.

A second very attractive approach would be to simulate activity in the virtual environment and test different infrastructure configurations within the simulation. This approach has the advantage that the simulation can be run at high speed, allowing configurations to be quickly tested and then modified. The disadvantages are that it requires the construction of a simulator and that the results of the experiment would be largely dependant on the quality of the simulation. Any doubts as to the realism of the simulation would result in doubts about the findings of the experiment.

A third approach, and the one used in the experiments, was to leverage MASSIVE-3s record and replay facilities (Greenhalgh et al, 2000) to combine the authenticity of real use with the controlled conditions of a simulation approach. When the MASSIVE-3 system is running a recording mechanism can be used to record the activity experienced by a particular environment replica to a *history file*. The history file contains a checkpoint of the state of the environment when recording started, and the sequence of time stamped events which were experienced by the environment during recording. Setting an environment to the state described by the checkpoint in the history file and then injecting the sequence of events into the environment at the appropriate intervals, previous virtual activity can be replayed. By replaying the same recordings through virtual environment systems configured in different ways, each configuration experiences exactly the same events and so allows a fair comparison of each approach. By replaying the recordings at high speed further advantages of the simulation approach can be realised allowing the large problem space created by the different possible mechanisms and parameter sets to be explored rapidly.

6.1.2 Item Roles

The approach taken to identifying roles in the virtual environment was to rely on the geometry URL of an artefact as this proved the greatest distinguisher of item role in the exploratory experiments. Along with an entity a geometry item is the only item type which can be relied on as existing in an artefact's sub-tree within an environment's scene graph. While some complex interactive artefacts may include behaviours or entities representing separately articulating sub-parts, some may not. All artefacts that can be observed and manipulated by users include at least one geometry item. Sub-trees in the scene graph which do not include geometry items cannot be manipulated and so can be ignored. By providing a mapping between one or more geometry URLs and a deep behaviour configuration the artefacts in the virtual world can be split up into a number of roles which are treated differently. Replaying the same recording multiple times using different mappings from URL to configuration allows different configuration sets to be tested.

6.1.3 Persistence

Although the prototype implementation allows the per-item customisation of mechanisms for consistency, distribution, persistence and access control, the measurement of persistence is the easiest optimisation to quantify, for example by measuring the bandwidth of data written to the persistent store. In addition the analysis of the exploratory experiments revealed configurations which should provide significant optimisation. These configurations are both a promising starting point for the experiments and allow the hypothesise made after the initial experiments to be tested. Concentrating on the well known parameters of persistence mechanisms reduces the complexity of the problem space to be addressed to more manageable proportions.

6.1.4 Recordings

A potential problem with concentrating on the hypothetical optimisations discussed in chapter 2 is that the experiments would not be able to claim that per-item infrastructure mechanisms generally optimised the performance of collaborative virtual environment platforms, only that potential optimisations noted through the analysis of a particular application can be realised. In order to determine whether the hypothetical optimisations applied only to the exploratory experiments or could generally be realised here in other applications, the same configurations were tested using multiple sets of recordings. As the corpus of recordings made in MASSIVE-3 grows the use of record and replay as the basis of an evaluation methodology becomes more and more attractive as systems can be tested with greater amounts of realistic data from an increasing range of applications.

6.1.5 Experimental Platform

The original replay mechanisms were designed to support the concept of temporal links (Greenhalgh et al, 2000) which allow a recording to be played back in a replay environment which can be linked to like any other locale. By linking a live environment to a replay environment, visitors to the live environment can move around the replay to view it from any angle. Although temporal links provide a very flexible mechanism for viewing recordings from inside live virtual environments they do not support the concept of "altering the past" by interacting with the replaying environment. For this evaluation we wanted the system to respond to the events reintroduced into the system as if they were live events. Consequently, in order to perform the experiments, the MASSIVE-3 record and replay mechanisms had to be modified to allow the replay Environment's scene graph to be modified during replay and to allow very high speed playback.

6.2 Direct Optimisation

The goal of the first experiment performed with the prototype implementation was to see if it could support the different roles of items in a virtual world identified in Chapter 2. The analysis of the exploratory experiments showed that items in the experiments were treated by users in a variety of different ways and that treating the items differently at an infrastructure level could make potentially large savings. Typically users would create an item then immediately manipulate it. This resulted in a burst of updates at the start of an item's life during which time it was far more volatile than normal. The analysis showed that delaying an item's time to persistence would save a large amount of traffic to the persistent store for very little impact on the items durability. It is this hypothesis which is tested by this experiment. In section 6.2.1 the method is described, before section 6.2.2 presents the results.

6.2.1 Method

The experiment measured the number of bytes written to the persistent store when a number of different item groups were made persistent. The groups were: all items in the environment, the non-embodiment items, only the embodiment items, and each *class* of item (items having the same non-embodiment geometry URL). For each of these item groups the time from being created to being made persistent was varied from 0 to 1000 seconds. The combination of an item group and a time-to-persistence specified a configuration. For each configuration the recording of each session from the exploratory experiments was replayed using the modified replay application, a total of 250 MB of recordings. The replay application checked the persistent store's auditing statistics after each event was replayed. When these statistics changed, the replay application logged the new byte count and the timestamp of the replayed event. Each individual persistent write could then be analysed, or the logs aggregated to provide an overview of the results.

6.2.2 Results

Figure 6-1 shows the total number of bytes written to the persistent store (by the ServerPreMulticastPersistence deep behaviour) after processing all of the exploratory experiment recordings. It clearly shows that the savings hypothesised after analysing the exploratory experiments can be realised. Excluding embodiment items from the set of items being made persistent reduces the traffic to the persistent store by nearly 66% from 6×10^7 bytes to just over 2×10^7 bytes. This large reduction is despite the fact that the 84% of the items in the virtual world are still being made persistent. It is because of the big difference in the activity of items in embodiments compared to added items that this saving is possible. Entities which represent embodiment positions are continuously updated while the user is present in the virtual world and moving around. Entity items representing an embodiment's head are updated every time the user moves the mouse in order for the head to reflect the users "gaze direction" implicit in the position of the mouse cursor. Entity items representing an embodiment's hand are updated continuously as the user manipulates artefacts in the virtual world. It is because of this extremely volatile behaviour that the removal of so few embodiment items causes such a large reduction in data traffic. By not making the 114 embodiment entity items persistent (38 embodiments each with an embodiment entity, head entity and hand entity) a saving of 4×10^7 bytes is made. By comparison, the 596 added artefacts each with a single entity result in 2×10^7 bytes of traffic. The cost of making every update to an embodiment entity item persistent is 3.5×10^5 bytes and the total cost of each embodiment around 1×10^6 bytes of data, while the cost of making every update to an added artefact persistent is 3.3×10^4 bytes. Clearly it is worth distinguishing between the embodiment and added item roles.

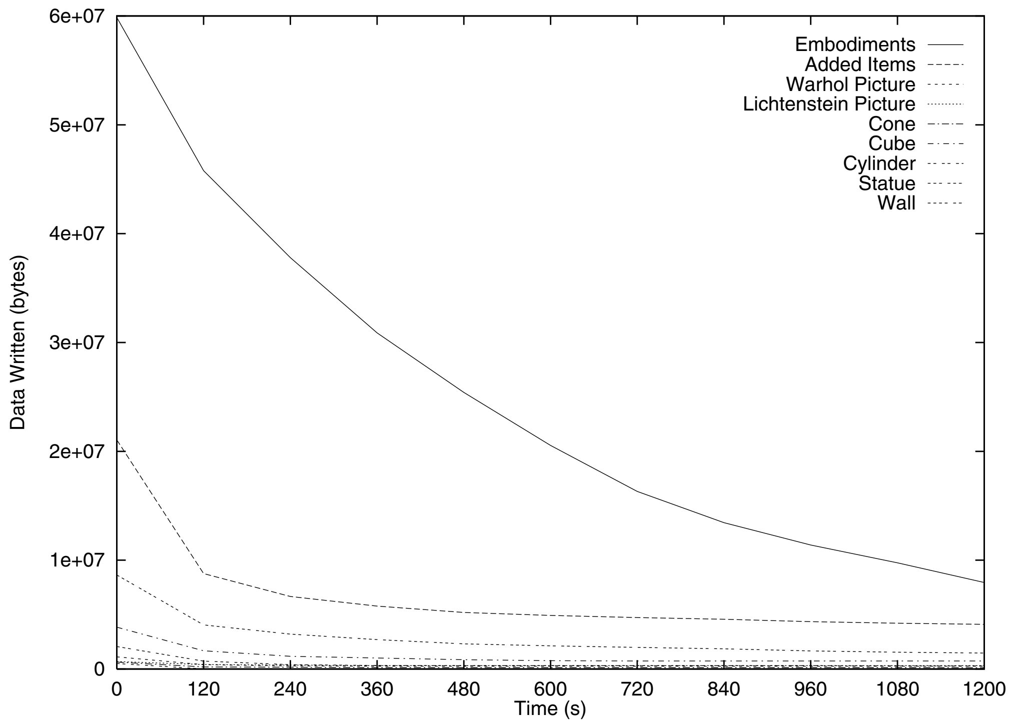


Figure 6-1 Persistent data traffic versus time-to-persistence for the exploratory experiment data.

