The Role of Structural Reflection in Distributed Virtual Reality

Paul Okanda
Distributed Multimedia Research Group,
Computing Department,
Lancaster University,
Lancaster, LA1 4YR.
+ 44 (0) 1524 593315
okanda@comp.lancs.ac.uk

Gordon Blair
Distributed Multimedia Research Group,
Computing Department,
Lancaster University,
Lancaster, LA1 4YR.
+ 44 (0) 1524 593809
gordon@comp.lancs.ac.uk

ABSTRACT

The emergence of collaborative virtual world applications that run over the Internet has presented Virtual Reality (VR) application designers with new challenges. In an environment where the public internet streams multimedia data and is constantly under pressure to deliver over widely heterogeneous user-platforms, there has been a growing need that distributed virtual world applications be aware of and adapt to frequent variations in their context of execution. In this paper, we argue that in contrast to research efforts targeted at improvement of scalability, persistence and responsiveness capabilities, much less attempts have been aimed at addressing the flexibility, maintainability and extensibility requirements in contemporary Distributed VR applications. We propose the use of structural reflection as an approach that not only addresses these requirements but also offers added value in the form of providing a framework for scalability, persistence and responsiveness that is itself flexible, maintainable and extensible.

Keywords

Distributed Virtual Environment (DVE), Virtual Reality (VR), Reflection, Object Behaviour, Adaptation.

INTRODUCTION

Multi-participant shared virtual world applications are real-time distributed simulations in which users navigate and interact within a two or three-dimensional virtual environment. These applications range from non-persistent, short-duration sessions with few users and limited data (e.g. racing online games, virtual shopping applications) to persistent, long duration sessions with many users and voluminous shared data (e.g. virtual communities, multi-participant virtual museums, online role-playing games and collaborative design applications).

Recent research has been aimed at developing distributed platforms that can support DVE applications running on the public internet. This has proved extremely challenging, particularly in massively multi-participant applications where thousands of users potentially interact in real-time with each other and with thousands of autonomous entities using uncontrolled network and local (processor, memory) resources. In an effort to better address these challenges, researchers have identified various capabilities that a DVE system should offer.

Such systems have requirements which include the following:

- Scalability: the ability to continue functioning satisfactorily as the system's execution context changes in size or volume in order to meet diverse user needs.
- Persistence: capacity to remain active even when some/all user sessions have terminated.
- Responsiveness: capability of responding to user demands within a prescribed time frame guaranteeing sustained support for high levels of interaction between many users.
- Flexibility: ability to satisfy differing system constraints and user needs with fluctuations in the system's execution environment.
- Maintainability: the ease with which the DVE system can be modified to correct faults, improve performance, or other attributes.
- Extensibility: the ease with which the DVE can be altered to increase the system's functional capacity.

The main focus of research on DVE platforms has been on the first three capabilities and, as a result, a number of techniques both at the platform and application level have emerged.
To improve scalability, existing published works propose a wide range Virtual World partitioning approaches from static coarse-grained partitions [3] to interest management (perception-based) approaches (VS) [11]. For example, MASSIVE 1-2 [6] combines spatial awareness mechanisms with information aggregation algorithms to provide better support for introducing contextual factors into awareness negotiations.
To address persistence requirements, some DVE platforms such as Continuum [5] implement mastership transfer within peer-to-peer

Conference '00, Month 1-2, 2000, City, State.

architectures. Others maintain centralised databases that regularly maintain versions of object states. For example, in Open Community (OC), a *Persistence server* writes out a disk based version of objects on a regular basis so that if the server has to be terminated then restarted, the disk file is used to regenerate the original set of objects.

To provide support for real-time interaction, researchers in DVE systems have attempted to implement fully distributed architectures together with multicast grouping of clients, e.g. DIVE [3]. Others, e.g. Virtual Society [11], attempt to improve robustness and reduce packet/message delays inherent in single-server architectures by incorporating multicast grouping together with multiple servers each of which provides a specific data set.

(A detailed analysis of techniques used in DVEs can be accessed in [12]).

In contrast, there has been much less effort on addressing the *flexibility*, *maintainability* and *extensibility* requirements of contemporary DVEs. We propose the use of *structural reflection* as an approach that not only addresses these requirements but also offers added value in the form of providing a framework for *scalability*, *persistence* and *responsiveness* that is itself *flexible*, *maintainable* and *extensible*.

This paper is structured as follows:

Section 2 presents a background on reflection. It defines, justifies and details different types of reflection. Section 3 then provides an insight into our overall approach while a description of our system's design is covered in section 4. Implementation details and an overall architecture are covered in section 5 followed by details of experiments and their evaluation in section 6. Section 7 then presents related work and finally, section 8 concludes the paper.

2. BACKGROUND ON REFLECTION

2.1 Definition of Reflection

The conceptual origin of reflection could be traced to Smith [15] who introduced it thus:

For the purposes of this paper, we provide a simple context-specific definition of structural reflection in DVEs as;

'a design principle that allows a Virtual World to have a representation of itself in a manner that makes its adaptation to a changing environment possible'.

2.2 Why Reflection?

The motivation for all reflective systems could broadly be considered to stem from two concerns. These are:

1. *The desire for open implementation* [2],[10]. The classical view in software design is to handle complexity by the use of abstraction (from simple to high level) to hide implementation details from the users. This black-box approach to design promotes re-use of components but it is not always desirable to hide all implementation details from the user. This is because hiding implementation details necessitates making implementation decisions on behalf of the application regardless of how essential the information the application has on the use of a particular module is. The ultimate objective of open implementation is to overcome this problem by exposing the implementation details of the system. This must however be achieved in such a way that there is a principled division between the functionality they provide and the underlying implementation. In this context, the former can be thought of as the base interface of a module and the latter as a meta-interface whose purpose is to provide access to the meta-level of the system. This approach is captured by Rao [14]:

   'A system with an open implementation provides (at least) two linked interfaces to its clients, a base-level interface to the system's functionality similar to the interface of other such systems, and a meta-level interface that reveals aspects of how the base-level interface is implemented'.

