Scaling OpenSimulator : An Examination of Possible Architectures for an Internet-Scale Virtual Environment Network

A dissertation submitted in partial fulfilment of the requirements
for the degree in Software Engineering

by

Justin Clark-Casey
Kellogg College
University of Oxford
2010

Abstract

This dissertation describes an analysis of possible architectures for an Internet-scale virtual environment network. These are treated as evolutions of an existing virtual environment architecture embodied in an open-source project called OpenSimulator. Distributed computing concepts and Z schemas are used to examine both the existing OpenSimulator architecture and the alternative configurations.

Table of Contents

Introduction

I'd like to start by explaining what I mean by the term 'Virtual Environment' (VE). At the heart of a VE lies a model of a place. This model contains objects, whether simple primitives such as spheres or cubes or more complicated things such as tables and chairs. It may also contain other ideas of place such as terrain or weather. Entities in the model may be subject to rules such as gravity or classical mechanics.

Users interact with this model via a graphical or textual representation through which they can examine, create or modify elements of the environment. Often, the users themselves will be represented inside the VE by an object, otherwise known as an avatar.

Although in principle a VE could be accessed by just a single user, the most interesting ones allow many people to be present in the same environment at once. People can interact via their avatars, chatting to each other in voice or text or changing the objects around them.

Many successful VEs exist today. One is Blizzard's World of Warcraft[1], an extremely popular role playing game. Linden Lab's Second Life[2] is another – a 'virtual world' in which individuals can own virtual land, create objects for sale and socialize.

On the technical level, what these and other successful VEs have in common is a considerable degree of architectural centralization. Though the environment is often distributed over a large number of separate machines, all these machines are situated in the same location and controlled by a single organization. These machines will share the same set of back end data services for the management of services such as user identity and item inventory. Access to the VE is often allowed only through dedicated and proprietary client software - software provided by the same organization that runs the environment.

For my project I wanted to explore how such a centralized VE architecture might be evolved into one that is 'Internet-scale' such as the World Wide Web (WWW). The WWW is much larger than any existing VE system – the WWW has 1.6 billion estimated users[3] while even the most popular online role playing game has only 11.5 million monthly subscribers[4].

But achieving Internet-scale requires properties other than sheer size. For instance, the WWW, in contrast to existing VE systems, has an extremely distributed architecture where any organization can host its own webserver. Equally, unlike existing VEs, the WWW has no centralized data services – information can be hosted anywhere on the Internet by anybody.

The vehicle for my analysis is an open-source project called OpenSimulator. The aim of OpenSimulator is to provide a general platform for running virtual environments. The catalyst for its creation was the open-sourcing of the 3D client provided by Linden Lab for accessing its Second Life virtual world. This history means that OpenSimulator's current VE architecture is heavily influenced by that of Second Life, to the point where Linden Lab has led interoperability experiments between servers running on its own codebase and instances of OpenSimulator.[5]

However, because OpenSimulator is open-source, people and companies have been able to freely

1 http://www.worldofwarcraft.com
2 http://secondlife.com
3 http://www.internetworldstats.com/stats.htm
4 http://eu.blizzard.com/en/press/081223.html
5 https://blogs.secondlife.com/community/features/blog/2008/07/08/ibm-and-linden-lab-interoperability-announcement

speculate about alternative system designs and create experimental implementations. My analysis draws considerable inspiration from this work.

The dissertation is divided up into two major sections. In the first, entitled “OpenSimulator Overview”, I will describe OpenSimulator's current system design. In doing so, I will draw from various distributed computing concepts such as levels of transparency. I will also set out Z models for the different architectural components.

In the second section, “Scaling OpenSimulator”, I will start by putting forth the necessary architectural features that I think that an Internet-scale network of VEs must possess. Then, I'll examine how well the current OpenSimulator design fits these criteria and the extent to which they could be met by evolutionary architectures.

Please note that in my project I am largely unconcerned with the scalability issues associated with the 3D simulation itself. This includes issues such as how to distribute object movement updates to observers, graphical rendering issues and the timeliness of object movement updates. The only place where I will explore issues like these is where they have an impact on the broader Internet-scale architecture. This is akin to the difference between analysing the scalability of an individual website and the WWW as a whole.

I also will not investigate the role of standards. Of course, in the long run, establishing common communication protocols is crucial for growing any large scale architecture. But as of yet, there is no broad agreement in the VE technical community as to what that architecture would look like. I believe that the task right now is to experiment with different arrangements using existing de-facto protocols before distilling these lessons learnt into formal standards.

OpenSimulator Overview

In this section I'll lay out my description of OpenSimulator's current architecture. I concentrate on aspects of the system that are important for a later examination of both its own Internet-scale potential and that of alternative systems. The current architecture I will refer to as the "classic architecture" in order to distinguish it later on from the alternatives.

To set the scene, I'll start with a broad description of OpenSimulator. A more detailed analysis will then examine the system through the lens of distributing computing ideas such as transparency and distributed garbage collection. This will lead to a Z schema model of the classic architecture – a formal description of the system will hopefully act to improve the general clarity of the analysis and provide a basis for formal reasoning when we come to consider how the classic architecture could be extended or changed.

The section will end with a discussion of the communication protocol between OpenSimulator servers and the clients used to access them. Although I'm unconcerned with the fine details, the broad features of the approach employed by OpenSimulator will be important when I come to formulate the necessary properties of an Internet-scale VE network.

System Architecture

Clients, Simulators and Services

At the broadest architectural level, the OpenSimulator system consists of 3 different component types; clients, simulators and services.

A client is an executable running on the user's local system. It graphically renders the simulation's objects and avatars. It also handles input from the user.

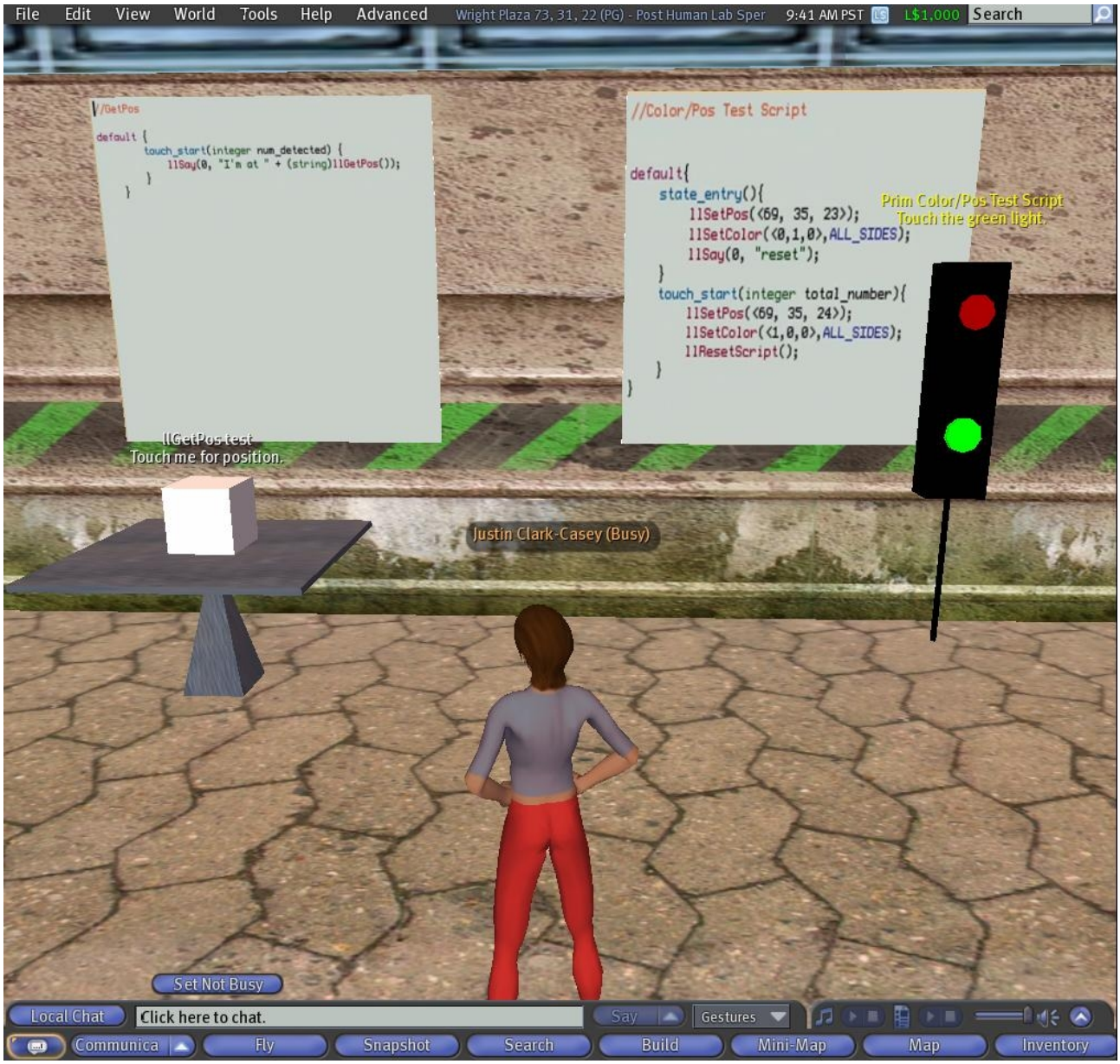


Diagram 1: An example Second Life client interface. Here, most of the screen is taken up with a view of the 3D simulation, with the user's avatar in the middle.