The first optimisation does not affect the durability of the virtual environment as the state of embodiment items cannot be meaningfully restored in the case of a crash. If the persistent state of an embodiment item is recovered from a checkpoint after a failure the embodiment remains an empty shell until it is re-inhabited by a reconnecting user, something which may or may not occur. In fact the presence of "empty" embodiments caused as users connected to BT's Ages of Avatar virtual world (Benford et al, 1998) caused a great deal of confusion. Allowing embodiments to exist in a world without an associated user undermines the role of an avatar to represent a user's presence in the virtual world. As such embodiments should not be made persistent and the large savings noted above can be made without impacting reliability.

Figure 6-1 also shows that further large optimisations can be made with limited impact on the durability of the virtual environment by delaying the addition of the ServerPreMulticastPersistence deep behaviour, and so the persistent storage of updates to the item. If added items are made persistent only after a 120 second delay, then the traffic to the persistent store is reduced to less than 1×10^7 bytes – less than 50% of the total written if added items are made persistent immediately and less than 17% of the total written if all items are made immediately persistent. The reason for the further reduction in traffic is because of the turbulent initial period of activity experienced by most of the added items in the exploratory experiments. Most of the items were added to the virtual world and were repeatedly updated as they were moved into their initial positions. After this period most items were updated much less frequently or were not updated at all. We hypothesised that large savings could be made by identifying this typical life cycle of an object and using deep behaviours to tailor infrastructure mechanisms. Figure 6-1 shows that these savings can be realised. Note that increasing the delay beyond 120 seconds gives greatly diminishing returns. After the initial burst of updates, most items are updated very infrequently and so increasing the time to persistence does not save much persistent store traffic, but does increase the chance of updates being lost through failure. Ideally, each item should be made persistent immediately after it has been moved into its stable initial position, but not before. Although this moment comes at a different point in each item's lifetime, 120 seconds is a good approximation for this data set.

Although delaying an item's time to persistence can reduce the traffic to the persistent store, it also allows updates to be lost or the very existence of items to be lost if a failure occurs during the delay period. While this loss is very limited in terms of the state of the persistent virtual environment as a whole it could potentially be disorienting for users who have already *experienced* updates and see the world return to a previous state after a failure. These semantics are commonly encountered in failure recovery via backup systems, where data not yet backed up is lost, however it can lead to confusion in human in the loop systems, such as collaborative virtual environments, where changes are experienced and then disappear. In addition, in applications such as on-line world editing, it is the items which are being manipulated at the time of the failure which are lost. The framework allows a number of solutions to this problem, some of which are presented here. The items manipulated by a user could be made persistent at the client. In the event of a server failure the state stored at the client could be merged with the world on reconnection.

Certain important items such as walls could be made persistent immediately, while other items use delayed persistence. Users could explicitly mark an item as completed when it had been moved into its initial position, allowing persistence to be applied from that moment.

6.3 Optimisation Generality

It is extremely encouraging that the hypothetical optimisations identified in Chapter 2 can be realised, however if these optimisations only apply to the museum experiments then the usefulness of the deep behaviour framework is extremely limited. For example an analysis similar to that of Chapter 2 would have to be performed for each application, to identify the item roles, their characteristics and appropriate deep behaviours for each set of items. If, however, the optimisations identified after studying the museum experiments could be applied to other applications, then the framework is demonstrably more generally useful. For example, suites of optimisations, heuristics and deep behaviours could be developed and then generally applied to applications without detailed analysis of each application being needed. To determine whether the optimisations developed above apply to different applications the experiment was repeated using a set of recordings made of the activity in the Avatar Farm experiment discussed in section 6.3.1.

6.3.1 Avatar Farm

Avatar Farm represents a very different type of application to the exploratory experiments. The exploratory experiments used a persistent virtual environment which concentrated on on-line world development; Avatar Farm focuses on parallel streams of on-line narrative. The activity in Avatar Farm consists of 7 actors using desktop and immersive VR clients to enact scenes in 4 virtual worlds explored by four members of the public who both watch and participate in the on-line narrative.

6.3.2 Method

The method used in this experiment was identical to the method discussed in section 6.2. The only differences were that the Avatar Farm recordings were used and new geometry to deep behaviour mappings were used. Because avatar farm used a far greater number of geometry URLs than the exploratory experiments, only mappings to all items and non-embodiment items were defined.

6.3.3 Results

The total bytes written to the persistent store when all items in Avatar Farm are subject to the ServerPreMulticastPersistence deep behaviour and when only items with the Non-Embodiment role are made persistent are shown in Figure 6-2 (note use of log scale necessary for data written axis). It is apparent that one optimisation has transferred well to the Avatar Farm application and the other has not. The gulf between embodiment items and the other items is even greater than before. If all items in the environment are made persistent immediately then 1×10^8 bytes are written to the persistent store. If only non-embodiment items are written to the persistent store then only 2×10^5 bytes are written to the store. By excluding avatars from the set of items which are written to the store after each update the traffic to the store can be reduced by three orders of magnitude. Once again this saving is essentially cost free, as the recovery of avatar positions from the persistent store does not make sense. In order to make the Avatar Farm application persistent only 2×10^5 bytes need to be written to the persistent store, but if all items are treated equally 100 MB of unnecessary data is written to the store. However, the time-to-persistence optimisation fares less well. Although it was able to further reduce the traffic to the persistent store by 50% by waiting 2 minutes when replaying the exploratory experiment data, with the avatar farm data a much smaller reduction is made. With the exploratory experiment data a slight delay in persistence avoided a large, early burst of updates for a slight reduction in the durability of the world. In the case of the Avatar Farm experiment a slight reduction in persistent data traffic can be realised for a steadily increasing delay in the time-to-persistence. The explanation for this behaviour is simple: in the exploratory experiments the nature of the on-line editing application meant that users generally created items and then edited them, resulting in the early burst of updates, whereas in Avatar Farm users manipulated existing items as part of a narrative. The narrative application resulted in fewer updates to non-embodiment items and caused updates to occur to the items at appropriate moments in the narrative, which might be arbitrary points in the items lifetime, rather than consistently early in the items lifetime. The experiment is successful in that it shows which optimisations identified in one application can apply to another, but also highlight that some optimisations might only apply to a sub-set of applications.

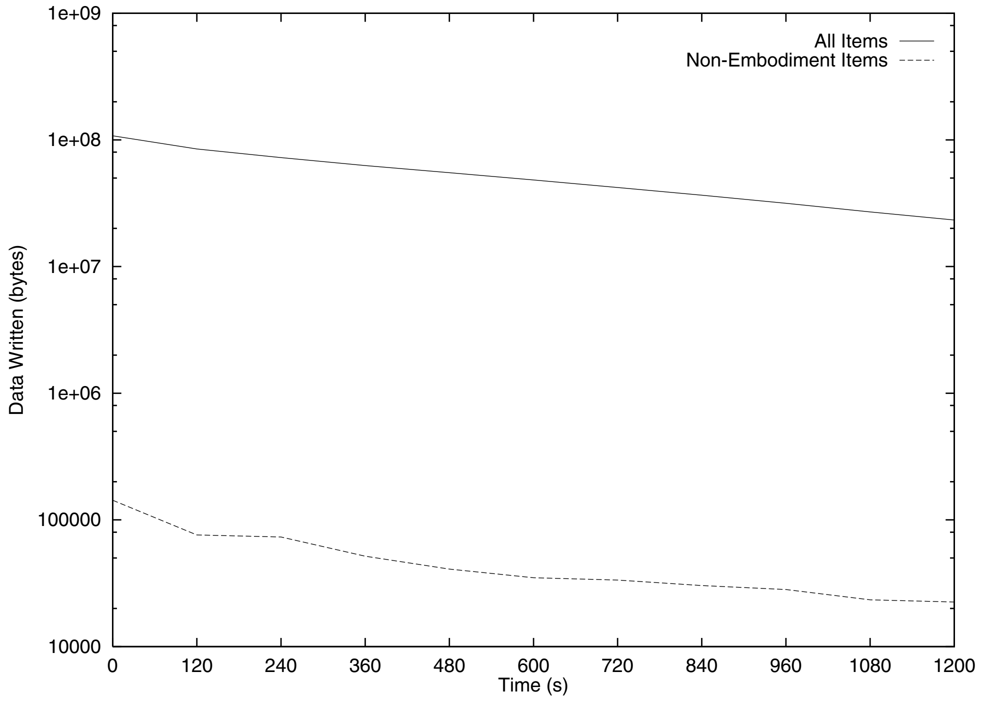


Figure 6-2 Persistent data traffic versus time-to-persistence for the Avatar Farm data.