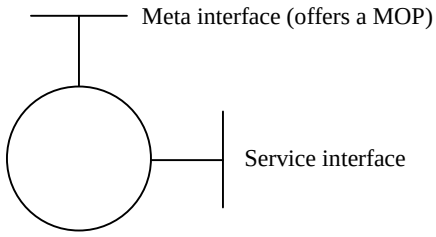

Figure I An Open implementation

   It is important to note that in object-oriented systems, this meta-level interface is often referred to as the meta-object protocol for the object (or MOP) [9]. The Common Lisp Object System (CLOS) MOP for instance creates a reflective object system, using its own mechanisms to create an object-oriented representation of its behaviour.

2. *The desire to provide a principled (as opposed to ad hoc) means of accessing the underlying implementation of a system.* The ability to access the underlying implementation mechanism of a system could be useful in two main aspects:

   Inspection: Reflection can be used to inspect the internal structural behaviour of a language or system. Exposing the system's underlying implementation subsequently makes it straight-forward to insert additional structural behaviour to monitor implementation.

   Adaptation: Reflection can also be used to adapt the internal behaviour of a system either by changing the interpretation of an existing feature (by modification or replacement) or by adding new features.

The use of reflection also has some potential drawbacks. The first drawback is that its use inevitably incurs an additional performance overhead. The most observable issue is the requirement that additional code be used to resolve the precise interpretation of behaviour in the system. The second obvious drawback results from the desire to open up the implementation. Care must always be taken by designers to maintain system integrity when the programmer has open access to the implementation.

2.3 Types of Reflection

Reflective computation can be categorised into two complementary types:

- Structural Reflection: enables the inspection, addition, removal or modification of the encapsulated features of base-level entities, such as functionality (operations, methods) or state (variables, attributes and constants).
- Behavioural Reflection: concerns computation about the interpreter (the virtual machine). It exposes the execution environment and enables one to reason about the way the base-level program is executed.

A reflective language or system can provide both types of reflective facilities. It is desirable, however, that the Meta-Object Protocol provides a uniform way to do both kinds of reflective computation, perhaps using two distinct interfaces which (ideally) employ the same syntactic and semantic conventions.

3. OVERALL APPROACH

Our conviction is that conventional DVE platform architectures are unable to cope effectively with their inherent *flexibility, maintainability* and *extensibility* requirements as a result of two reasons:

- Firstly, as discussed above, their black-box nature inevitably creates a bias in the performance of the resulting implementation since the platform designers have to decide before-hand and make a choice on the implementation, then lock that decision inside the black-box.
- Secondly, even in instances where access to the platform implementation is enabled, their highly coupled nature makes implementation choices of certain services hard-coded in the implementation of others. This intertwining of code inevitably reduces the platforms to monolithic pieces of system software. This makes dynamic adaptability apriori an impossibility.

The above two reasons provide the drive for our use of reflection and more specifically structural reflection coupled with an object oriented approach in our implementation.

As stated earlier, the motivation for this work is to incorporate flexibility, maintainability and extensibility into DVEs. The next section provides details of our design.

4. SYSTEM DESIGN

4.1 The Object Model

Reflection per se does not support flexibility, incrementality or ease of use as this only comes about through the additional application of object-orientation.

This view is supported by Kiczales et al [9] who points out an important synergy between reflection and object-oriented computing thus:

'Reflective techniques make it possible to open up a language's implementation without revealing unnecessary implementation details or compromising portability; and object-oriented techniques allow the resulting model of the language's implementation and behaviour to be locally and incrementally adjusted'.

This provides the inspiration for our use of an object-oriented approach in our design.

In our object model, an object consists of:

- a set of accessible attributes,
- a set of methods to get and set these attributes (collectively forming the interface of the object),
- a set of associated behaviours,
- one or more renderings of the object.

Active objects (e.g. avatars) possess all the four elements while passive objects (e.g. components of the DVE terrain) contain all elements except the set of behaviours.

4.2 The Role of Behaviour

The design of DVEs seeks to model VR applications around various interpretations of reality. Real life artefacts exercise their behaviour to perpetuate their significant subsistence. For example, human beings exercise their 'eating' behaviour without which they would not meaningfully exist. Behaviour is also used to describe artefacts in real life. For instance, within the animal kingdom, mammals nourish their young with milk secreted by mammary glands. Plants on the other hand are defined as living things typically lacking locomotive movement or obvious sensory organs and possessing cellulose cell walls. The phrases 'nourish their young with milk secreted by mammary glands' and 'typically lacking locomotive movement' are observable behaviours that define the existence of human beings and trees as mammals and plants respectively. The fact that behaviour forms an integral part of the existence of real life artefacts gives it a crucial role in our attempts to model them. In VR, behaviour provides a handle in the capture (simulation) of real world phenomena and their run-time adaptation policies/mechanisms.

Behaviour is the way in which the state of an object's attributes changes over time. For instance, an object may have an attribute called 'location'; as it moves around, its location changes. The way in which its location changes over time is its behaviour. Object behaviours at the application level could be classified into four broad categories based on their frequency of change and predictability. They could also be considered to arise from a corresponding set of four basic types of objects. The table below classifies and briefly defines different behavioural forms with real-world examples of objects that exhibit them.