Except during login, the client connects and interacts exclusively with a simulator. A simulator in OpenSimulator is a server responsible for maintaining simulation state information and

communicating changes in that state with the connected clients. Singhal and Zyda describe this particular approach to state communication and management as the centralized repository model [Singhal and Zyda p110]. Changes in state can come from many sources such as a direct response to user actions (for example, is the user makes their avatar push an object), from scripts acting on the object or from environmental effects (one such being simulated gravity).

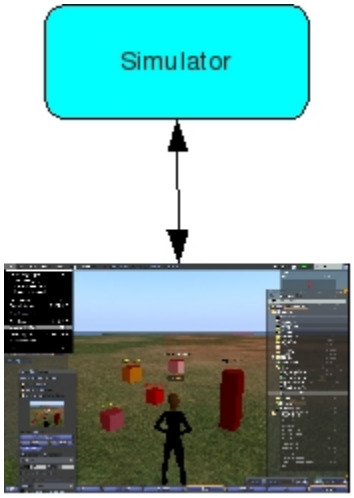


Diagram 2: Interaction between a client and a simulator

In the OpenSimulator system, simulated space is divided up into fixed areas called 'regions'. These are situated at fixed locations on a 2D 'grid'. A single simulator can simulate one or many regions, limited only by processing and network resources.

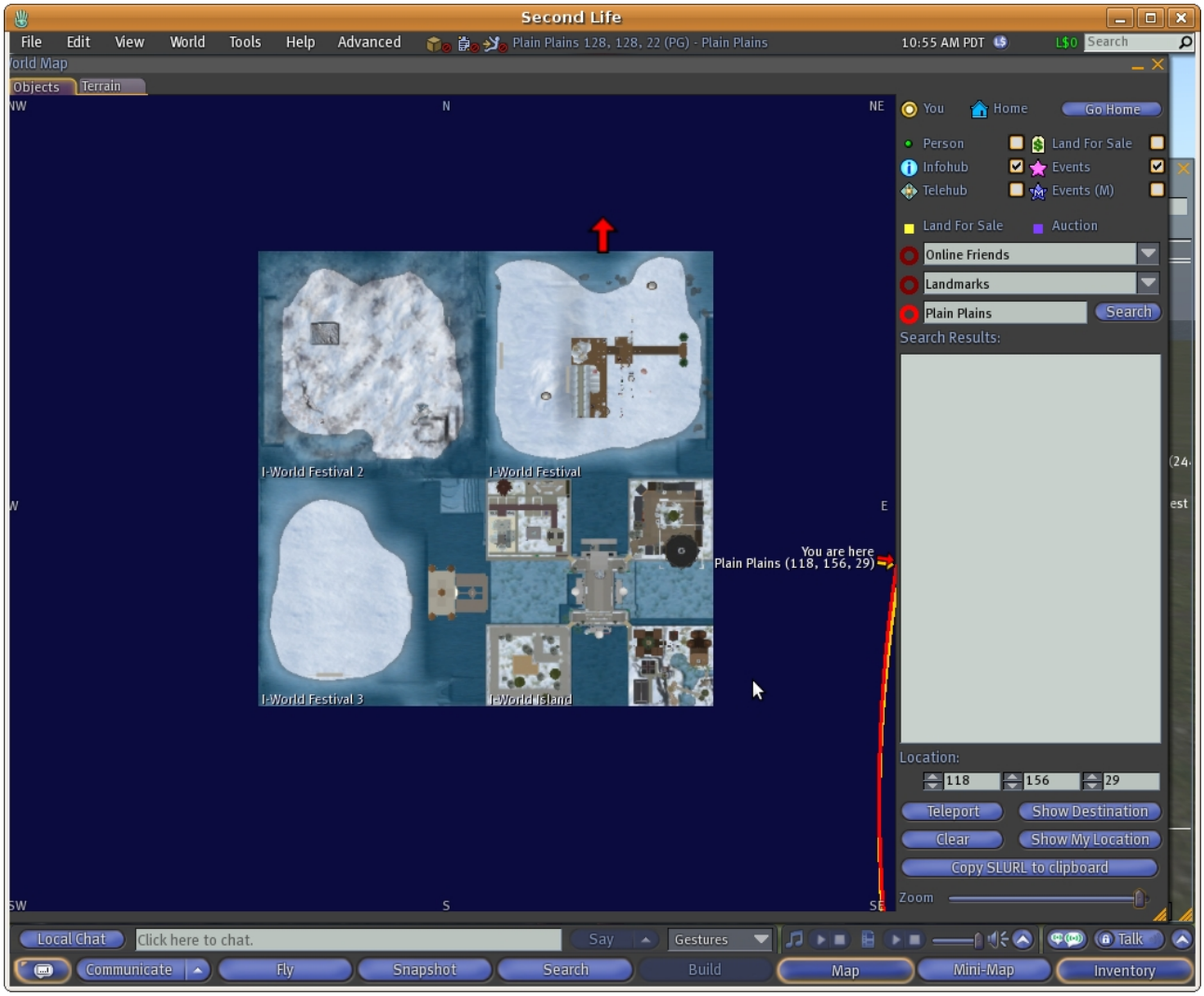


Diagram 3: 4 neighbouring regions in an OpenSimulator grid, here shown on a client's map view.

Regions can be placed next to one another on the grid even if those regions are hosted by simulators running on different machines. A client situated in one region is notified of simulation events occurring in neighbouring regions. This allows the client to render a contiguous 3D environment that is not bounded by the resources of a single simulator.

Avatars travel between regions either by directly moving into neighbouring regions or by teleporting. In the former case there is no interruption in the 3D experience – the avatar can be moved smoothly from one region to another even if the destination region is being simulated on another machine. In the teleport case, there is a discontinuity (an intervening black 'loading' screen) whilst moving from the original location to the teleport destination.

Simulators interact with services via well-known Uniform Resource Locators (URLs) . Services provide resources in common to all the simulators. In this project, I look closely at 4 of these.

- **Asset service.** This service stores all media data persistently. Discrete items of media data are called assets in OpenSimulator terminology. Types of asset include textures, scripts, sounds and serialized object representations.
- **User service.** This persistently stores user data, such as names and biographies. It also

handles initial user login to the virtual environment.

- **Inventory service.** In Opensimulator each user has an inventory. Items in this inventory hold references to assets, which as we've seen include textures, sounds or serialized copies of objects taken from the simulated environment. The inventory service persistently stores this information.
- **Grid service.** The grid service co-ordinates the positioning of regions in the grid. Other region simulators request this information in order to find out about their neighbours.

There are other services available in the OpenSimulator codebase such as instant messaging. These are not relevant to this project so I will not discuss them any further.

Of the three components in the OpenSimulator system – clients, simulator and services - the most complex are the clients and simulators. The client is complex largely because of the work involved in graphical rendering of the environment. The simulator is complex because of its responsibility for maintaining the simulation itself – tasks which include sending and receiving data from remote clients, updating and maintaining simulation state, running physics processes and executing scripts.

Services, on the other hand, are much simpler. Often, they consist of little more than an interface that wraps around some means of persisting data (in a database, for example). They do not carry any state information between requests. In the classic architecture access to these services always takes place via the simulator – the client does not retrieve assets or inventory data directly. This is illustrated in diagram 4.

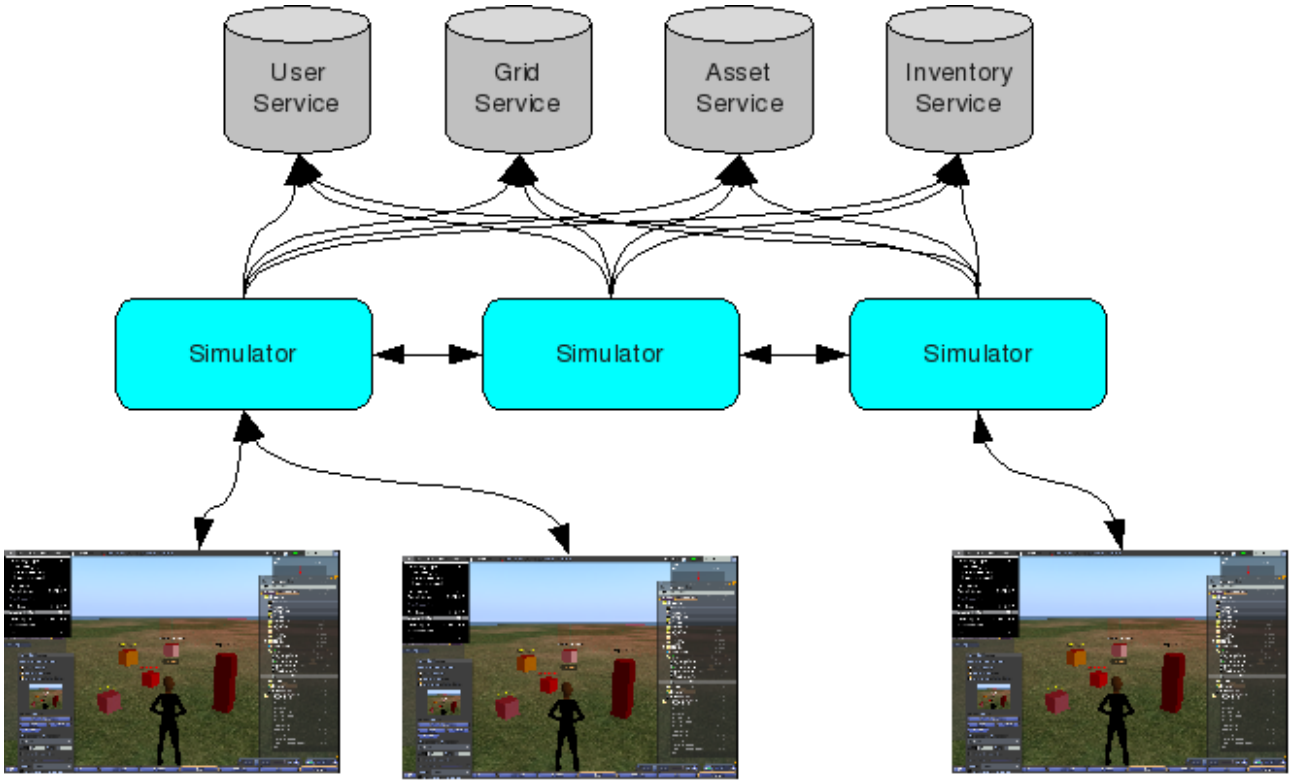


Diagram 4: Interaction of clients, region simulators and services in the classic grid architecture. Only the simulators have access to the VE services – clients always send and receive data through a simulator instance.