6.4 Indirect Optimisation

The previous sections show that identifying the different roles which items in the virtual environment perform and treating each role appropriately can optimise the performance of infrastructure mechanisms in collaborative virtual environments. In addition it shows that at least some of the roles and optimisations identified in one application can potentially apply to other applications, making it possible to at least partially reuse analyses of virtual environments and identify common patterns of behaviour between applications. The previous experiments show that deep behaviours can be used to optimise system performance by specifying mechanisms and parameters directly. The experiments described in this section show that the presence of deep behaviour annotations allow infrastructure mechanisms outside the scope of a particular deep behaviour to be optimised. The deep behaviour annotation can be regarded or interpreted as a more general form of meta-information which allows infrastructure mechanisms to distinguish between items and so treat them differently. For example while the TrustedPersistence deep behaviour (described in Chapter 4) directly specifies a consistency and persistence mechanism, a caching mechanism might use this deep behaviour annotation to deduce that the item is not going to change often and is an important item worth caching. By using deep behaviours in this way there does not need to be a 1-1 mapping between deep behaviours and optimised infrastructure mechanisms. A complex caching mechanism might assign a value to each deep behaviour and attempt to cache high value items preferentially. A simpler caching mechanism might attempt to cache items annotated with a certain deep behaviours in preference to all other items. It is this second approach which is evaluated in the experiment described here.

6.4.1 Method

In order to see if the hypothetical improvements to the performance of caching could be achieved, the test harness application was modified to simulate a client application. When joining a virtual world the application first looks in its cache for items before requesting their state. This mechanism both improves the speed at which a virtual environment can be viewed (analogous to web browser caching) and potentially allows parts of the virtual world to be viewed off-line or during periods of disconnection. Like any cache, however, the number of items which are stored cannot be unbounded, and so a cache replacement policy must be implemented to decide which items are removed from the cache when it is full. The experiment compared two approaches to caching: the Least Recently Used (LRU) caching algorithm (widely used in operating system page caches) and a novel approach which uses the LRU mechanism for cache replacement decisions, but only caches items annotated with certain deep behaviours. This novel approach was dubbed Selective LRU (SLRU) caching.

Both caches were implemented by maintaining a linked list of items in the cache ordered by the sequence of accesses to the items. Accesses were defined both as item additions and updates. As the environment is initially spooled to the client the cache sees a sequence of add events and fills the cache with the contents of the environment. Subsequent additions are prepended to the list and where they increase the list size over a defined threshold the items at the end of the list are removed from the cache. When an item in the cache is updated it is moved to the start of the list so only the least recently updated items are removed when the cache grows. The LRU cache performs this processing for any access, the SLRU cache examines the scene graph at each access; and only performs processing for items annotated with a specified set of deep behaviours. The cache state is made persistent in a Store allowing the previous state of the cache to be read when the application starts and before the environment is spooled. The performance of the caches are measured by querying each cache when the new environment is spooled to see how many items in the virtual environment exist in the cache. This method simulates the behaviour of a user returning to the environment they have previously visited. If the client cache performed well during the previous visit most of the items which were cached will still exist in the virtual environment. The recordings of each session of the exploratory experiments were replayed through the test application in chronological order and the performance of the two caches measured at the start of each replay. This process was repeated for maximum cache sizes ranging from 10 items to 1000 items at which point the performance of both caches levelled out. The measurements recorded the current maximum cache size, the total number of items in the virtual environment, the number of cache hits achieved by each cache, the cache utilisation of each cache and the activity of each cache.

6.4.2 Results

Figure 6-3 shows the percentage of successful hits achieved by the LRU and SLRU caches for a range of cache sizes. It is clear that the SLRU cache provides a significant performance advantage over LRU caching. At the smallest cache size of 10 elements, SLRU successfully services 1.75% of cache queries where LRU only manages 0.52%, less than 30% of the performance. When the cache size is increased to 200 elements, 15% are returned by SLRU and 7.2% by LRU. At 500 elements the differentiation increases with SLRU servicing 25.5% and LRU 7.8% - LRU again achieves less than 30% of SLRU performance. When the cache size approaches 1000 elements both caches contain the majority of the virtual environment and so the gap in performance narrows. Increasing the cache size beyond 1000 elements results in no performance gain, with both caches servicing 95% of queries. When used to cache items in a large-scale virtual environment a client cache could not possibility contain all of the items. While current virtual environment systems support arbitrarily large environments, the storage and processing capabilities of a single client node are fundamentally limited. For this reason the SLRU cache is a significant improvement over LRU caching for the situations normally encountered in collaborative virtual environments, where the total size of the environment is much larger than the maximum possible client cache size.

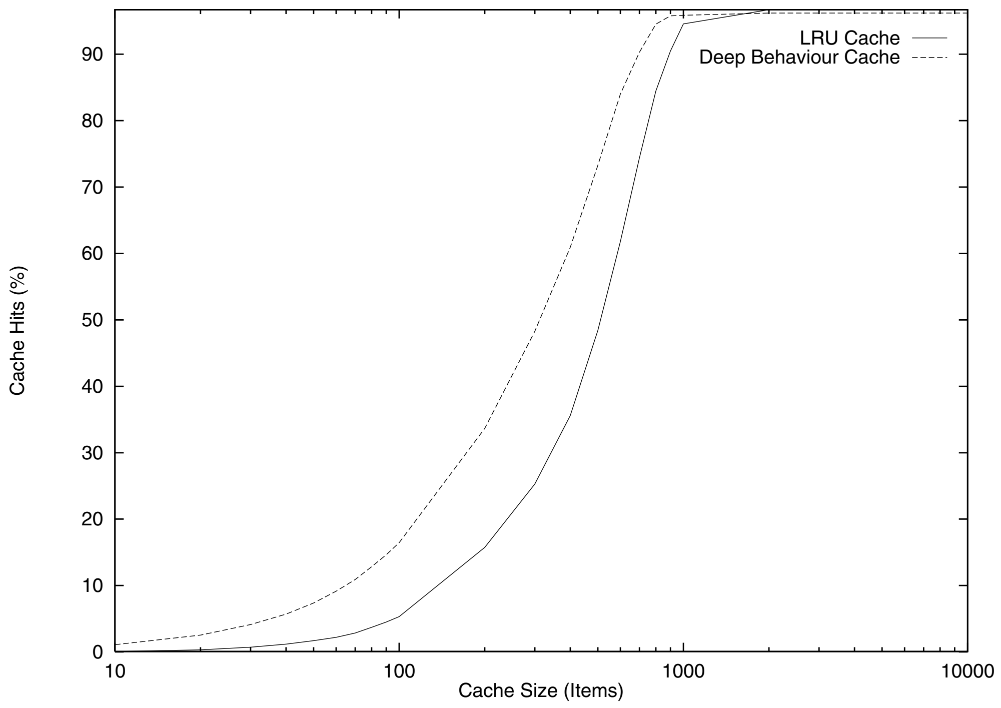


Figure 6-3 Hit rates for LRU and SLRU caches with 10 to 1000 element caches.

In addition to providing superior cache performance for a given cache size, the operation of SLRU caching is also more efficient than LRU caching due to its selective nature. Although SLRU caching must perform more processing on an access to determine whether to cache an item or not, that cost provides great savings both in the storage space utilised by the cache and the work done serialising items to the cache. Figure 6-4 shows the number of times the LRU and SLRU approaches replaced items for maximum cache sizes ranging from 10 to 1000 items. The most noticeable difference between the performance of the two caches is for a cache size of 10 items. The LRU cache performs 66892 writes compared to 10255 writes performed by SLRU. This large discrepancy is due to the LRU cache being too small to hold all of the rapidly changing embodiment items in the environment and so thrashing as items are removed from the cache and then replaced. The SLRU approach does not suffer from this problem as it does not cache the un-annotated embodiment items. With cache sizes between 100 and 1000 items both caches replace items less frequently as the caches grow and so are more likely to contain an accessed item. During this period the SLRU cache replaces 30% to 60% fewer items as it is not affected by volatile embodiment items. When the cache size reaches 900 items, the activity of the SLRU cache levels out as it contains all of the annotated items in the environment and so never replaces items in the cache.