Class	Description	E.g.
Static	have a state that never changes - are therefore deterministic.	*Desk*
Dynamic	change state over time but changes are predictable - a function of time and a set of pre-defined parameters.	*Fan*
Non-intelligent	respond to changes in their environment in straight forward way.	*Door*
Intelligent	governed by unpredictable goals, are therefore non-deterministic.	*Human beings*

Figure II Object Behaviour Classification

We look into an object model that has three categories of associated behaviours:

- Application (shallow) behaviours: are application level and may or may not trigger changes in the system. For example, the simulation of an avatar's change in location (motion) is an application behaviour.

- Platform (deep) behaviours: are system level and exist at the application level as representations of middleware services or mechanisms. For example, a particular consistency policy that implements a receive-order sequence of events is a platform behaviour.
- Hybrid (shallow-deep) behaviours: these are application-system level with an implementation that causally cuts across the entire DVE. For instance, an event channelling protocol that has application-level input in form of packet loss detection is a hybrid behaviour.

4.3 The Meta-model

We adopt the object model earlier described in sub-section 4.1 and use techniques that allow the above three categories of behaviour to be encoded and subsequently be evolved and adapted at run-time.
In particular, we define a meta-interface (Meta-Object Protocol) which essentially offers structural reflective capabilities on application objects with operations that:

- *discover* the internal details of an object in terms of attributes, behaviours etc,
- *insert* a new attribute, behaviour or rendering,
- *delete* an existing attribute, behaviour or rendering or
- *replace* an existing attribute, behaviour or rendering.

The diagram below provides a simplified representation of the meta-interface.

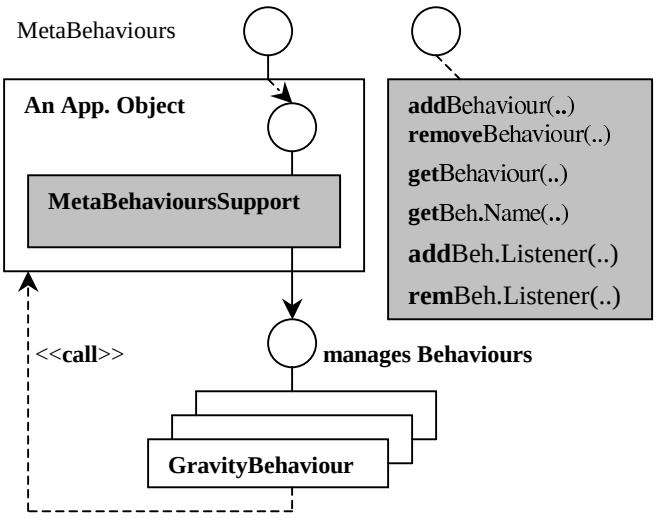

Figure III Design of the Meta-interface

This MOP can then be used for adaptation over the object model described earlier.
Adaptation is essentially the alteration of the underlying implementation of a system in order to suit the needs of its fluctuating execution environment. These fluctuations range from user subjectivity to the system's infrastructural setting.
Adaptation in DVEs should result in applications that are flexible in two main dimensions:

- static flexibility: such as customisation to particular individual/group practises or subjectivity in their demands.
- dynamic flexibility: in response to run-time changes in execution environments in the course of specific collaborations or even specific collaborative sessions.

To achieve the above two dimensions, there is an obvious need to consider adaptation within the entire DVE and in this regard, we identify two types:

- External adaptation: adaptation functions are provided outside an application object, either in a management subsystem, provided by the user or a combination of both.
- Self-adaptation: adaptation management is implemented within an object, i.e. objects continually monitor the execution environment and adapt themselves on the fly.

5. IMPLEMENTATION

5.1 Overall Architecture

A platform has been implemented based on the above design. This platform offers dynamism in DVEs via exploitation of application-specific semantics and run-time execution environment awareness. It provides the application designer with access to application objects as well as mechanisms encapsulated in six service bundles within a middleware platform called ReflectivePING[1].

The service bundles, each with its own set of pluggable mechanisms include: *Concurrency, Replication, Interest Management, Persistence, Consistency* and *Event Channelling*.

The diagram below illustrates the platform's overall architecture.

[1] ReflectivePING is an enhanced version of **P**latform for **I**nteractive **N**etworked **G**ames – the original non- reflective version was designed by a number of partners in a EU funded project.