Transparency

Since OpenSimulator's classic architecture is to some extent a distributed system, we can examine it in terms of the dimensions of transparency defined by the ISO International Standard on Open Distributed Processing, as described by Emmerich [Emmerich p19].

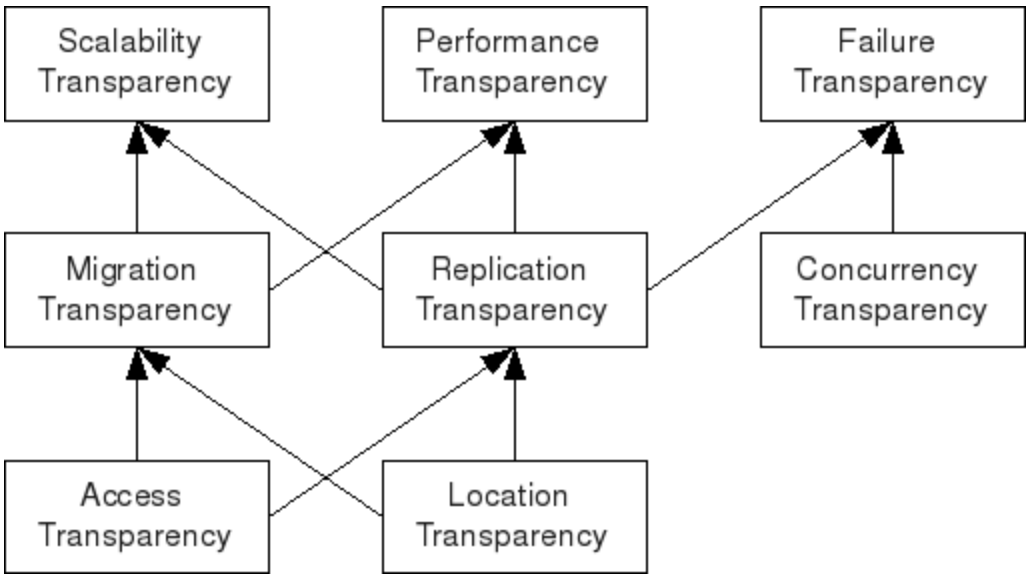


Diagram 5: Dimensions of Transparency [Emmerich p19]

With OpenSimulator, we can analyse transparency from two different points of view. The first is transparency as seen in the user's interactions with a simulator via their client. The second is transparency with the system itself, that is, between simulators and between a simulator and the services.

- **Access transparency.** Access transparency requires that communication with a software component occurs over the same application interface no matter whether that component is local or remote.

  This applies at the client level in the classic architecture - a user can see and move into regions in the same way no matter whether those regions are hosted on the same simulator or on different computers.

  The same is true at the system level. In OpenSimulator, simulators and services can be hosted on different machines, on the same machine or even within the same process. But no matter what the configuration, the interactions between the components are abstracted behind the same interfaces.

- **Location transparency.** Location transparency requires that components can be referenced in service requests without the need to know the physical address of the component.

  OpenSimulator is a location transparent in the same way as the Web. The system can be set up so that clients refer to simulators by their fully qualified domain names, allowing the IP address of a simulator to be changed without any corresponding change on the client. The

same is true for communication between simulators and services.

- **Migration transparency.** Migration transparency requires that components can be migrated from one host to another without requiring any special action from users of those components.

  Component migration doesn't occur in the OpenSimulator system. It is true that on the client level, user and object representations migrate as they move between regions hosted on different simulators. However, this migration is represented by changes in the VE's spatial model and hence is not transparent to other users and objects that are watching them – users and objects are literally moving across virtual 3D space.

  Moreover, on the system level there is no ability to transfer the model of a space to another machine without disruption of that space and the clients that are connected to it. This will have ramifications when we come to look at scalability and performance transparencies.

- **Replication transparency**. Replication transparency requires that a caller be unaware if they are interacting with a single component or one of a number of replicas.

  Since no state information is kept between simulator calls to the services, different service instances can easily be substituted for each other in dealing with different requests. Hence, replication transparency is possible on the service level using the same techniques as employed to load balance websites and Web services [Elson and Howell].

  However, replication transparency between client and simulator is problematic. Simulators, by their very nature as environment models, carry around a vast amount of state that is frequently changing and constantly communicated to clients. Replicating a region would not achieve much since there would need to be a constant flow of information between the original and replica in order to keep the two in sync. Replication transparency is not possible in the classic architecture at the simulator level.

- **Concurrency transparency.** Concurrency transparency requires that any concurrency occurring within a shared component is invisible to callers of that component.

  This is easily achieved for services due to their stateless and isolated nature. For instance, all assets in the asset service are immutable so there is no chance that one update will interfere with another. Equally, inventory requests are always concerned with the user's own inventory rather than any shared data structure.

  But it's a very different story with the simulators. An intrinsic feature of the virtual environment is that it allows users to interact with each other rather than exist in isolation – visibility of concurrency is one of the core features of the simulator. Transparency here is not an option.

- **Scalability transparency.** Scalability transparency requires that it be possible to scale a system by adding more copies of a component without requiring any adjustment by existing component users.

  This is possible at some levels in the OpenSimulator system. In the case of services, as

previously mentioned, load-balancing stateless replicas can easily be added to serve requests made to the service URLs. The simulated area of the entire OpenSimulator installation can be expanded by adding simulators and placing their regions into new slots in the 2D grid, hooking up their service requests to common service points.

Scaling individual regions is much more of a challenge. Each region is fixed in area and can only be simulated by a single machine. As the number of objects or users in a region grows, more load is placed on this single system. Because there's no replication transparency between simulators, the only way to scale is to increase the power of the machine hosting the simulator. But there's a limit to this, particularly since, in a VE system, simulation processing demands grow exponentially rather than linearly with an increasing number of interacting avatars and objects.

We shall talk more about these issues later on.

- **Performance transparency.** Performance transparency requires that the means by which component performance is maintained is opaque to the callers.

  Services can improve performance by adding more replicas, though the limiting factor may become the component from which those replicas are drawing information (such as a database).

  At the simulator level, it's also possible to improve performance but the techniques here are extremely visible to the user. One technique is to hold VE events involving many avatars at the intersection of 4 different regions, where those regions are running on separate machines. Each machine still needs to communicate state to all clients (since clients can see into neighbouring regions), but other workloads such as physics simulation are more distributed.
- **Failure transparency.** Failure transparency requires that the failure of a component is concealed from the caller, such that the caller does not need to handle that failure.

  Failure transparency is possible for services. As we've seen, services have been replication and concurrency transparency - a request can be routed to a waiting replica if the original service instance fails.

  Failure transparency for simulators is impossible due to the lack of replication transparency. If a simulator fails then the clients are simply disconnected. In principle, it might be possible to teleport them to a different location if an impending failure is detected but this does not make simulators failure transparent.

Asset Immutability

Before we go on to formally model the OpenSimulator system, I want to look at the topic of asset immutability. Asset immutability in terms of OpenSimulator is the fact that once an asset has been created and stored, whether that asset is a texture, a script or a notecard, subsequent edits always create entirely new assets rather than any modification to the existing asset.

This allows OpenSimulator to use a 'copy on write' approach to asset creation. For instance,

suppose that Ann creates a teapot which she takes into her inventory. The act of taking stores the teapot as a serialized asset in the asset service with an item in the inventory service that holds a reference to that serialization. Ann then gives a copy of the teapot item to Bob. An item appears in Bob's inventory that represents his copy of the teapot but which points to the same serialized asset referenced by Ann's item. When Bob wants to create the teapot in another simulator, the serialized teapot is fetched from the asset service and deserialized to the 3D environment.

In the meantime, Ann decides that she wants a longer spout on her teapot. So she changes the teapot in the simulation and re-saves it to her inventory. But this doesn't affect the original teapot – by the act of saving Ann creates an entirely new asset and her inventory item is updated to reference the new asset rather than the original. Bob can continue to use the original teapot as long as he likes.

This copy on write system has the following advantages.

- **Efficiency.** In certain cases, this system is much more storage efficient. In a large VE, popular objects or assets may copied thousands of times or more, virtually always without further editing. By only creating a new asset when data is changed, the asset service stores much less data.
- **Caching.** Since assets are guaranteed never to change, they can be cached with impunity. This is helpful on the server side since distributed simulators can maintain local copies of assets, reducing the load on the asset service and enhancing performance.