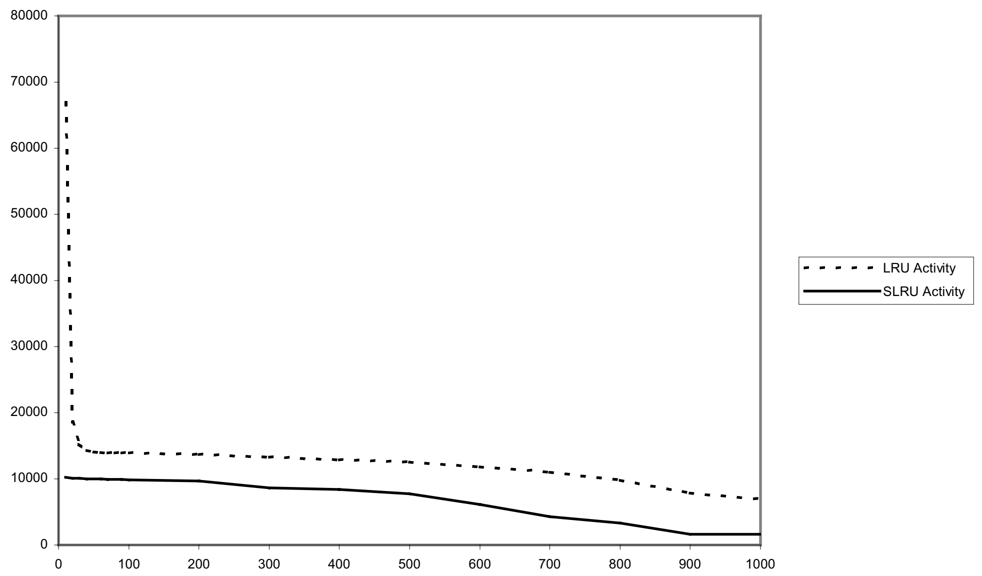


Figure 6-4 Cache activity for LRU and SLRU caches with maximum cache sizes from 10 to 1000 items.

Figure 6-5 shows the storage space utilised by the LRU and SLRU caches for maximum cache sizes of between 10 and 100000 items. For maximum cache sizes up to 100 items, both approaches utilise the entire allowed storage space, with LRU using the maximum space up to a maximum cache size of 300 items. Between 100 and 2000 the space used by both approaches grows, although the SLRU growth is slower. At 2000 SLRU reaches its maximum usage of space and contains 842 items compared to LRU which contains 1689 items. SLRU is utilising 42% of the cache and 84%, twice as much storage space is being used by LRU. Between 2000 and 5000, LRU continues to use more storage space for no increase in performance before finally plateauing at 2680 items. With the maximum number of items set to 5000, LRU uses 3 times as much storage space as SLRU for a 0.4% advantage in cache hits.

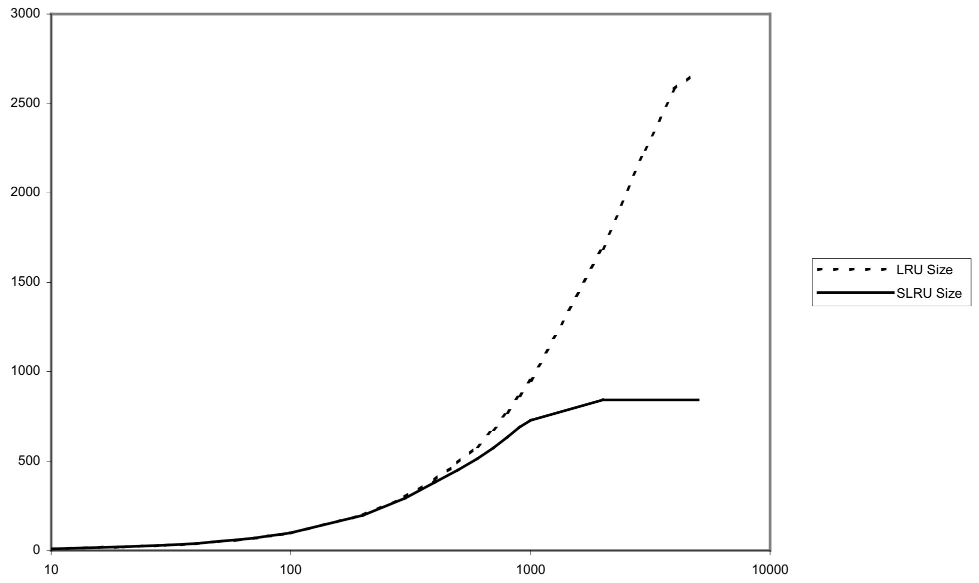


Figure 6-5 Utilised cache size for LRU and SLRU caches with maximum cache sizes 10 to 5000 items.

6.5 Conclusion

This chapter has shown that the hypothetical optimisations presented in Chapter 2 and Chapter 4 can be realised using the prototype implementation. The experiments have demonstrated that per-item infrastructure management can significantly improve the efficiency of collaborative virtual environments. By acknowledging that there are very different roles performed by items in a virtual environment and by tailoring the treatment of those items to their requirements the best mechanisms and configurations can be utilised by all items. CVEs have historically treated all items equally and ended up searching for a best compromise solution to infrastructure challenges. In contrast a per-item approach to infrastructure allows a tailored solution to be used for each item and allows multiple approaches co-exist in a single CVE.

In addition to fulfilling its primary goal of demonstrating the improvements which can be realised using the deep behaviour approach, conducting these experiments has allowed the development and refinement of a promising methodology for rapidly testing and evaluating infrastructure mechanisms for CVEs. By combining the highly dynamic and reconfigurable nature of the deep behaviour framework with the realistic and repeatable test conditions provided by record and replay novel mechanisms can be quickly prototyped and tested in realistic situations. Many of the existing event filters can be re-used to implement novel mechanisms through their orchestrated operation. Radically different approaches to consistency can be implemented simply through the configuration of event pipes and routing filters and the ordering of existing filters to update various replicas of an environments state. Composite mechanisms such as the TrustedPersistence deep behaviour (described in chapter 4) can be realised by combining existing mechanisms. Alternatively semantically independent mechanisms can be used in partnership and their combined effects analysed. The ability to replay activity in virtual environments at speeds limited only by the performance of the replaying machine allows mechanisms to be subjected to significant amounts of testing in relatively short periods of time and for near optimal parameters for mechanisms to be determined in acceptable time frames. As the corpus of recordings grows this methodology will become increasingly valuable as it will allow mechanisms to be tested in a wide range of situations.

7 Conclusion

This chapter concludes this thesis by detailing the contributions made, the future directions the work could take, and makes some final remarks which place this work within current trends in computer science at large. Section 7.1 details the original contributions which this work has made to the field of collaborative virtual environments and Computer Science. Section 7.2 then outlines a number of areas in which this work could be extended, both technically and via complimentary work in the fields of Sociology and Human Computer Interaction (HCI). Finally section 7.3 attempts to paint a picture of the future of large-scale persistent virtual environments and the role of this work in that future.

7.1 Contributions

The original contributions made by this thesis can be viewed from a number of perspectives. Section 7.1.1 looks at the philosophy of this thesis. It compares the views, approach and priorities of this work with the accepted views shown by previous CVE research to identify the new approaches taken by this thesis. Section 7.1.2 details the contributions made to the theory of CVE research and the field of Computer Science. These contributions are mainly in the development of the model described in Chapter 4 and the two concepts of distributed event filters and deep behaviours which it introduces. Section 7.1.3 looks at the contributions made to the field by the realisation of the concepts introduced in the thesis. It discusses the large amount of implementation performed in terms of its use as a proof of concept, but also as an extremely flexible framework for rapidly prototyping future CVE systems.

7.1.1 Philosophy

This work has proposed a philosophy which is significantly different to the established approach to Collaborative Virtual Environments. Previously the critical activity in virtual environments was “doing”. The development of applications and environments was an off-line necessity which formed a precursor to doing. Previous work focused on allowing more people to do things, different things to be done and for things to be done in higher fidelity. This work shifts the focus by proposing that the critical activity in collaborative virtual environments is being. By being constantly available a virtual environment can be relied on as a place where communication and collaboration can take place. It can become an alternate place where community can be built and then all of the activities which were previously the focus of virtual environment research can take place. The emerging breed of on-line massively multiplayer games highlight the importance of this community over content approach. While the games strive to create a virtual world which is high-fidelity, highly scalable and highly interactive, the most important goal is that the environment supports community. The environment needs to be a place people want to be more than a place where people want to do. Any barriers to that are fatal. A system which shuts down every day to provide people with more to do at the expense of providing a place for people to be is likely to fail.