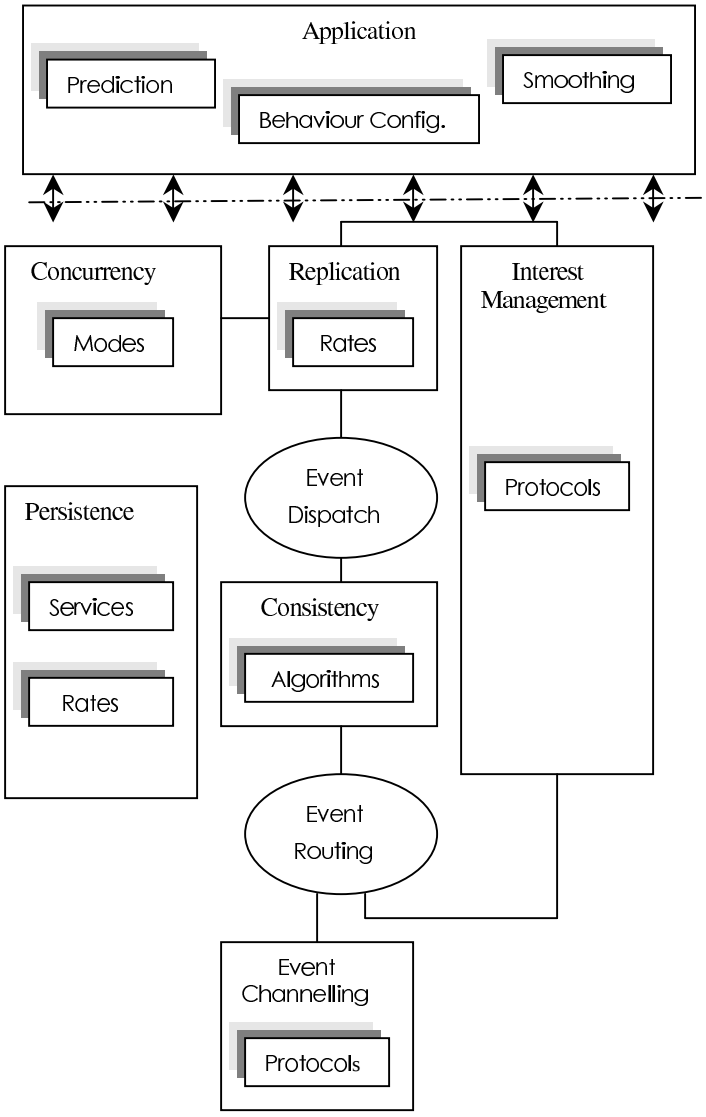

Figure IV Architectural Design

The rationale for the architecture above has a basis on the earlier identified need for incorporation of flexibility and run-time adaptation in contemporary DVEs. This must be considered over a set of services and mechanisms with policies defined to manage their dynamic configuration over an execution kernel.

At the *Object and Event Management Layer*, five service bundles present run-time pluggable or unpluggable mechanisms as detailed below:

- Concurrency comprising:

Lock Transfer Mode [standard or predictive] with normal change of mastership and subsequent transfer of locks between nodes versus predictive anticipation of mastership by nodes hence transfer of locks to implement entity ownership.

- Replication consisting of:

Rate [standard, high or low] with provision of multiple instances of the same object at different nodes varying such that it can be set at run-time.

- Persistence constituting:

Service-type [in-memory or in-disk] with processor and memory resources determining circumstances under which there should be switches.

Check-point Rates [low, standard or high] with snap-shot taking of the simulation state set as a variable that can be altered dynamically.

- Consistency comprising:

Algorithm [receive-order, priority-order or total-order] with receive-order using simple FIFO event ordering in satisfactory network conditions and when weak consistency is not an issue, priority-order such that there's a reference to event creation time at the application level and total-order when strong consistency is a major concern.

- Interest Management consisting of:

Protocol [spatial or publish-subscribe] with spatial based protocols used in perfect network conditions and publish-subscribe protocols applied when there is a need to filter event transmission to nodes according to relevance.

Based on the application object behaviour classification presented in sub-section 4.2, the *Application Layer* presents instances of application-specific mechanisms. We pick on examples that apply in our experimental scenario (as shall be seen later in section 6.1) and categorise them into:

Prediction [on or off] involving modelling of deterministic behaviours at nodes to compensate for high latency with increased processing by each entity through envisaging the Master avatar's trajectory.

Behaviour Configuration [drop, pick or replace] involving dynamic dropping/picking/replacement of behaviours depending on Local/External load levels or User preferences e.g. replacement of rich text with plain.

Smoothing [simple, standard or complex] constituting algorithms applied to counter jerking visual effects on the avatar's trajectory depending on the rate at which updates are sent to the node.

Finally, the *Communication Layer* comprises the Event Channelling service bundle with:

Protocol [reliable, unreliable or Application Level Framing] with reliable channelling used to relay events that require high levels of reliability, unreliable channelling used when high system load levels presents a bigger problem than reliability and Application Level Framing (ALF)[2] when local resources are available and some form of application control over packet loss detection/recovery is important.

We choose to focus our efforts on Replication, Consistency and Event Channelling service bundles for our experiments since efforts to address scalability, responsiveness and persistence concerns have focused on the Interest Management, Concurrency and Persistence service bundles.

Each mechanism is represented as a pluggable or unpluggable behaviour at the application level. Behaviours can be broken down into individual constituent parts called Behavioural

[2] A networking service protocol model that explicitly includes an application's semantics in the design of that application's protocol [Floyd 90].

Attributes (BAs). We define a Behavioural Attribute (BA) as a separable part of the behaviour of an object. Considering motion in a DVE, *InertiaSlave* (an algorithm that models the deterministic *Inertia* behaviour at the slave simulations) is a BA of the behaviour *Inertia*. It encapsulates a reactive program and can be configured or reconfigured individually using properties/methods/events. A reactive program describes a behaviour (or Behavioural Attribute) and its associated state.
We use *a reactive programming* approach to avail a flexible paradigm for encoding reactive systems, especially those which are dynamic since it provides application programmers with a fine control over concurrency, event broadcast and several primitives for gaining fine control over program execution. More specifically, we use a tool called Junior (Jr). The next section explains the reactive programming paradigm.

5.2 Reactive Programming

Reactive programming is a process which involves the encoding of reactive instructions. Since active objects have their own specific behaviour and react continuously to events occurring in their environment (interactions with other objects or time progression), programming active objects (e.g. avatars) in a shared virtual world is essentially a form of reactive programming.
Junior is a Java-based kernel model for reactive programming that defines concurrent reactive instructions communicating using broadcast events [7]. Our choice of Junior is influenced by the fact that its reactive approach avails a flexible paradigm used for programming reactive systems especially those that are dynamic (i.e. the number of components and their connections change during execution).
Programming in Jr is essentially a four-stage process that involves:

1. declaring a reactive machine – to run the program
2. writing a reactive instruction – to describe the application program
3. dropping the program into a reactive machine
4. running the reactive machine – done using a non-terminating loop that cyclically makes the machine and program react.

Below is an example to illustrate the above process.

```
import junior.*;

public class Behaviour
{
public static void main(String[] args){
        Machine machine[1] =
        Jr.Machine[3](Jr.Loop(Jr.Seq(Jr.Atom(new
        ReceiveOrderBA()),Jr.Stop()[2])));
machine.react()[4];
}
}
```

The above excerpt runs *Receive-OrderBA,* a platform (deep) behaviour from the Consistency service bundle. *Receive-OrderBA()* is a description of the application program which in this instance has the code which orders events First-In-First-Out (FIFO) from the Object and Event Management Layer to the Application Layer.