  On the client side, caching is even more important since the communication bandwidth between the client and a region simulator is lower and generally less reliable. Assets cached on a client's local system only need to be downloaded once.
- **Thread safety.** Since assets are immutable, no special action is required to make sure that one thread doesn't read an asset at the same time that another thread is updating it. This makes it much easier to maintain transparencies such as concurrency and replication in the asset service. There is also some benefit to simulators as no thread synchronization is needed when it comes to manipulating assets.

But there are also some considerable disadvantages to the asset copy on write system.

- **Inefficiency.** In some situations, a succession of assets are created rapidly that are never copied or referenced again. For instance, if a user is in a rapid test and update cycle while developing a script, all the intermediate copies of the script are saved even though they quickly become completely unreferenced. The storage requirements for scripts are magnitudes smaller than they are for textures but the increase in objects could cause problems for some systems.
- **Deletion issues.** Regarding assets, the OpenSimulator system faces the same problems of garbage collection as seen in systems involving multiple layers of middleware [Emmerich, p241]. Asset references are spread over a potentially very large number of regions on potentially a large number of different machines, and not all of these regions may be actively simulated at any one time. Determining whether an asset is unreferenced becomes an extremely difficult task. Hence, no attempt has yet been made in the OpenSimulator system to delete unused assets.

Service, Simulator and Grid Models

This section presents models of the OpenSimulator service and simulator components using Z schemas in order to provide a more accurate and rigorous description of the components that we've described and to form a basis for later formal extension and analysis. These schemas will be abstract distillations of actual implementations so that irrelevant details present no distraction.

We'll start by detailing the basic types that are used in common between all OpenSimulator components. Then we'll look at service schemas before moving on to those of the simulator. Finally, we'll finish with a few schemas that enforce conditions on the grid as a whole.

Common Data Types

Pretty much every entity in the OpenSimulator system has a unique id. These are Universally Unique Identifiers (UUIDs) [RFC 4122]. However, the details of the construction of this identifier don't really concern us – all we need to know is that any randomly generated UUID is effectively unique within the domain of UUIDs since the probability of collision is negligible, the domain of possible values being extremely large. Therefore, we won't worry about enforcing the uniqueness of IDs in our model of OpenSimulator's classic architecture.

We'll simply label a UUID as an ID

[ID]

As UUIDs are not easy for people to read or remember many entities in the system will also have human readable names, though there is no requirement that these be unique.

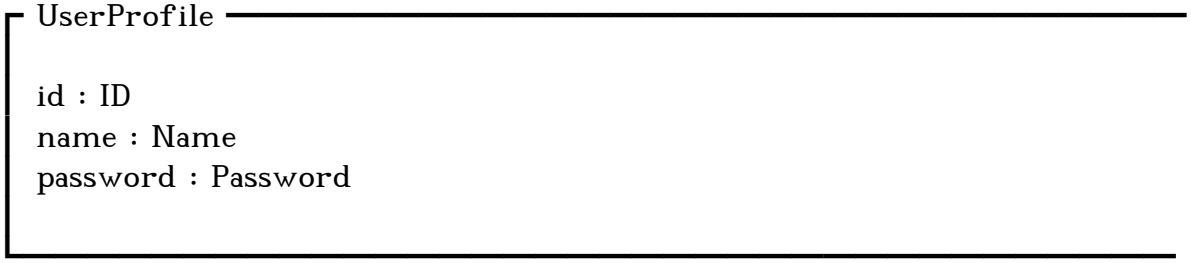
[Name]

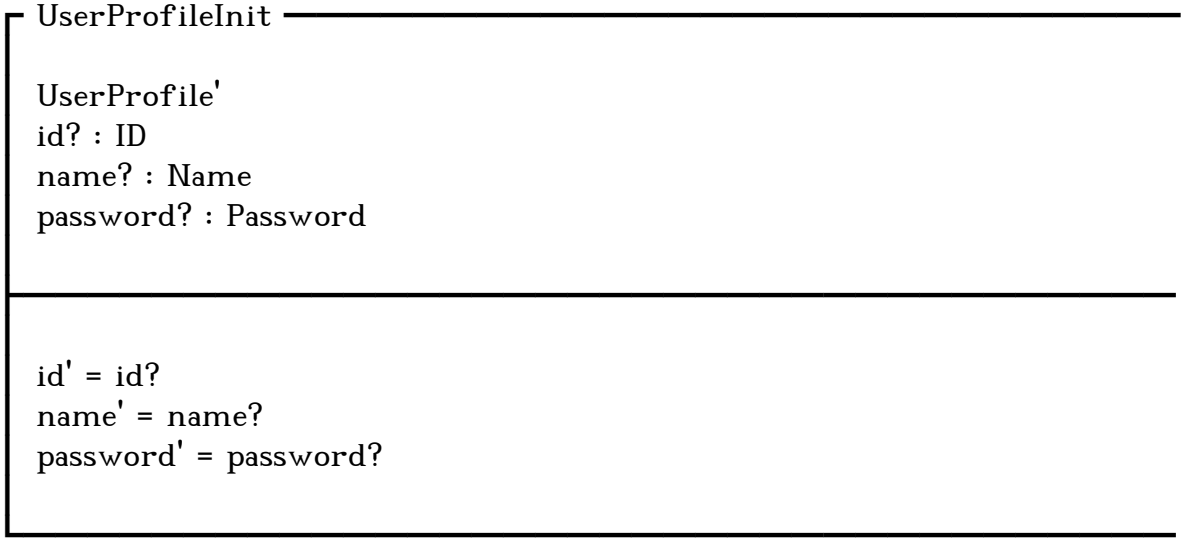
User Service

The user service needs one datatype of its own, namely the password by which users can authenticate themselves on login.

[Password]

Conceptually, the user service can be broken up into two components, user profiles and user sessions. We'll start with user profiles. These contain both the descriptive data about a person – here simply their name – and information necessary to authenticate them – their password. User profiles are identified by a unique id.





A user session is established when a particular user logs on to the system. Each session has a unique id.

```
UserSession'
sessionId? : ID
─────────────────────────────────────────
sessionId' = sessionId?
```

The user service holds both a list of users, whether logged in or not, and a list of current sessions. A user can only have one session at a time. A session also has a one to one relationship with a user – two users can't share the same session. We will specify these conditions in the schema by making the sessions and profiles declarations of UserService partial injective functions.

```
UserService
─────────────────────────────────────────
profiles : ID ⤔ UserProfile
sessions : ID ⤔ UserSession
─────────────────────────────────────────
∀ id : dom profiles • id = (profiles id).id
dom sessions ⊆ dom profiles
```

On initialization, the user service starts off without any sessions since nobody is yet logged in. Profiles supplied at initialization originate externally, usually from persistent storage.

```
UserServiceInit
─────────────────────────────────────────
UserService'
profiles? : ID ⤔ UserProfile
─────────────────────────────────────────
profiles' = profiles?
sessions' = ∅
```

The first pair of operations that we need are those which add and remove profiles from the service.

```
AddUserProfile0
```

$$
\begin{array}{|l}
\Delta \mathit{UserService} \\
\mathit{userProfile?} : \mathit{UserProfile} \\
\hline
\mathit{profiles}' = \mathit{profiles} \oplus \{ \mathit{userProfile?}.\mathit{id} \mapsto \mathit{userProfile?} \} \\
\mathit{sessions}' = \mathit{sessions} \\
\hline
\end{array}
$$

If we remove a user profile that has an active session then we'll also have to remove that session in order to maintain the integrity of the information held in the service.

RemoveUserProfile0

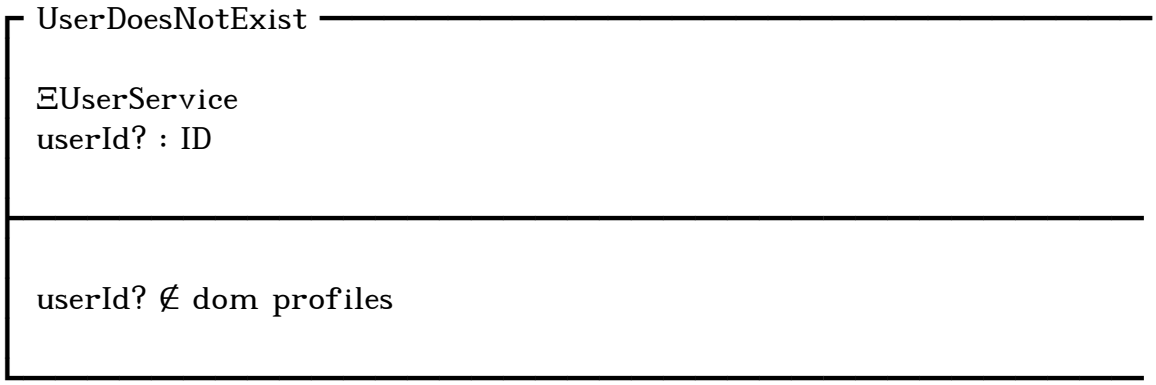
$$
\begin{array}{|l}
\Delta \mathit{UserService} \\
\mathit{userId?} : \mathit{ID} \\
\hline
\mathit{userId?} \in \mathrm{dom}\ \mathit{profiles} \\
\mathit{profiles}' = \{ \mathit{userId?} \} \lhd \mathit{profiles} \\
\mathit{sessions}' = \{ \mathit{userId?} \} \lhd \mathit{sessions} \\
\hline
\end{array}
$$