This may sound like an argument to stop working on the technical aspects of CVEs and concentrate on the sociology of community building; it is not. While the community building aspect of CVEs is very important, the environment must be able to grow and change with the community it supports. While an attractive meeting space with welcome banners might be a perfect venue for members of the new community to get to know each other, it might not be perfect a month later when the community decides to get down to work, or play. In order to avoid the frustration of a world which cannot grow with its community and the frustration of a world which is not continuously available, future CVE systems must be extremely flexible. As this thesis has shown, engineering the flexibility needed by these environments is a major technical challenge. While CVE systems may be able to support limited evolution by providing on-line tools for world editing or by allowing new application level behaviours to be introduced into the system there is still the possibility that the limits of the system's flexibility will be reached. For example most of the current CVE platforms could not cope with an environment which needed to evolve to support secure transactions for e-commerce. Their infrastructure assumes the need for optimistic, timely interaction over reliable transactions.

In most cases, the best that could be done is that a *hole* could be dug in the CVE system to allow an e-commerce system to take over at the appropriate moments. Clearly this is not an ideal solution. The question of how much flexibility these systems will require is impossible to answer. Rather than trying to predict the flexibility required in the future, an alternative approach is to attempt to engineer ultimate flexibility - a platform which will be able to cope with any future demands and unforeseen changes which are required. While this initially appears as daunting as predicting where flexibility will be needed, any Turing complete language can – at least theoretically – provide an amount of flexibility limited only by what is programmatically possible. It is clear, though, that a Turing complete language is quite a long way from the ideal next generation CVE platform. For a start most languages do not support run-time loading of new modules. Adding this basic requirement results in frameworks like Bamboo, where nothing is assumed except the need to load modules. While this framework is sufficient for building continuously available, flexible CVE platforms, there is clearly a lot which is left to the application developer. Adding the assumption that all CVE systems rely on events routed between nodes brings us to the level of the framework discussed in this thesis. The framework is only a little less flexible than Bamboo, but provides support for the notion of networking which significantly eases the burden on the application developer. The approach taken by both of these frameworks is to develop a set of invariants and then make sure that everything else is flexible and optional. A first attempt at the set of invariants for CVEs might be the assumptions that they are networked and support multiple users. Everything else cannot be assumed: network protocols, topologies, infrastructure mechanisms and everything else must be dynamic, reconfigurable and flexible. By building platforms like Bamboo and DEF which make different assumptions about invariants the best mix of support and flexibility can be discovered.

Another area in which the approach taken by this work differs from most other CVE systems is in its support for per-item infrastructure mechanisms. Just as nothing can be assumed about the requirements of an evolving virtual environment, neither can anything be assumed about the combination of requirements needed at any one time. The ability to load a new consistency or transaction mechanism to cope with e-commerce requirements in a virtual environments is useless if that consistency mechanism must apply to every item in the virtual environment. The per-item approach is a natural consequence of the flexible, evolving virtual environment platform, but an approach which distinguishes this work from most previous CVE research.

7.1.2 Theory

This work has developed two concepts – distributed event filters and deep behaviours – which add significantly to the collection of techniques in the CVE developer's armory. In addition the techniques differ from previous approaches in that they specifically tackle the problems faced in the development of the emerging breed of flexible, dynamic CVE systems which are needed to support continuously available persistent virtual environments, environments which grow with the communities which they support while remaining constantly available to those communities. In addition to providing specific approaches to engineering the flexibility needed by this new breed of system, this work serves as a guide for the development of future techniques and architectures to support these new flexible CVE platforms. The model presented in this thesis represents a single point in a problem space which trades off flexibility for support. While it is unlikely that this point is the optimal solution for all problems, the process of getting to this point (recorded in chapters 4 and 5) serves as a guide for CVE developers needing to get to a different point in the problem space.

Widening the context from the field of CVEs to that of Computer Science in general reveals different theoretical contributions. Although the pipes and filters architecture used by the DEF framework is widely used in Computer Science, there are a number of features which are novel to the DEF framework (as much as that claim can be made within a field as large as computer science of which the author cannot possibly be completely aware). The support for position and constraints and requirements within event pipes is a novel and extremely useful feature which makes using complex configurations of pipes and filters extremely simple. Rather than having to be aware of all of the filters in a pipe, the application developer needs only to list the constraints which matter to the filter being added and then add it to the pipe. Then when the set of filters changes the constraints can automatically be checked to ensure the validity of the new configuration. The use of a hierarchical naming scheme for filters is another novel feature which provides filter constraints and requirements with a certain amount of future proofing. Filters specifying constraints on classes of filters will behave correctly if new filters in that class are implemented in the future. Similarly new filters can be given a name which matches with an appropriately specific existing filter in order to behave correctly with existing filters.

The caching mechanisms used in the execution of the event pipe are also an innovation which might be useful in other event pipe architectures where nothing can be assumed about the data passing through the pipe after each call to a filter, but where operation should be optimised for the simple case where data doesn't often change.

7.1.3 Realisation

The significant amount of implementation which has taken place throughout the work has produced a working prototype implementation based on MASSIVE-3. This prototype demonstrates a wide range of deep behaviours and by implementing deep behaviours as object annotations in the scene graph allows virtual environments utilising the deep behaviours to be built and used with standard MASSIVE-3 tools. The prototype implementation allows continuously available persistent virtual worlds to be developed and evolve on-line in a way which was not possible with previous CVE platforms. Client applications developed for the experiments described in chapter 2 allow virtual environments to be built and changed on-line using near standard user clients which allow human viewpoint ad hoc modification of the virtual environment. A client application developed to demonstrate the framework allows the on-line creation and modification of artefacts which exhibit a wide range of deep behaviours, allowing the on-line configuration of the virtual environment infrastructure mechanisms (Video Figure 3 demonstrates these facilities). The same tool allows multiple infrastructure mechanisms to co-exist in the virtual world.

While the flexibility engineered into the prototype implementation was primarily to allow the fluid evolution of continuously persistent virtual environments, it also provides a powerful framework for the development of CVE systems themselves. One of the early indications of the power of the framework was the realisation that by configuring event pipes and filters in the extended MASSIVE-3 system it could be made to emulate at least parts of the architectures of the MASSIVE-1 (Greenhalgh and Benford, 1995) and MASSIVE-2 (Greenhalgh and Benford, 1997) systems previously developed in the Communications Research Group. Despite the radical differences between the peer-to-peer unicast and multicast approaches of the early systems and the client-server approach of MASSIVE-3, all three architectures could be realised within the new platform by just reconfiguring event pipes, potentially at run time. This highlights the usefulness of the DEF framework as a tool for rapidly prototyping new systems. Rather than developing an entirely new CVE system to test a new topology or technique, a new configuration of event pipes can be developed to prototype the new approach. Similarly new infrastructure mechanisms can be rapidly prototyped by developing new filters and introducing them into the event pipe network.

The test harness developed for the evaluation described in chapter 6 provides another set of useful tools for the future development of CVE systems. The record and replay based test harness provides a simple but realistic approach to the testing of both techniques and mechanisms. It allows mechanisms to be quickly tested with an existing corpus of recordings before the lengthy process of human testing need be undertaken. Conversely any subsequent human testing can provide further recordings for future tests. By testing entire CVE systems in the same way some of the problems inherent in prototypal systems can be avoided, whereas manual testing is a lengthy process which tends to be avoided in the rapid development of prototype research platforms the test harness allows new code to be rapidly subjected to large amounts of real world use without significantly slowing down the development process.

7.2 Future Work

The scope of this work has been limited to the technical development of a new model for infrastructure processing in persistent collaborative virtual environments. The work has looked at the details of the model to ensure that it scales to support arbitrarily sized virtual environments and numbers of users. It has attempted to ensure that the model is flexible enough to support the widest conceivable range of infrastructure mechanisms. A proof of concept implementation has been developed and then evaluated to assess the hypothetical benefits imagine during the model's development. While the work has produced a working technical solution to many of the challenges facing continuously available persistent collaborative virtual environments, there are clearly many other questions which need to be answered.

Section 7.2.1 presents some options for presenting deep behaviours to users using either real world or deep metaphors. Section 7.2.2 discusses how the model might work with various alternative programming paradigms. Section 7.2.3 then discusses the need for a deep behaviour language before section 7.2.4 lists some tools which are needed before deep behaviours could easily be used in a production environment. Section 7.2.5 highlights some implementation issues which should be resolved if the prototype is used beyond its current role as a proof of concept and finally section 7.2.6 argues that a large scale trial is needed to really test the framework.

7.2.1 User Interface

Virtual environments have historically been less intuitive to use than their emulation of the real world would suggest. Simple tasks such as navigation around virtual environments or artefact manipulation tasks have proved extremely difficult for novice users to perform. A CVE system which complicates matters by making every object in the virtual world behave subtly differently by applying different infrastructure mechanisms to each seems to guarantee even more confusion and bewilderment. However, the questionnaire responses described in chapter 2 suggested that users actually expected artefacts in the virtual world to behave differently in some circumstances.

Some users expected large and important landmarks to remain unchanged, so they could be relied on. Others expected structural artefacts to be more difficult to manipulate than *decorative* artefacts. The deep behaviour model allows these differences to be articulated and respected by the CVE system. However, the development of an appropriate way to present the differences to users is an important challenge. The user interface must clearly convey the differences which users expect without making the virtual environment more confusing.