5.3 Adaptation Management

Adaptation management concerns the monitoring of objects, the decision making based on observed trends, and the subsequent enactment of the decisions through a feedback and control loop.
Our meta-interface drives such behavioural changes as addition/removal at run-time of pluggable or unpluggable purely application behaviours, purely platform behaviours and hybrid behaviours.
We perform various instances of both:

- coarse-grained adaptation at run-time for instance in addition/removal or replacement of algorithms earlier mentioned in the Consistency service bundle or protocols in the Event Channelling service bundle.
- fine-grained adaptation for instance in configuration of rates used within the Replication service

The diagram below gives a visualisation of adaptation management in our architecture.

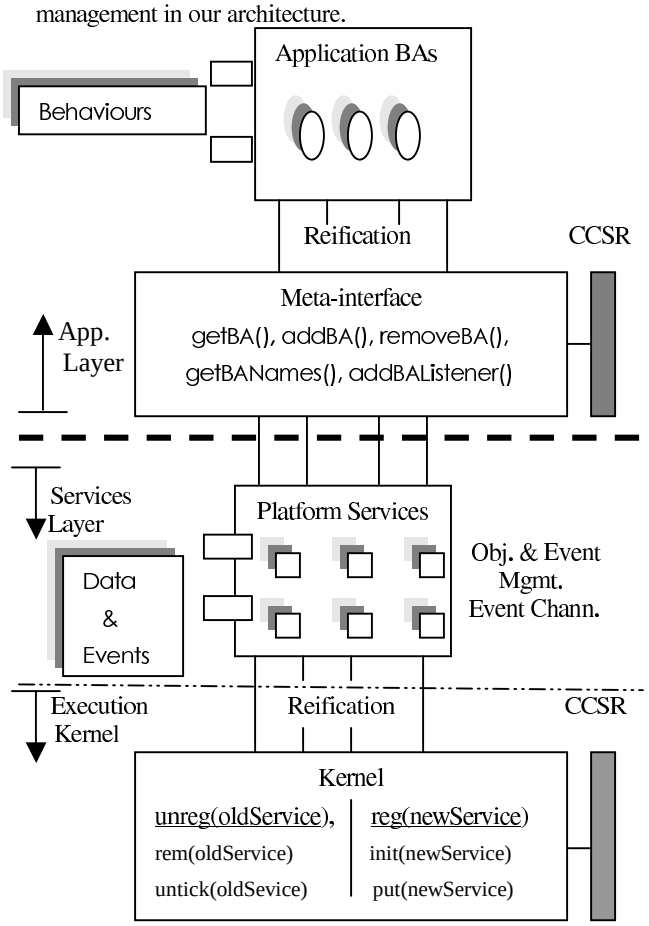

Figure V Adaptation Management

The *Application Layer* models both application behaviours and also a representation of system behaviours, thus providing a common metaphor for adapting the system. Run-time adaptation of the application-specific behaviours occurs within this layer while the more generic system behaviours adapt via configuration and reconfiguration of platform services. In both cases, though this is modeled as changes in behavioural attributes. To support this, the meta-interface offers operations to discover, insert, delete and replace both application and system behaviours via such constructs as `addBA(), getBA()` etc.

The *Services Layer* comprises the entire Platform's service bundles mentioned in **Figure IV.** These are handled in form of data structures and events**.**

The Execution Kernel offers a Causally Connected Self Representation (CCSR) of the Platform services and a reification that enables transparent (from the application programmer's viewpoint) unregistration (of an old service) and registration (of a new service). Just like application-specific behaviours, the run-time configuration of platform services is done in the form of operations that the *Application Layer's* meta-interface offers. Hence invoking these operations at the *Application Layer* triggers corresponding actions within the OpenPING's execution kernel to unregister 'old' services and register 'new' ones dynamically.

6. EXPERIMENTS AND EVALUATION

From our implementation, we have set up four experiments that focus on allowing developers to adapt object behaviour at run-time**.** Our experimental prototype is a simple 'RobotWar' game in which remote users attempt to 'fire' at one another's robots using 'canons'**.** In the game, each user has ownership of a single robot (replicated at all remote sites) which can move around and 'holds' a 'canon' that 'fires' at the rest of the users' robots at a key-press**.** The challenge is to evade all the opponents' 'missiles' and at the same time 'shoot' down their robots**.**

Below is a screenshot of the simulation with two users' views of the game arena each with an execution-monitoring panel below it**.**

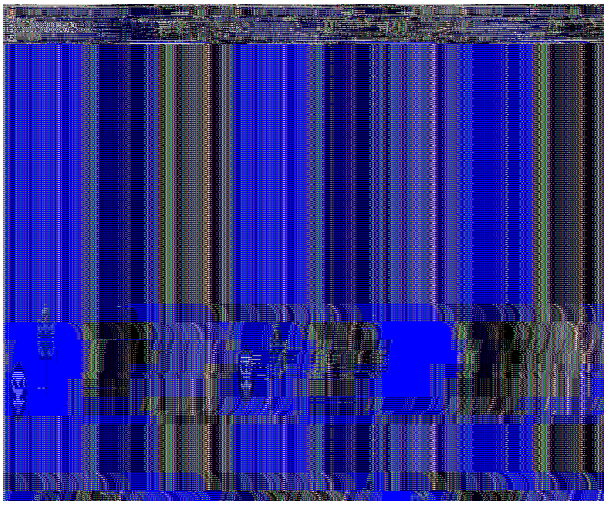

Figure VI Screenshot of 'RobotWar' game

In the 'RobotWar' game, our interpretation models context-specific application (shallow) behaviours alongside standard implementations of platform (deep) and hybrid (shallow/deep) behaviours as shown in the experiments briefly outlined below**.** (Further details can be accessed in the author's PhD thesis currently in preparation)**.**

6.1 Expt. 1 – Application Behaviours

Aim: To enable dynamic addition/removal of deterministic and non-deterministic application BAs that impact on system resources.