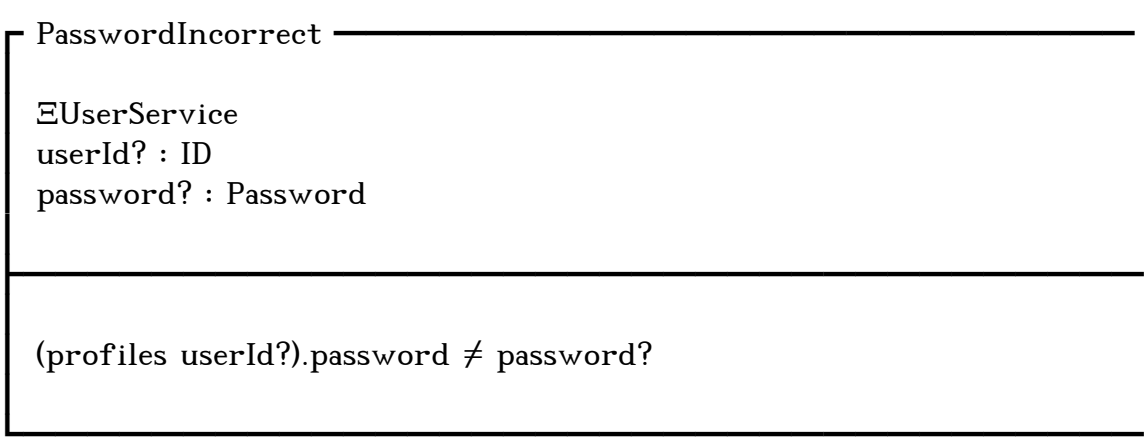
Now let's describe operations that control user login and logout. To login, a user must supply a password that matches that held in their profile. If this and other login conditions are met then a user session is associated with that user. The session information is sent back to the client.

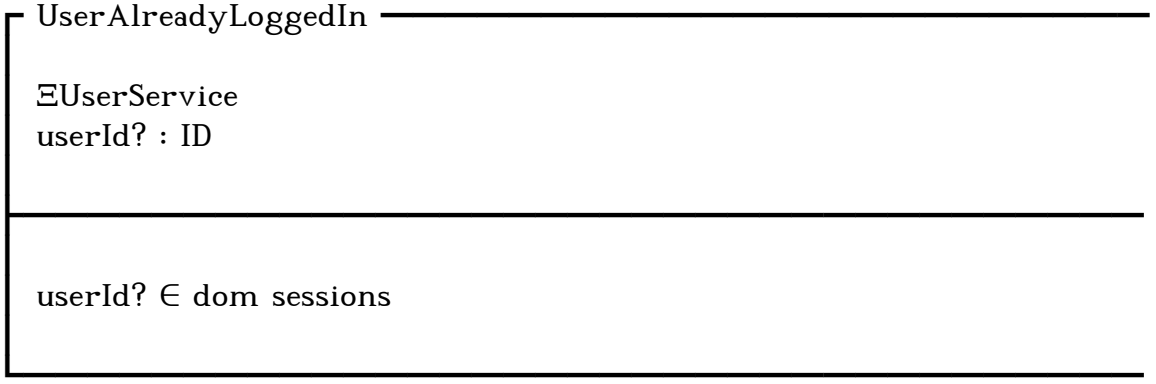
LoginUser0

$$
\begin{array}{|l}
\Delta \mathit{UserService} \\
\mathit{userId?} : \mathit{ID} \\
\mathit{password?} : \mathit{Password} \\
\mathit{session?} : \mathit{UserSession} \\
\hline
\mathit{userId?} \in \mathrm{dom}\ \mathit{profiles} \\
\mathit{userId?} \notin \mathrm{dom}\ \mathit{sessions} \\
(\mathit{profiles}\ \mathit{userId?}).\mathit{password} = \mathit{password?} \\
\mathit{profiles}' = \mathit{profiles} \\
\mathit{sessions}' = \mathit{sessions} \oplus \{ \mathit{userId?} \mapsto \mathit{session?} \} \\
\end{array}
$$

Failure to meet different login conditions results in different failures.







These make up the total operation

$$LoginUser \triangleq LoginUser0 \lor UserDoesNotExist \lor PasswordIncorrect \lor UserAlreadyLoggedIn$$

When a user logs out they have to supply their session id so that the request cannot be spoofed by a

third party. We're going to split the LogoutUser0 operation into a raw user logout operation and a session ID check that we'll be able to reuse later.

LogoutUserRaw

$$\Delta UserService$$
$$userId? : ID$$

$$profiles' = profiles$$
$$sessions' = \{ userId? \} \lhd sessions$$

CheckSessionId

$$\Xi UserService$$
$$userId? : ID$$
$$sessionId? : ID$$

$$userId? \in dom\ sessions$$
$$(sessions\ userId?).sessionId = sessionId?$$

$$LogoutUser0 \triangleq LogoutUserRaw \land CheckSessionId$$

As with login, failure to meet the logout conditions results in different types of failure.

UserNotLoggedIn

$$\Xi UserService$$
$$userId? : ID$$

$$userId? \notin dom\ sessions$$

SessionIdIncorrect

$$
\begin{array}{|l}
\Xi \mathit{UserService} \\
\mathit{userId?} : \mathit{ID} \\
\mathit{sessionId?} : \mathit{ID} \\
\hline
\mathit{userId?} \in \mathit{dom}\ \mathit{sessions} \\
\mathit{sessions}\ \mathit{userId?}.\mathit{sessionId} \neq \mathit{sessionId?} \\
\hline
\end{array}
$$

Together with LogoutUser0 these make up the total operation LogoutUser.

$$\mathit{LogoutUser} \triangleq \mathit{LogoutUser0} \lor \mathit{UserDoesNotExist} \lor \mathit{UserNotLoggedIn} \lor \mathit{SessionIdIncorrect}$$

Later on, other services will need to retrieve user profile and session information so we need to define suitable query operations.

$$
\begin{array}{|l}
\mathit{GetUserProfile0} \\
\hline
\Xi \mathit{UserService} \\
\mathit{userId?} : \mathit{ID} \\
\mathit{profile!} : \mathit{UserProfile} \\
\hline
\mathit{userId?} \in \mathit{dom}\ \mathit{profiles} \\
\mathit{profile!} = \mathit{profiles}\ \mathit{userId?} \\
\hline
\end{array}
$$

$$\mathit{GetUserProfile} \triangleq \mathit{GetUserProfile0} \lor \mathit{UserDoesNotExist}$$

$$
\begin{array}{|l}
\mathit{GetUserSession0} \\
\hline
\Xi \mathit{UserService} \\
\mathit{userId?} : \mathit{ID} \\
\mathit{session!} : \mathit{UserSession} \\
\hline
\mathit{userId?} \in \mathit{dom}\ \mathit{profiles} \\
\mathit{userId?} \in \mathit{dom}\ \mathit{sessions} \\
\mathit{session!} = \mathit{sessions}\ \mathit{userId?} \\
\end{array}
$$

$$\text{GetUserSession} \triangleq \text{GetUserSession0} \lor \text{UserDoesNotExist} \lor \text{UserNotLoggedIn}$$

Asset Service

Assets are immutable and opaque blobs of data associated with an ID. Hence, we need a blob data type

[BLOB]

and a simple asset schema

Asset

$$id : ID$$
$$data : BLOB$$

AssetInit

$$Asset'$$
$$id? : ID$$
$$data? : BLOB$$

$$id' = id?$$
$$data' = data?$$

The asset service itself is also very simple, consisting of one function through which assets can be referenced.

AssetService

$$assets : ID \rightarrowtail Asset$$

$$\forall id : dom\ assets \bullet id = (assets\ id).id$$

Assets are injected from some external persistence mechanism when the service is initialized.

AssetServiceInit

$$
\begin{array}{|l}
\hline
\mathit{AssetService}' \\
\mathit{assets}? : \mathrm{ID} \pfun \mathit{Asset} \\
\hline
\mathit{assets}' = \mathit{assets}? \\
\hline
\end{array}
$$

Since assets are immutable, the only operations necessary are those to add them.

$$
\begin{array}{|l}
\hline
\mathit{AddAsset0} \\
\hline
\Delta \mathit{AssetService} \\
\mathit{asset}? : \mathit{Asset} \\
\hline
\mathit{asset}?.\mathrm{id} \notin \mathrm{dom}\ \mathit{assets} \\
\mathit{assets}' = \mathit{assets} \oplus \{ \mathit{asset}?.\mathrm{id} \mapsto \mathit{asset}? \} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\mathit{AssetAlreadyRegistered} \\
\hline
\Xi \mathit{AssetService} \\
\mathit{asset}? : \mathit{Asset} \\
\hline
\mathit{asset}?.\mathrm{id} \in \mathrm{dom}\ \mathit{assets} \\
\hline
\end{array}
$$

$$\mathit{AddAsset} \triangleq \mathit{AddAsset0} \lor \mathit{AssetAlreadyRegistered}$$

And those to retrieve them.

$$
\begin{array}{|l}
\hline
\mathit{GetAsset0} \\
\hline
\Xi \mathit{AssetService} \\
\mathit{assetId}? : \mathrm{ID} \\
\mathit{asset}! : \mathit{Asset} \\
\end{array}
$$

$$
\begin{array}{|l}
\hline
\\
assetId? \in dom\ assets \\
asset! = assets\ assetId? \\
\\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
AssetNotRegistered \\
\\
\Xi AssetService \\
asset? : Asset \\
\\
\hline
\\
asset?.id \notin dom\ assets \\
\\
\hline
\end{array}
$$

$$GetAsset \triangleq GetAsset0 \lor AssetNotRegistered$$

Inventory Service

The inventory service handles the management of user inventories.