The use of metaphors has been proved to be extremely valuable in the design of user interfaces, allowing users to understand something alien and complex in terms which are simpler and more familiar. The most common metaphors being the desktop, files and folders used in modern operating systems. Collaborative Virtual Environments and Virtual Reality in general have exploited the use of the real world as a metaphor for communication, visualisation and navigation of complex data and systems.

One approach to presenting the differences in the deep behaviours of artefacts in the virtual environment would be to use real world metaphors. In this case the metaphors should ideally be based on the properties of the object. For example, otherwise identical artefacts might behave differently because of the different substances from which the objects are made. The different properties of the substances such as their density or hardness could be used to explain the behaviour of the otherwise similar artefacts. Some example metaphors are listed below.

- Weight. An artefact's weight and so the speed at which it can be manipulated could be used to represent the persistence or consistency of an object. Heavy objects would be slow to manipulate and so easy to make persistent or keep consistent across multiple replicas. Light objects could be rapidly manipulated, but use less conservative persistence or consistency mechanisms.

- Hardness. Environments which implement the DelayedPersistence deep behaviour (used in the experiments presented in Chapter 6) could visually represent the state of the artefact using a hardness metaphor. Objects could be represented as initially clay like and easy to manipulate after their creation and then harden once they become persistent and more difficult to manipulate. Visually this could be represented using physically based modeling approaches (Baraff and Witkin, 1998) to make the soft artefacts deform like jelly or soft clay.
- Evaporation. Environments using automatic garbage collection of added objects to avoid the clutter experienced in the exploratory experiments might present these mechanisms using an evaporation metaphor. Objects added to the world would initially appear solid, but slowly become more and more translucent until they disappeared completely. Manipulating a translucent object would restore it to its initial state. Manipulations could slow the evaporation process until it stopped completely, or a specific manipulation could remove the garbage collection deep behaviour.
- Locking. Rather than using the physical attributes of an object, real world access control mechanisms could be used to represent and manipulate the deep behaviours which apply to artefacts. Artefacts which can only manipulated by certain users might be visually annotated with virtual locks or keypads, while virtual keys would allow the visual delegation of privileges from administrators to other users.

An alternative approach to explaining the differences between objects using real world metaphors is to use the deeper metaphors approach proposed by Ivan Vaghi (Vaghi, 2001). With this approach, rather than explaining the differences in behaviour as side effects of substance properties, the underlying infrastructure issues are symbolically presented to users. An artefact might be visually annotated with icons to represent different persistence mechanisms or a bar representing the time before an object becomes persistent. This approach is used in the demonstration client as it makes the mapping of deep behaviour to artefact clear and so works well to explain and demonstrate the framework. However, it forces the user to understand the potentially abstract or complex infrastructure mechanisms, and so might be inappropriate for use by novice users. The interface shown in Video Figure 3 is a very simple example of this deeper metaphors approach.

7.2.2 Programming Paradigms

Although this work has been heavily influenced by the model of virtual world as shared, distributed scene graph, the model described in chapter 4 has been deliberately kept independent of this approach. While MASSIVE-3 and the prototype implementation used a scene graph to annotate items with deep behaviours, this relationship could also be implemented by a reference, for example in an object oriented approach such as those used by LambdaMOO and its descendents (for example VWorlds). When the behaviour sharing described in chapter 4 is considered the scene graph approach moves even closer to the object model paradigm. Deep behaviours which annotate multiple objects to provide common functionality appear much more natural if viewed as parent objects with multiple children. Treating deep behaviours as abstract classes in an inheritance hierarchy also makes sense where application level behaviours are composed using an OO style inheritance hierarchy as is the case in LambdaMOO style systems. If deep behaviours are independent of the application behaviour hierarchy then the root of the hierarchy could include references to the deep behaviours used by the object. However, it is unlikely that application behaviours could be kept independent of deep behaviours, so the two hierarchies could be combined into a single run-time inheritance hierarchy in which deep behaviours exist near the root of the hierarchy and application behaviours appear as branches and leaves. The DEF framework could be integrated into the object oriented approach described above by giving each object an event pipe which is populated by the filters needed to implement the behaviours operating on the object.

Alternatively the division between scene graph and filter network which exists in the current prototype implementation makes the architecture look similar to the Message Oriented Middleware (MOM) paradigm which separates systems into application data and communications layer sections.

7.2.3 Deep Behaviour Language

Recent work on object patterns (Gamma et al, 1995) has highlighted the usefulness of naming even very complex or abstract approaches to problems to provide a common language for practitioners to discuss these approaches. A similar approach is likely to be very useful in the development and use of deep behaviours. It is not a trivial task. In the development of the deep behaviours presented in this thesis the formulation of an appropriate name was often difficult. While the TrustedPersistence deep behaviour was easy to name after the problem of trusting predicted results described by Dourish (Dourish, 1996), others became just a concatenation of the mechanisms which make up the behaviour, such as ServerPreMulticastPersistence. The latter approach is less desirable as it highlights the implementation of a behaviour over its semantic meaning. In order to be most useful as abstractions of individual mechanisms they should be named semantically. A user or administrator should annotate artefacts with the semantic role of an item. An item should exhibit persistence which can be trusted by an observer rather than be subject to an arbitrary collection of mechanisms. Future work in this area should establish the correct level of abstraction for deep behaviour names and heuristics for naming them which make the formulation of a comprehensible language of deep behaviours possible.

7.2.4 Tools

For deep behaviours to move from being an interesting research topic to a framework used in the construction and optimisation of production virtual environments, a suite of tools are needed to make the construction, configuration and application of deep behaviours simple.

7.2.4.1 On–Line Deep Behaviour Configuration

The most fundamental need is for a tool which allows the run time annotation of items with deep behaviours. This tool for creating <item, deep behaviour> relationships could be a command line tool which used identity numbers to map items to behaviours, a graphical scene graph which allowed behaviours to be dragged and dropped onto items, or an alternative 3D user client which allowed the visualisation and modification of deep behaviours. The final example could use Ivan Vaghi's techniques for infrastructure visualisation (Vaghi, 2001). Making these visualisations an optional mode of the standard user client would allow users view the deep metaphors when they encountered a problem in the virtual world, make changes to the infrastructure configuration which was causing the problem and then turn off the deep metaphor annotations. When the metaphor of the virtual world was broken by infrastructure issues, the infrastructure could be made visible, modified and then the original metaphor returned to.

7.2.4.2 On-Line Infrastructure Analysis

The other important tool required for the successful exploitation of deep behaviours is an analysis tool to determine exactly how items in the virtual world are behaving, how they differ from each other and so what roles exist in the virtual world. The evaluation chapter presented a primitive off-line version of this tool consisting of the rapid replay test harness and the scripts used to analyse the behaviour of different items. The ideal tool for administrators of persistent virtual environments would be one which performed this analysis on-line, automatically measuring key criteria such as persistent storage bandwidth and consistency, ranking items by criteria and identifying roles in the rankings. The gathered information could be viewed periodically by an administrator and deep behaviour decisions made. The tool could also generate alert messages when an item moves outside reasonable conditions allowing the administrator to take action in the event of exceptional conditions.

7.2.4.3 Automated Analysis and Deep Behaviour Configuration

The development of this tool is clearly a challenging task in itself. However, it could conceivably be taken a step further towards a system which analysed the operation of the virtual world and made deep behaviour decisions automatically. If an item survives a long time without changing it is cached; if it changes often it is only periodically made persistent; if it changes in bursts it is made persistent after a batch of updates and so on. This idea is attractive and was initially considered as the primary goal of this work, however it is flawed in that the system knows nothing of the semantics of items. An automated tool might see that an item has been changed frequently and so guess that it is important, when it might only be an item which users are playing with, but that is not particularly important. Similarly an item which has not been changed for a long time could be assumed to be unimportant and so garbage collected, when it might in fact be an important land mark which is not changed because it is relied on by users. By having users or administrators specifically annotating an item they can take in to consideration both the semantics and the infrastructure needs of an item. These annotations could potentially be used to indirectly drive the operation of secondary infrastructure mechanisms such as the SLRU caching mechanism discussed in chapter 6, but it is unlikely that the entire system could be automated. The system cannot guess the semantics of an item, but it can be guided towards that understanding.

7.2.5 Implementation

The issues discussed above are clearly more important than any technical improvement to the prototype implementation which is currently perfectly adequate as a proof of concept. However, there are a number of areas in which it could be improved or aspects which should perhaps have been implemented differently.