Implementation: The experiment is designed such that *GravityBA* is added or removed at run-time. The system can also self-adapt by using a set policy to add/remove *GravityBA* by continuously monitoring a feed-back loop on local system load.

Another policy selects one amongst a pre-defined set of local 'light-weight' (bandwidth-hungry) and 'heavy-weight' (processor/memory-hungry) *InertiaBA* algorithms at the same time adding or removing *BounceBA***.**

Code:

To illustrate the above, below is an excerpt from the class `GravityChannPolicy.java` that adds/removes at run-time, the purely Application Layer BA *Gravity* at the press of a button**.**

```
MetaBehaviours robotMeta =
(MetaBehaviours)env.linkedObject();
Robot robot =
(Robot)((ReactiveSimObject)robotMeta).getEntityPee
r();
// reverse the GravityBA and change the object's graphical
representation
if (robotMeta.getBA("Gravity")!=null){
   robotMeta.removeBA("Gravity");
   robot.setIconFileName(images[1]);
}//ending 'if….'
else {
   robotMeta.addBA("Gravity",
(BehaviouralAttribute) new GravityBA() );
robot.setIconFileName(images[0]);
}//ending 'else….'
```

Results: When the system executes, there is observable dynamic configuration (replacement, dropping, picking) of Behavioural Attributes depending on local & external load for the best visualization in dealing with Local Client Delay (LCD) at the graphics and rendering level.

Various models of the deterministic BA *Inertia* are applied depending on replication rates and this causally tweaks smoothing and prediction algorithms in force at any one time to mask Client-Client-Delay.

Evaluation: The above results exemplify OpenPING's incorporation of adaptation management. The framework implements this in form of objects continually monitoring the execution environment and adapting themselves at run-time. The four lines of code in the `'if-else'` expression above show the simplicity and expressiveness with which application-level behaviours are configurable.

6.2 Experiment 2 – Platform Behaviours

6.2.1 Consistency Service

Aim: To drive run-time causal addition/removal of the Consistency service algorithms: Receive-order, Priority-order and Total-order.
Implementation: *Receive-orderBA* uses simple FIFO event ordering and as such is good enough in satisfactory network conditions. *Priority-orderBA* is used whenever network conditions (monitored via disparities in Master and Slave object positioning) are unsatisfactory.
The system adjusts to the increase in system load by sacrificing strict event ordering (that is activated by *Priority-orderBA*). Conversely, the system fine-tunes itself to a decrease in system load by activating strict event ordering at the platform.
Total-OrderBA's use is not illustrated in this experiment but it is worth noting that it's implementation suits simulations in which very strict consistency is of paramount importance.
Result: The framework's execution is such that *Priority-orderBA*'s addition is causally triggered at the instant the application-level behaviour *InertiaSlaveSimpleBA* is added and the behaviour *Receive-orderBA* causally activated whenever *InertiaSlaveComplexBA* is executed. This evidence reveals how much like application behaviours, platform behaviours can flexibly be configured run-time to conform to fluctuating network and system resource availability.
Evaluation: This experiment illustrates how OpenPING's flexibility facilitates adaptation to fluctuations in load levels and network conditions within the system. Its provision of a Meta Object Protocol (MOP) avails a set of meta behaviours (accessible to the DVE designer at the application level) that support the designer in his choice of implementation from a variety of mechanisms to suit different execution conditions.

6.2.2 Event Channelling Service

Aim: To drive dynamic causal addition/removal of the Event Channelling service protocols: Reliable event channelling and Unreliable event channelling.
Implementation: The system adjusts to the absence of dead-reckoning (prediction and smoothing) mechanisms by causal activation of the reliable packet delivery protocol to compensate for jitter. It also adjusts at run-time to the presence of dead-reckoning by activation of the 'light-weight' but unreliable event delivery protocol.
Additionally, it adapts to the extra load at the application-level whenever *GravityBA* is added by using 'light-weight' unreliable event channelling; also switching at run-time to reliable delivery immediately *GravityBA* (hence additional load) is non-existent.
Result: The system's application switch to *GravityBA* causally activates a switch by the Event Channelling service to Unr*eliableEventChannelBA* in order to counter the effects of the extra load (at the application level) that *GravityBA* adds. Conversely, whenever the Behavioural Attribute *GravityBA* is disabled, ALF*EventChannelBA* is activated such that the platform takes advantage of application involvement in event delivery and recovery.
Evaluation: This result demonstrates OpenPING's provision of multiple infrastructure mechanisms that support real-time interaction. The experiment shows how through the use of structural reflection, it is possible for the DVE designer to exploit the relationship between the application and the underlying platform to optimise the entire system's execution.

6.3 Expt. 3 – Hybrid Behaviours

6.3.1 Replication Service

Aim: To drive dynamic configuration of the rate at which peers in the DVE replicate their states to one another.
Implementation: The application designer can either decrease or increase replication rates at will by activation of the *ForceSynchroBA* to suit a range of network and system resource availability conditions.
Result: The slave (receiving) peers adjust to increase/decrease in replication rates by the Master replica which causally triggers a switch between the Consistency BAs: *Receive-orderBA* and *Priority-orderBA*.
Evaluation: This illustrates fine-grained adaptation by the system in which functions are provided outside an application object, by the user. It shows two instances of adaptation incorporated in the framework; one in which the DVE designer gains total control of the replication rate to peers and another in which receiving peers adjust dynamically to changes in rates at which updates are received.