Each inventory item has a name and an association with an asset. Items can be copied between users and the copies can change their metadata (such as their name) without the need to duplicate the underlying asset. We'll reflect this by storing an asset ID in the Item schema. The item also has an ID of its own.

Item

$$
\begin{array}{l}
id : ID \\
name : Name \\
assetId : ID
\end{array}
$$

ItemInit

$$
\begin{array}{l}
Item' \\
id? : ID \\
name? : Name \\
assetId? : ID \\
\hline
id' = id? \\
name' = name? \\
assetId' = assetId?
\end{array}
$$

An inventory consists of a collection of items which are associated with a particular user. The user is identified by their ID.

Inventory

$$
\begin{array}{l}
userId : ID \\
items : ID \pfun Item \\
\hline
\forall\, id : \mathrm{dom}\ items \bullet id = (items\ id).id
\end{array}
$$

As with the other services that we've seen, inventory initialization presupposes an external source of inventory data.

InventoryInit

$$
\begin{array}{l}
Inventory' \\
userId? : ID \\
items? : ID \pfun Item \\
\hline
userId' = userId? \\
items' = items?
\end{array}
$$

The inventory service itself can be defined as a function that references all the available inventories.

InventoryService

$$
\begin{array}{l}
inventories : ID \pfun Inventory \\
\hline
\forall\ id : \mathrm{dom}\ inventories \bullet id = (inventories\ id).userId
\end{array}
$$

InventoryServiceInit

$$
\begin{array}{l}
InventoryService' \\
inventories? : ID \pfun Inventory \\
\hline
inventories' = inventories?
\end{array}
$$

Administrators can add or remove entire inventories. In the classic architecture, the inventory service is available only to a single trusted organization so we don't need to worry about preventing untrusted entities from invoking these operations.

AddInventory

$$
\begin{array}{|l}
\Delta InventoryService \\
inventory? : Inventory \\
\hline
inventories' = inventories \oplus \{ inventory?.userId \mapsto inventory? \} \\
\hline
\end{array}
$$

RemoveInventory0

$$
\begin{array}{|l}
\Delta InventoryService \\
userId? : ID \\
\hline
userId? \in \mathrm{dom}\ inventories \\
inventories' = \{ userId? \} \lhd inventories \\
\hline
\end{array}
$$

InventoryDoesNotExist

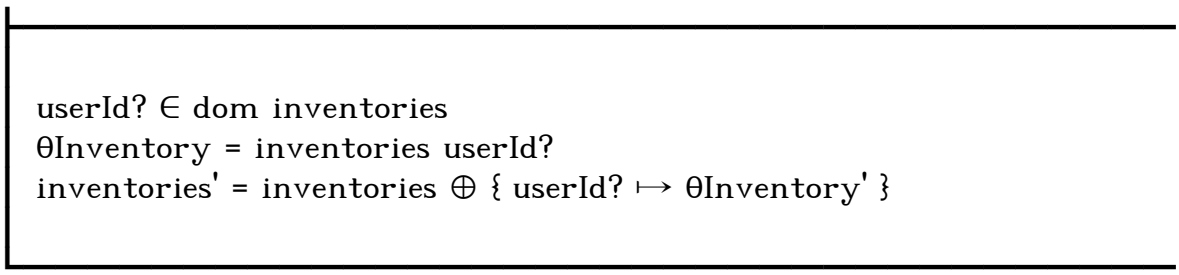
$$
\begin{array}{|l}
\Xi InventoryService \\
userId? : ID \\
\hline
userId? \notin \mathrm{dom}\ inventories \\
\hline
\end{array}
$$

$$RemoveInventory \triangleq RemoveInventory0 \lor InventoryDoesNotExist$$

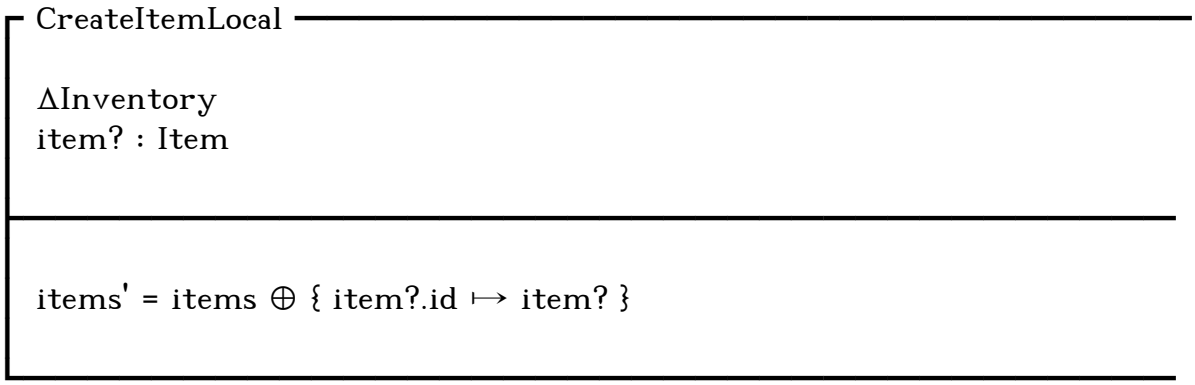
Users will want to create, read and delete inventory items. These requests are relayed from clients via simulators which allow users to execute operations only on their own inventories. To define these operations we need a promotion schema.

PromoteInventory

$$
\begin{array}{|l}
\Delta InventoryService \\
\Delta Inventory \\
userId? : ID \\
\end{array}
$$



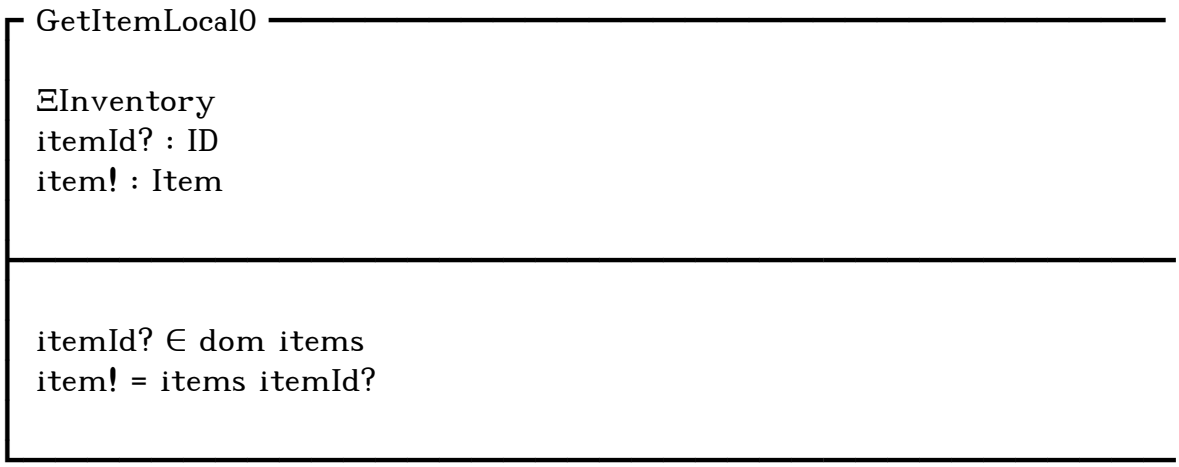
The local operations themselves are straightforward.

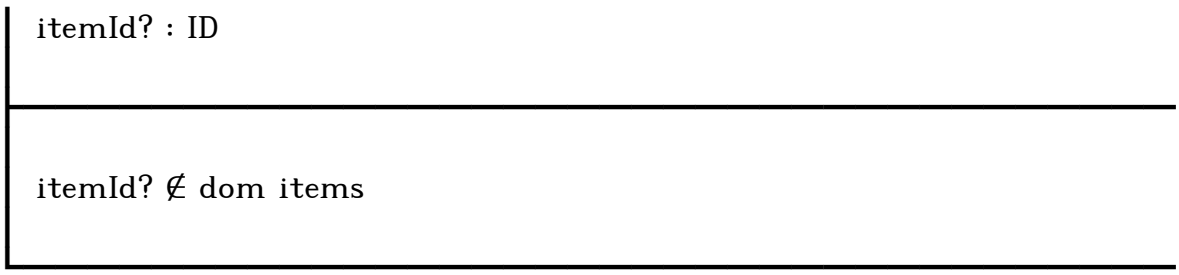


For the CreateItem total operation we'll take into account the possibility that the inventory the caller is trying to manipulate does not exist.