7.2.5.1 Deep Behaviour Object Model

The current lightweight object model used for deep behaviours is very awkward and should be changed. Initially, the annotation of deep behaviour items with their properties seemed attractive for a number of reasons. It made the modification of properties possible using the standard MASSIVE-3 API calls and allowed the leveraging of the standard mechanisms for persistence for deep behaviours and their properties. However, each type which is needed as a property of a deep behaviour must have a corresponding item type which can exist in the scene graph. If a deep behaviour needs a list property in the scene graph then a ListData class needs to be implemented which inherits from ItemData, allowing it to exist in the scene graph, contains a list and implements all of the methods needed for serialisation. This process must be repeated for each type which could be a member of a deep behaviour, resulting in parallel class hierarchies (properties and their scene graph representations) which must be maintained together and are largely redundant. The alternative approach is to implement a single SerialisableData class which inherits from ItemData and points to a serialisable class. This approach, used in the prototype implementation is the equivalent of embedding a void* pointer in the scene graph for every deep behaviour property. This means the vast majority of a deep behaviour's work consists of type checking its properties after it is deserialised or updated. Another problem caused by this approach is that it allows annotation to represent multiple relationships. A deep behaviour might be annotated with a number of properties and a number of deep behaviours. The property annotations represent an aggregation relationship, while the deep behaviour annotations specify behaviour. It is awkward to ensure that the deep behaviour and its child properties are subject to the child deep behaviour. In the future a deep behaviour and its properties should appear as a single composite object in the scene graph, not an annotation with annotations. Alternatively, the system should be re-engineered as a LambdaMOO style object model in which case complex behaviours would be objects like everything else.

7.2.5.2 Swizzling

Most persistent store implementations use a process of converting between memory addresses and unique keys in the persistent store known as Swizzling (Atkinson and Morrison, 1995). The serialisation approach utilised in the exploratory experiments did not require this process as all anonymous members of a named root object were stored in a single serialisation. When this mechanism was extended to allow for updating of stored member objects, such as individual items in a stored environment, the existing MASSIVE-3 identifications where used to provide storage keys for these member objects. While this was sufficient for objects of types which supported identification, such as items, environments and events, it meant that accessing arbitrary members of a stored object independently of the parent was not possible. The only possible workarounds were to modify each class to support identity, or to create a sub-class which provided identity and modify the parent class to use the new sub-class in place of the class with no support for identity. Both approaches were unsatisfactory as they conflicted the ideal of not modifying MASSIVE-3 classes just to support persistence. One of the main reasons that the deep behaviour object model was awkward is that many deep behaviours had to convert between native MASSIVE-3 classes and new wrappers which were needed for directly accessing objects in the persistent store. While MASSIVE-3's serialisation could be extended to elegantly break down and reconstitute composite objects to allow each object to be in its own database record, the problem of identifying arbitrary sub-objects made it difficult to take advantage of this approach. Modifying the prototype implementation to support full swizzling between memory locations and database keys would make the persistence service more general, much easier to use and truly independent of the other MASSIVE-3 classes.

7.2.6 Large Complex Worlds

As with all application frameworks and platforms, the real test of success or failure is to use it in anger and see how well the framework supports a real application. The framework presented in this work is no different. An extended period of use would test the ability of the framework to support the evolution of content and use in a virtual environment. Using the framework to develop a large, heterogeneous world would test the ability of the framework to support multiple infrastructure mechanisms simultaneously. Most importantly the development of a complex virtual world would allow the discovery of many item roles and the deep behaviours to support them. In many of the current virtual environment applications, users do little more than move around the environment or move artefacts in the environment. In these cases often only two item roles can be identified – the items which are parts of embodiments and the items which make up the rest of the world. In these applications the use of the deep behaviour framework appears to be overkill. Indeed a simple flag specifying whether an item is an embodiment would suffice in place of the apparently baroque use of annotations, event pipes and filters. It is only when virtual environment applications become significantly more complex that the full potential of the deep behaviour framework will become apparent. It is hoped that the framework will be able to evolve with the complexity of virtual environment applications just as it allows the virtual environments themselves to evolve with the communities which use them.

7.3 The Big Picture

Making collaborative virtual environments persistent is relatively easy. All that is required is that the content of the environment is written to stable storage so that changes are not lost when the system is shutdown or suffers a failure. This work has shown that making these persistent environments dynamic and flexible enough to evolve with use without being shut down is much more difficult. It is therefore tempting to solve the first problem and ignore the second. However, when persistent virtual environments are used by millions of users around the globe and around the clock, the thought of shutting them down for maintenance will be as unthinkable as closing down a web site like www.google.com. The next task is to build the large scale environments which thousands or millions of people want to use.

Appendix A – Video Contents

- Video Figure 1. A demonstration of the MOLE (MASSIVE On-Line Editor) used in the experiments described in chapter 2. The video first introduces the standard MASSIVE-3 user client and then details and demonstrates the features added to that client to allow on-line editing of a virtual environment's content.

- Video Figure 2. A number of video segments showing activity in the museum experiments described in chapter 2. The activity is shown at an accelerated rate and the segments present highlights from the experimental activity.

- Video Figure 3. A demonstration of multiple deep behaviours applied to different artifacts in a single virtual world. In addition to using distributed event filters to implement the deep behaviour, an event filter is used to delay events communicated between processes. This allows deep behaviours using different consistency mechanisms to be easily distinguished. The activity was recorded from 2 different users' perspectives. These video streams were synchronised and are presented simultaneously in the video figure. This makes the differences in consistency mechanisms clearer and allows the subjective differences produced by deep behaviours to be demonstrated.

References

Adams, D. (1979), *The Hitchhikers Guide to the Galaxy*, London: Pan Books Ltd.

America Online, Inc. (2001), "AOL Instant Messenger", http://www.aol.com/aim/homenew.adp (Verified 15th October 2001)

Atkinson, M. P., Bancilhon F., DeWitt, D. J, Dittrich, K. R., Maier, D., Zdonik, S. B. (1989), "The Object-Oriented Database System Manifesto", in *Deductive and Object-Oriented Databases, Proceedings of the First International Conference on Deductive and Object Oriented Databases* (DOOD 89), pp. 223-240.

Atkinson, M. P., Morrison, R. (1995), "Orthogonally Persistent Object Systems", in *VLDB Journal*, 4(3), pp 319-401.

Baraff, D. and Witkin, A. (1998), "Physically Based Modelling", SIGGRAPH Course Notes, 1998, B1-C12

Barrus, J.W., Waters, R.C., Anderson, D.B. (1996), "Locales: Supporting Lange Multiuser Virtual Environments", in *IEEE Computer Graphics and Applications*, 16 (6), Nov., 50-57

Beirbaum, A., Just, C., Hartling, P., Cruz-Neira, C. (2000),"Flexible Application Design Using VR Juggler", Technical sketch presented at SIGGRAPH 2000, New Orleans, July 2000.

Benford, S. D. and Fahlén, L. E. (1993), "Awareness, Focus, Nimbus and Aura - A Spatial Model of Interaction in Virtual Worlds", in *Proceedings of HCI International '93*, Orlando, Florida, 1993.

Benford, S., Greenhalgh, C., Brown, C., Wyver, J., Morphett, J., Walker, G. (1998), "Experiments in Inhabited TV", in *Proceedings of CHI'98*, 1998.

Borras, P., Doucet, A., Pfeffer, P., Tallot, D. (1989), "The O2 Programming Environment", in *Building an Object-Oriented Database System The Story of O2*, Francois Bachilhon, Claude Delobel, Paris Kanellakis Eds., Database Management Systems Series, Morgan Kaufmann, San Mateo, California USA

Bradner, E., Kellog, W.A. and Erickson T. (1999) "The Adoption and Use of BABBLE: A field study of chat in the workplace", in *Proceedings of ECSCW '99*, September 1999.

Bullock, A., Benford, S. (1999), "An Access Control Framework for Multi-User Collaborative Environments", in *Proceedings GROUP 99*, Phoenix, Arizona, USA.

Capps, M., Watsen, K., Zyda. M. (1999), "Cyberspace and Mock Apple Pie: A Vision of the Future of Graphics and Virutal Environments.", in *IEEE Computer Graphics and Applications, Special Issue on Virtual Reality*, November-December 1999.

Carey, R., Bell, G., Marrin, C.(1997), *ISO/IEC 14772-1:1997 Virtual Reality Modeling Language (VRML97)*, The VRML Consortium Incorporated, 1997.

Churchill, E. F. and Bly, S. (1999), "Virtual Environments at Work: Ongoing Use of MUDs in the Workplace", in *Proceedings of WACC '99*, San Francisco, CA, USA. ACM Press, 1999.

Coldeway, J., Keller, W. (1996), "Multilayer Class", in *Collected Papers from the PLoP 96 and EuroPLoP 96 Conferences*, Washington University, Department of Computer Science, Technical Report WUCS 97-07, February 1997.