6.3.2 Event Channelling Service

Aim: To enable dynamic causal addition/removal of the ALF Event Channelling protocol.
Implementation: While the system executes, an application switch to *GravityBA* causally activates a switch by the Event Channelling service bundle to *UnreliableEventChannelBA* such that the underlying platform makes up for the additional load at the Application Layer. Conversely, whenever *GravityBA* is disabled, ALF*EventChannBA* is activated to exploit the information that the application has on the game.
Result: This experiment shows that the DVE adapts to the increase in system load by sacrificing application-semantics' involvement in event delivery. Conversely, it adapts to a decrease in system load by activating reliable event delivery at the platform.
Evaluation: This is evidence that adaptation functions within the framework can be provided outside an application object in a management subsystem. The results of the experiment further prove that the framework's reflective model supports run-time adaptation even in instances where behaviours cannot explicitly be referred to as platform or application.

6.4 Expt. 4 – Performance Metrics

This experiment evaluates the performance overhead that is directly attributed to the additional code used to realise reflection hence run-time adaptation within the framework. It involved the

use of Intel PIII PCs with 128 MB – 256MB memory and 650MHz clock speeds in a 100 Mbps Fast Ethernet Local Area Network (LAN). All the experiments were done on single idle processors and averages (with typical variations measured at ± 2 milliseconds) taken over 100 independent runs.

Aim: To appraise performance metrics and scalability of the OpenPING framework.

Implementation: At start-up, a measure is done on the period of time it takes to load and initialise all services from the platform and start the 'RobotWar' game. To quantify the impact Behavioural Attribute (BA) configuration has on OpenPING's performance, subsequent measurements are made with varying numbers (N) of either Application behaviours or Platform/Hybrid behaviours loaded at the same instant. Measurements are also made to quantify the period of time it takes to configure/reconfigure BAs during normal operation (i.e. after start-up).

Result: It takes an average of 1,735 milliseconds to load the platform and the game at start-up. The total variance between time measurements regarding the configuration or re-configuration of all behaviours during normal operation is 31 milliseconds. Configuring (getting/adding or getting/removing) a single (or two) Application Behavioural Attribute(s) either at start-up or run-time (during execution) costs 16 ms of execution time while it costs a maximum of 31 ms of execution time to load as many as 10 Application behaviours at the same instant.

Loading a single Platform or Hybrid Behavioural Attribute (BA) at start-up or run-time costs 16 ms while it costs a maximum of 31 ms of execution time to load 10 of them at the same instant. The contribution this makes towards attainment of the recommended threshold for effective end-to-end lag in propagation of multimedia data (100 – 300 ms) [8] is not significant.

Below is a graphical representation of loading time (ms) against Behaviours (N) at start-up.

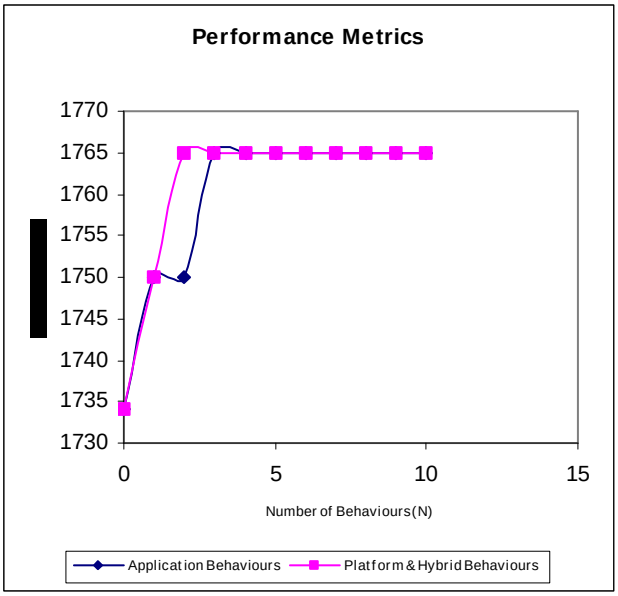

Figure VII Execution time for configuration of Platform Services and application-specific BAs at Start-up.

Evaluation: The figures above give credence to the fact that at just about 1% (of the total execution time) as an overhead incurred by the framework, incorporation of run-time adaptation through structural reflection offers tangible benefits.

The fact that as many as 10 Behavioural Attributes (BAs) are configured at the same instant (at start-up or during execution) without an exponential increase in execution time proves that the approach taken fully meets scalability demands in next generation DVEs.

6.5 Overall Evaluation

In summary, the experiments above:

- demonstrate how OpenPING's meta interface offers support to the designer in his/her choice from a variety of mechanisms in a flexible way.
- show how OpenPING's multiple infrastructure mechanisms (just like Application behaviours) co-exist to enable run-time configuration via policies that the DVE designer defines at the Application Layer.
- prove OpenPING's provision of support for dynamic as opposed to compile-time adaptation of application as well as platform behaviours either at start-up time or during execution.
- epitomize the simplicity, expressiveness and ease with which the DVE designer incorporates a number of OpenPING's mechanisms alongside application-specific behaviours.
- prove that the overhead incurred in execution time is not too big a price to pay in order to avail the full benefits of flexibility.

7. Related Work

7.1 MASSIVE-1,2

MASSIVE [6] (Model, Architecture and System for Spatial Interaction in DVEs) is an experimental prototype whose particular emphasis is on scalability and heterogeneity.

While the MASSIVE system is driven by these two key requirements, the need for incorporation of run-time adaptation is clearly evident. A case in point would be introduction of dynamism in mapping of multicast channel sets onto *third party objects* to achieve both scalability and flexibility.