CreateItem ≙ (∃ΔInventory • PromoteInventory ∧ CreateItemLocal) ∨ InventoryDoesNotExist

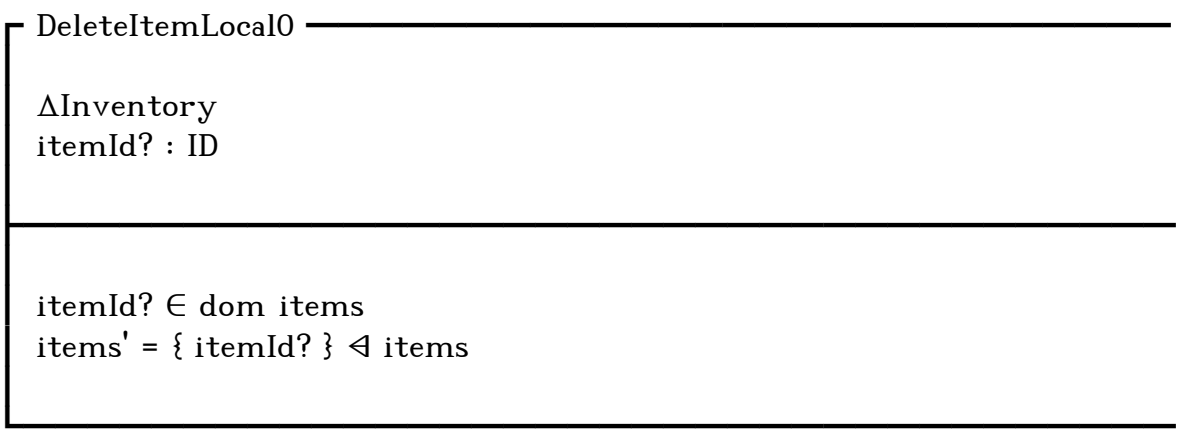
The total operation for retrieving an item via the service is fairly straightforward.





GetItem ≙ (∃ΔInventory • PromoteInventory ∧ (GetItemLocal0 ∨ ItemDoesNotExist)) ∨ InventoryDoesNotExist

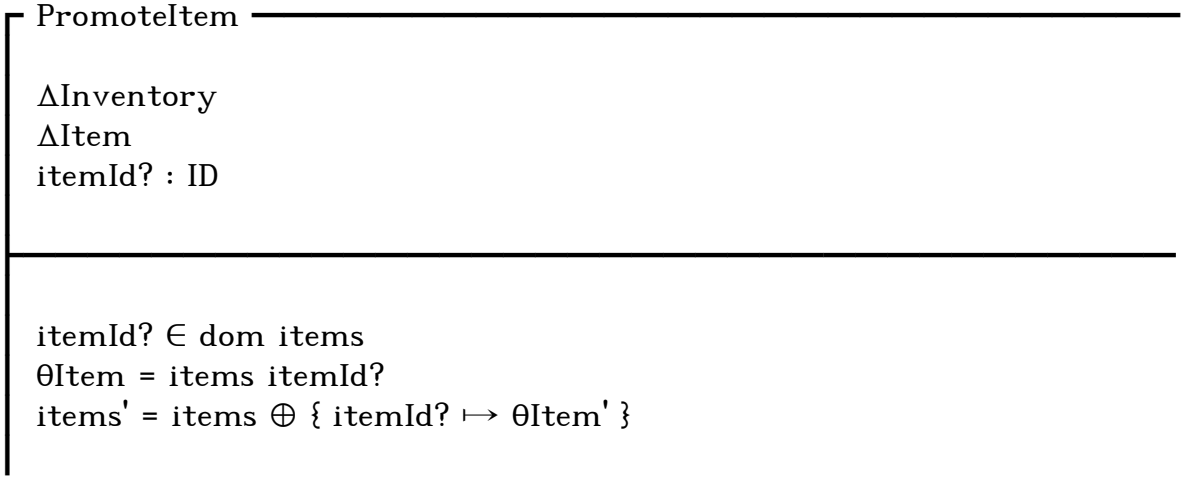
Deleting an item is also fairly simple.



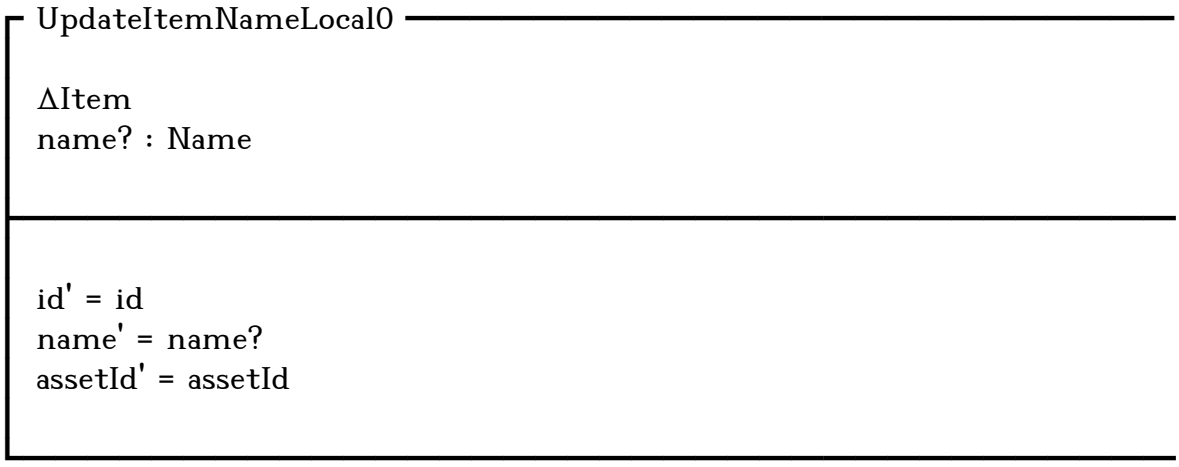
DeleteItem ≙ (∃ΔInventory • PromoteInventory ∧ (DeleteItemLocal0 ∨ ItemDoesNotExist)) ∨ InventoryDoesNotExist

Finally, we need inventory service operations that can update an item's metadata. In our model, both the name and the asset ID can be changed.

To perform operations on individual inventory items we need another promotion schema.



Now we can create separate name and asset ID updating operations.

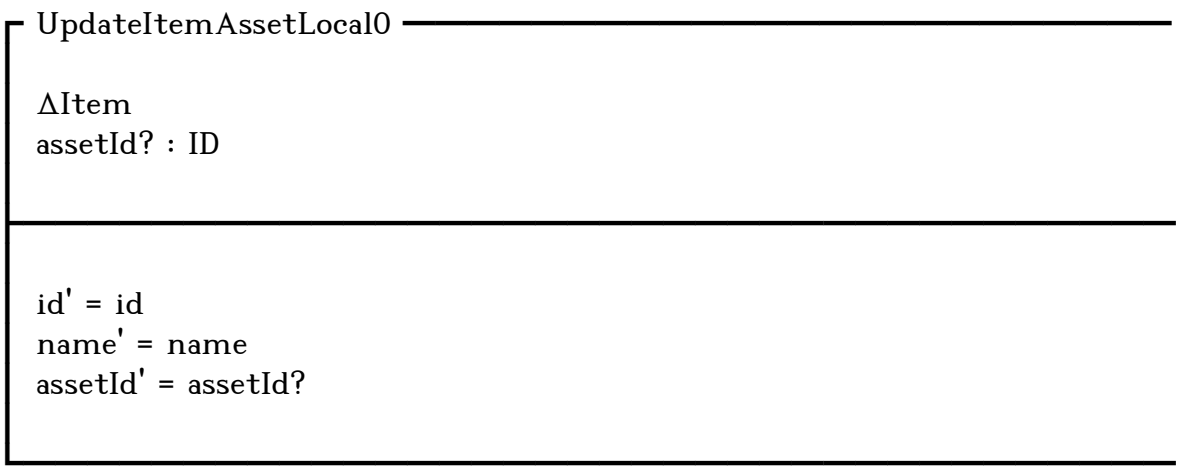


The local item name update operation needs to bubble up through two levels of promotion – one for the item in the inventory and another for the inventory in the inventory service.

$$UpdateItemNameLocal \triangleq (\exists \Delta Item \bullet PromoteItem \land UpdateItemNameLocal0) \lor ItemDoesNotExist$$

$$UpdateItemName \triangleq (\exists \Delta Inventory \bullet PromoteInventory \land UpdateItemNameLocal) \lor InventoryDoesNotExist$$

As you can imagine, the operation for updating an item's asset ID looks very similar.



$$UpdateItemAssetLocal \triangleq (\exists \Delta Item \bullet PromoteItem \land UpdateItemAssetLocal0) \lor ItemDoesNotExist$$

$$UpdateItemAsset \triangleq ((\exists \Delta Inventory \bullet PromoteInventory \land UpdateItemAssetLocal) \lor InventoryDoesNotExist$$

Grid Service

As we discussed previously, the classic OpenSimulator architecture places all regions in a single two dimensional grid. The grid service co-ordinates region positioning and allows neighbouring regions to find out about each other.

In the grid service we will model region records rather than regions themselves. This reflects the fact that simulators and the grid service are often distributed across different machines.

As with many of the other objects in the OpenSimulator system, regions have a unique id. They also have a name and an 2-dimensional co-ordinate.

RegionRecord

$$
\begin{array}{l}
id : ID \\
name : Name \\
x : \mathbb{N} \\
y : \mathbb{N}
\end{array}
$$

RegionRecordInit

$$
\begin{array}{l}
RegionRecord' \\
id? : ID \\
name? : Name \\
x? : \mathbb{N} \\
y? : \mathbb{N} \\
\hline
id' = id? \\
name' = name? \\
x' = x? \\
y' = y?
\end{array}
$$

The grid service holds the name of the grid in addition to information about the regions attached to it. All regions must have unique ids and a unique co-ordinate on the grid map.