Cruz-Neira, C., Sandin, D.J., DeFanti, T.A. (1993), "Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE.", in *Proceedings of SIGGRAPH '93 Computer Graphics Conference*, (ACM SIGGRAPH), August 1993, pp. 135-142.

Curtis, P. (1997), “LambdaMOO Programmer’s Manual”, ftp://ftp.reseach.att.com

Czernuszenko, M., Pape, D., Sandin, D. J., DeFanti, T. A., Dawe, G. L., Brown, M. (1997), "The ImmersaDesk and Infinity Wall Projection-Based Virtual Reality Displays.", in *Computer Graphics*, 31(2), May 1997, pp. 46-49.

Date, C. J. (2000), *An Introduction to Database Systems*, Addison Wesley Longman, pp. 453-502.

Deux, O. et. al. (1990), "The Story of O2" in *Transactions on Knowledge and Data Engineering*, 2(1), March 1990, pp. 91-108

Dourish, P. (1996), "Consistency Guarantees: Exploiting Application Semantics for Consistency Management in a Collaboration Toolkit", *In Proceedings of the ACM Conference on Computer-Supported Cooperative Work* CSCW'96 (Boston, Mass.). New York: ACM.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995), *Design Patterns Elements of Reusable Object-Oriented Software*, Addison Wesley.

Garfinkel, S., Spafford, G. (1996), *Practical UNIX & Internet Security*, O'Reilly.

Garton, L., Haythornwaite, C. and Wellman, B. (1997), "Studying Online Social Networks", in *Journal of Computer-Mediated Communication*, 3(1), 1997.

Geiger, K. (1995), *Inside ODBC*, Microsoft Press, 1995.

Goldberg, A., Robson, D. (1983), *Smalltalk-80: The language and its implementation*, Addison Wesley, 1983.

Gong, L. (1998), "Secure Java Classloading", in *IEEE Internet Computing*, 2(6), November/December 1998, pp. 56-61.

Gosling, J., Joy, B., Steele. G. (1996), *The Java Language Specification*, Addison Wesley Developers Press, Sunsoft Java Series, 1996.

Greenhalgh, C. (1997), "Analysing movement and world transitions in virtual reality tele-conferencing", in *Proceedings of the 5th European Conference on Computer Supported Cooperative Work* (ECSCW'97), Lancaster, UK, September 1997, Kluwer.

Greenhalgh, C. and Benford, S. (1995), "MASSIVE: A Virtual Reality System for Tele-conferencing", in *ACM Transactions on Computer Human Interfaces (TOCHI)*, 2(3), pp. 239-261, ISSN 1073-0516, ACM Press, September 1995.

Greenhalgh, C., and Benford, S. (1997), "A Multicast Network Architecture for Large Scale Collaborative Virtual Environments", in *Multimedia Applications, Services and Techniques - ECMAST'97, Proceedings Second European Conference*, Serge Fdida and Michele Morganti (eds.), Milan, Italy, May 21-23, 1997, pp. 113-128, Springer.

Greenhalgh, C., Purbrick, J., Benford, S., Craven, M., Drozd, A. and Taylor, I. (2000)
Temporal links: recording and replaying virtual environments, in Proceedings of the 8th ACM international conference on Multimedia (MM 2000), pp. 67-74, ACM Press.

Greenhalgh, C., Purbrick, J., Snowdon, D. (2000) "Inside MASSIVE-3: Flexible Support for Data Consistency and World Structuring", in *Proceedings of the Third ACM Conference on Collaborative Virtual Environments* (CVE 2000), San Francisco, CA, USA, September 2000, pp. 119-127, ACM Press

Hamilton, G. (1997), "JavaBeans API Specification Version 1.01", http://java.sun.com/beans

Hindmarsh, J., Fraser, M., Heath, C., Benford, S. and Greenhalgh, C. (2000), "Object-Focused Interaction in Collaborative Virtual Environments", in ACM Transactions on Computer-Human Interaction (ACM TOCHI) Special Issue on Collaborative Virtual Environments, S. Benford, T. Rodden and P. Dourish (eds.), 7 (4), December 2000, ACM Press

Horstmann, M., Kirtland, M. (1997), "DCOM Architecture", http://msdn.microsoft.com/library/backgrnd/html/msdn_dcomarch.htm

Hubbold, R., Cook, J., Keates, M., Gibson, S., Howard, T., Murta, A., West, A., Pettifer, S. (1999), "GNU/MAVERIK: A micro-kernel for largescale virtual environments", in *Proceedings of VRST'99, ACM Symposium on Virtual Reality Software and Technology 1999*, , London, December 1999, pp. 66-73, ACM Press.

ICQ, Inc. (2001), "The ICQ Internet Chat Service", http://www.icq.com (Verified 15th October 2001)

Keller, W. (1997), "Mapping Objects To Tables: A Pattern Language", in *Proceedings of 1997 European Pattern Languages of Programming Conference*, Irrsee, Germany, Siemens Technical Report 120/SW1/FB 1997.

Keller, W. (1998), "Object/Relational Access Layers A Roadmap, Missing Links and More Patterns", in *Proceedings of the 3rd European Conference on Pattern Languages of Programming and Computing*, UVK Universitätsverlag Konstanz, 1999, ISBN 3-87940-655-3

L. Bergmans and M. Aksit (2001), “Composing Crosscutting Concerns Using Composition Filters", to be published in *Communications of the ACM*, October 2001, ACM Press.

Leigh, J., Johnson, A. E., DeFanti, T. A. (1996), "CAVERN: A Distributed Architecture for Supporting Scaleable Persistence and Interoperability in Collaborative Vritual Environments", in *Virtual Reality*, 2(2), pp. 217-237.

Morningstar, C. and Farmer, R. (1990), "The Lessons of Lucasfilm's Habitat", in *Cyberspace: First Steps*, Michael Benedikt(ed.), 1990, MIT Press.

Muller, P-A. (1997), *Instant UML*, Wrox Press, Birmingham, UK, 1997.

Oikarinen J., Reed D. (1993), *Internet Relay Chat (IRC) protocol*, Internet RFCs, RFC1459.

Orfali, R., Harkey, D., Edwards, J. (1996), *The Essential Distributed Objects Survival Guide*, John Wiley and Sons, New York, New York, USA

O'Ryan, C. (2000), "The Design and Performance of a Pluggable Protocols Framework for Real-time Distributed Object Computing Middleware", in *Proceedings of Middleware 2000*, Pallisades, New York, April 3-7 2000, ACM Press.

Paramount Pictures (1998), "The Truman Show", http://www.trumanshow.com (Verified 15th October 2001)

Petersen, K., Spreitzer, M. J., Terry, B. T., Theimer, M. M., Demers, A. J. (1997), "Flexible Update Propagation For Weakly Consistent Replication", in *Proceedings of the 16th ACM Symposium on Operating Systems Principles* (SOSP-16), Saint Malo, France, October 5-8, 1997, pp 288-301.

Pettifer, S., Cook, J., West A. (2000), "DEVA3: Architecture for a large-scale distributed virtual reality system", in *Proceedings of ACM Symposium in Virtual Reality Software and Technology 2000* (VRST'00), Seoul, Korea, October 2000

Ritchie, D. (1984), "The Evolution of the Unix Time-sharing System", in *AT&T Bell Laboratories Technical Journal 63,* 6(2), October 1984, pp. 1577-93.

Sense8 Corp. (1998), "WorldUp Users Guide", Release 4.

Terry, D. B., Petersen, K., Spreitzer, M. J., Theimer, M. M. (1998), "The Case for Non-transparent Replication: Examples from Bayou", in *IEEE Data Engineering*, December 1998, pp. 12-20.

Thelen, F., Beckert (1997), "POET SQL Object Factory - Technical Overview", http://www.poet.com/sql_tech_over.htm, POET GmbH, 1997

Tichy, W. (1985), "RCS: a system for version control", in *Software Practice & Experience*, 15(7), July 1985, pp. 637-654.

Vaghi, I. (2001), *Augmenting The Virtual*, PhD Thesis, Nottingham University, 2001.

Vellon, V., Marple, K., Mitchell, D., Drucker, S. (2000), "The Architechture of a Distributed Virtual Worlds System", http://www.vworlds.org/Docs/ArchOverview/oousenix.htm

Waldo, J. (1999), “The Jini Architecture for Network-centric Computing”, in *Communications of the ACM*, July 1999, pp. 76-82.

Watsen, K., Zyda, M. (1998), "Bamboo - A Portable System for Dynamically Extensible, Real-time, Networked, Virtual Environments", in *The Proceedings* *for the 1998 IEEE Virtual Reality Anuual International Symposium* (VRAIS '98), March 14-18, 1998 Atlanta, Georgia, USA. IEEE.

Yahoo! Inc. (2001), "Yahoo!", www.yahoo.com (Verified 15th October 2001)