7.2 DIVE

Developed by the Swedish Institute of Computer Science as a research prototype, DIVE [3] has dynamic behaviours of objects described by interpretative scripts in Tcl. These scripts can be evaluated on any node where an object has a replica and a script is typically triggered by events in the system such as user interaction signals, timers, collisions etc. DIVE has strengths in its performance, use of multicast-based distribution with LAN-bridging and unicast-only support, flexible Tcl scripting, support for sub-division and subjectivity with audio and video support. Its weaknesses are in its assumption that networks have low-loss and reasonably high band-width hence low latency for collaborative manipulation. In early versions of DIVE [3], the ISIS toolkit [1] uses a multicast protocol to distribute changes and set locks. All

nodes are guaranteed to have seen the same sequence of events, which while good for system integrity, provides limits on scalability for instance in DIVE where an upper limit of ten peers was set. On the other hand, in the absence of the ISIS toolkit, consistency guarantees which inevitably improve interactive manipulation especially in environments with high network latencies are non-existent.

7.3 CONTINUUM

Continuum [5] is a research project carried out at France Telecom R&D that targets the design of an open and adaptable platform to support large-scale virtual worlds with emphasis on real-time distributed simulations, multi-player online games and collaborative (design or engineering) applications on the public internet. The framework prototype is based on a flexible Java-based middleware called Jonathan [4] with which RMI and CORBA compliant platforms can be built using appropriate binding techniques. New services can be made available at any time and used in existing applications since application semantics is transparent to infrastructure components. Continuum offers an array of service options but these are essentially compile-time and do not come with an interface or execution kernel that supports run-time adaptation of mechanisms.

7.4 CAVE

In his PhD thesis 'Continuously Available Virtual Environments' presented at Nottingham University in October '01, Purbrick [13] investigates persistence in DVEs. CAVE associates behaviour with platform services in much the same way the application level provides a handle on objects. It however has a limited scope as it only tackles the issue of persistence in continuously available large-scale virtual environments.

8. CONCLUSION

This paper has outlined the need for dynamic adaptation as a means to achieve better flexibility, maintainability and extensibility and also offer support in a flexible way for the run-time incorporation of scalability, persistence and responsiveness techniques. Incorporating dynamically evolving application-specific wishes by making modifications (on the middleware or application) at compile-time is not ideal especially if the application involves real-time interaction and requires round-the-clock availability.

To support dynamic adaptation, this paper has detailed how our framework facilitates not just the co-existence of multiple alternative infrastructure mechanisms but additionally, rather than applying a single mechanism to all environmental scenarios, mechanisms can be tested, replaced, configured or dropped at the application level in the same manner that behaviours in the application are.

Hence we argue that in distributed virtual reality, the use of reflection at the application level to design a meta-interface through which internal managers monitor and adapt platform and application behaviour dynamically is the way forward in the design of next generation DVEs.

8. REFERENCES

[1] Birman K., R. Cooper, T. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck and M. Wood, *"The ISIS System Manual, Version 2.1"*, Cornell University, September 1990.

[2] De Volder K., P. Steyaert, 'Construction of the Reflective Tower Based Open Implementations', Technical Report vub-prog-tr-95-01; Available from Programming Technology Lab, PROG(WE), Vrije Universiteit, Brussel, Pleinlaan 1, 1050 Brussel, Belgium, 1995.

[3] Frecon Emmanuel, Marten Stenius, ***"DIVE: A Scalable Network Architecture for Distributed Virtual Environments"***, Distributed Systems Engineering, 5(3), pp 91-100, 1998.

[4] Frederic Dang Tran, B. Dumant, F. Horn, J.B. Stefani, *"Jonathan: an open distributed processing environment in java"*, Middleware '98, IFIP International Conference on distributed Systems Platforms and Opens Distributed Processing, Lake District, United Kingdom, September '98. Available at http://www.objectweb.org/.

[5] Frederic Dang Tran, Anne Gerodolle, *"An Object-oriented Framework for Large-scale Networked Virtual Environments"*, Springer-Verlag, In Proceedings of the 6th International Euro-Par Conference, Munich, Germany, September 2000.

[6] Greenhalgh C., Benford S., *"MASSIVE: A Distributed Virtual Reality System incorporating Spatial Trading"*, 15th IEEE International Conference on Distributed Computing Systems (ICDCS '95), 1995.

[7] Hazard Laurent, Jean-Ferdy Susini, Frederic Boussinot, *"The Junior Reactive Kernel"*, Research Report, CNET/INRIA RR-3732, July 1999.

[8] ITU90, *"Effect of propagation delays on communication quality"*, International Telecommunications Union (ITU) SG12, February 1990.

[9] Kiczales, G., J. des Rivieres, D.G. Bobrow, *"The Art of the Metaobject Protocol"*, MIT Press, 1991.

[10] Kiczales, G., *"Towards a New Model of Abstraction in the Engineering of Software"*, In Proceedings of IMSA '92 (Workshop on Reflection and Meta-Level Architectures), pp 1-11, A. Yonezawa, B.C. Smith (Eds), Tokyo, November 1992.

[11] Lea Rodger, Yasuaki Honda, Kouchi Matsuda, 'Virtual Society: Collaboration in 3rd spaces on the internet', The Journal of Collaborative Computing, 1997.

[12] Okanda, P., Blair, G., *"Analysis of Techniques used in Distributed Virtual Environments"*, Internal Report Nº. MPG-02-01, Computing Department, Lancaster University, November 2002.

[13] Purbick James, *"Continuously Available Virtual Environments"*, PhD. Thesis submission, Nottingham University, UK '01.

[14] Rao, R., *"Implementational Reflection in Silica"*, Proceedings of ECOOP '91, Lecture Notes in Computer Science, P. America (Ed), pp 251-267, Springer-Verlag, 1991.

[15] Smith, B.C., *"Procedural Reflection in Programming Languages"*, PhD Thesis, MIT, Available as MIT Laboratory of Computer Science Technical Report 272, Cambridge, Mass., 1982.