GridService

$$
\begin{array}{l}
name : Name \\
regionRecords : ID \pfun RegionRecord
\end{array}
$$

$$\forall s, t : \mathrm{ran}\ regionRecords \mid s.id \neq t.id \bullet \neg (s.x = t.x \land s.y = t.y)$$

$$\forall id : \mathrm{dom}\ regionRecords \bullet id = (regionRecords\ id).id$$

Here is the grid service initialization schema. As with the other services, when a grid service is initialized existing region records come from some external source.

GridServiceInit

$$GridService'$$
$$name? : Name$$
$$regionRecords? : ID \rightarrowtail RegionRecord$$

$$name' = name?$$
$$regionRecords' = regionRecords?$$

The service needs to provide operations to register and deregister regions.

RegisterRegion0

$$\Delta GridService$$
$$regionRecord? : RegionRecord$$

$$regionRecords' = regionRecords \oplus \{ regionRecord?.id \mapsto regionRecord? \}$$

RegionAlreadyRegisteredToService

$$\Xi GridService$$

```
regionId? : ID
```

```
regionId? ∈ dom regionRecords
```

RegisterRegion ≙ RegisterRegion0 ∨ RegionAlreadyRegisteredToService

DeregisterRegion0

```
ΔGridService
regionId? : ID
```

```
regionId? ∈ dom regionRecords
regionRecords' = { regionId? } ⩤ regionRecords
```

RegionNotRegisteredToService

```
ΞGridService
regionId? : ID
```

```
regionId? ∉ dom regionRecords
```

DeregisterRegion = DeregisterRegion0 ∨ RegionNotRegisteredToService

In addition, we need an operation that allows simulators to find out about their hosted region's neighbours. To do this, we first define a function that will give us the absolute value of its input.

```
abs : ℤ ⇸ ℕ
```

```
∀ n : ℤ | n < 0 • abs(n) = -n
∀ n : ℤ | n ≥ 0 • abs(n) = n
```

Then we can use this to find all the relevant region records.

GetNeighbours

$$\Xi GridService$$
$$x : \mathbb{N}$$
$$y : \mathbb{N}$$
$$regionRecords! : \mathbb{P}\ RegionRecord$$

$$regionRecords! = \{ r : ran\ regionRecords \mid \neg (x = r.x \land y = r.y) \land abs(x - r.x) \leq 1 \land abs(y - r.y) \leq 1 \}$$

Simulator

There are two distinct entities that we'll model for the simulation component of the virtual environment. These are avatars (the representations of the user controlled directly by a client program) and simulated objects. In our model, objects will have texture assets associated with them. These are sent by the simulator to the client when the avatar has them in view. We won't model any of the other usual object properties such as dimensions or size since these are not relevant to later analysis.

Object

$$
\begin{array}{l}
id : ID \\
name : Name \\
textures : \mathbb{P}\ Asset
\end{array}
$$

Object initialization can be triggered in a number of ways. Among these are creation from a serialized state stored in a user's inventory and direct creation from scratch, in which case default textures are supplied.

ObjectInit

$$
\begin{array}{l}
Object' \\
id? : ID \\
name? : Name \\
textures? : \mathbb{P}\ Asset \\
\hline
id' = id? \\
name' = name? \\
textures' = textures?
\end{array}
$$

Like objects, avatars are also associated with texture assets that hold their appearance. In addition, the avatar is always associated with a user session.

Avatar

$$
\begin{array}{l}
id : ID \\
user : UserSession \\
textures : \mathbb{P}\ Asset
\end{array}
$$

Avatars are initialized when a user logs in. The initialization data originates from the user service.

AvatarInit

$$
\begin{array}{l}
Avatar' \\
id? : ID \\
user? : UserSession \\
textures? : \mathbb{P}\ Asset \\
\hline
id' = id? \\
user' = user? \\
textures' = textures?
\end{array}
$$

We can think of a region as being a container that holds both objects and avatars. Regions are associated with a set of neighbours, all of which must have unique IDs.

In this model we're not going to worry about such details as where objects or avatars are actually located in a scene since this was not germane to later analysis.

Region

$$
\begin{array}{l}
record : RegionRecord \\
avatars : ID \pfun Avatar \\
objects : ID \pfun Object \\
neighbours : \mathbb{P}\ RegionRecord \\
\hline
\forall\ n, o : neighbours \mid n \neq o \bullet n.id \neq o.id \land n.id \neq record.id
\end{array}
$$

We're going to presume that regions can be initialized with pre-existing objects. However, regions always start empty of avatars.

RegionInit

$$
\begin{array}{l}
Region' \\
record? : RegionRecord \\
objects? : ID \pfun Object
\end{array}
$$

$$neighbours? : \mathbb{P}\ RegionRecord$$

$$record' = record?$$
$$avatars' = \varnothing$$
$$objects' = objects?$$
$$neighbours' = neighbours?$$

Regions are contained within a simulator. As stated previously, a simulator can host one to many regions. Simulators also hold references to all the VE services.

Simulator

$$regions : ID \rightarrowtail\!\!\!\!\!+\ Region$$
$$userService : UserService$$
$$assetService : AssetService$$
$$inventoryService : InventoryService$$
$$gridService : GridService$$

$$\forall\ r, s : \operatorname{ran} regions \mid r \neq s \bullet r.record.id \neq s.record.id$$

SimulatorInit

$$Simulator'$$
$$regions? : ID \rightarrowtail\!\!\!\!\!+\ Region$$
$$userService? : UserService$$
$$assetService? : AssetService$$
$$inventoryService? : InventoryService$$
$$gridService? : GridService$$

$$regions' = regions?$$
$$userService' = userService?$$
$$assetService' = assetService?$$
$$inventoryService' = inventoryService?$$
$$gridService' = gridService?$$

The operations that we want to consider for the simulator involve avatars and objects. For both these operations, we will need a promotion schema that can access the region and an error schema to handle the situation in which the requested region is not available.

PromoteRegion

$$\Delta Simulator$$
$$\Delta Region$$
$$regionId? : ID$$

$$regionId? \in \mathrm{dom}\ regions$$
$$\theta Region = regions\ regionId?$$
$$regions' = regions \oplus \{ regionId? \mapsto \theta Region' \}$$

RegionDoesNotExist

$$\Xi Simulator$$
$$regionId? : ID$$

$$regionId? \notin \mathrm{dom}\ regions$$

The first pair of operations add and remove avatars from regions, which is necessary when a user crosses or teleports in or out. We need an error schema to cover the case where an avatar is already in the region in question.

AddAvatar0

$$\Delta Region$$
$$avatar? : Avatar$$

$$avatar?.id \notin \mathrm{dom}\ avatars$$
$$avatars' = avatars \oplus \{ avatar?.id \mapsto avatar? \}$$

AvatarAlreadyInRegion

Ξ Region
avatar? : Avatar

---

avatar?.id $\in$ dom avatars

AddAvatar
$\triangleq$ ($\exists \Delta$Region $\bullet$ PromoteRegion $\wedge$ (AddAvatar0 $\vee$ AvatarAlreadyInRegion))
$\vee$ RegionDoesNotExist

RemoveAvatar0

ΔRegion
avatarId? : ID
avatar! : Avatar

---

avatarId? $\in$ dom avatars
avatar! = avatars avatarId?
avatars' = { avatarId? } $\lhd$ avatars

AvatarNotInRegion

Ξ Region
avatar? : Avatar

---

avatar?.id $\notin$ dom avatars

RemoveAvatar
$\triangleq$ ($\exists \Delta$Region $\bullet$ PromoteRegion $\wedge$ (RemoveAvatar0 $\vee$ AvatarNotInRegion))
$\vee$ RegionDoesNotExist

The second pair of operations add and remove objects from regions. In a VE these can be invoked for various reasons but the ones that I was concerned about in this project were creation from a user's inventory and removal from the region to a user's inventory.

AddObject0

$$\Delta Region$$
$$object? : Object$$

$$object?.id \notin dom\ objects$$
$$objects' = objects \oplus \{ object?.id \mapsto object? \}$$

ObjectAlreadyInRegion

$$\Xi\ Region$$
$$object? : Object$$

$$object?.id \in dom\ objects$$

$$AddObject \triangleq (\exists \Delta Region \bullet PromoteRegion \land (AddObject0 \lor ObjectAlreadyInRegion)) \lor RegionDoesNotExist$$

RemoveObject0

$$\Delta Region$$
$$objectId? : ID$$
$$object! : Object$$

$$objectId? \in dom\ objects$$
$$object! = objects\ objectId?$$
$$objects' = \{ objectId? \} \lhd objects$$

ObjectNotInRegion

$$\Xi \text{ Region}$$
$$\text{object?} : \text{Object}$$

$$\text{object?.id} \notin \text{dom objects}$$

RemoveObject
$$\triangleq (\exists \Delta \text{Region} \bullet \text{PromoteRegion} \land (\text{RemoveObject0} \lor \text{ObjectNotInRegion})) \lor \text{RegionDoesNotExist}$$

Teleporting an avatar from one region to another can now be expressed as an operation where an avatar is extracted from one region and added to another. To distinctly specify the destination region, we need to replace all instances of regionId? with destinationRegionId? for the AddAvatar operation.

TeleportAvatar
$$\triangleq \text{RemoveAvatar} \gg (\text{AddAvatar} [\text{destinationRegionId?} / \text{regionId?}])$$

Grid

Finally, let us formalize a description for the classic architecture as a whole.

In the classic architecture, a grid is composed of a set of regions. Both avatars and objects can only appear in a single region at a time – for the avatar this restriction restates our earlier login operation constraints where a user can only have one session at a time.

Services are considered part of the grid, though client programs running on user's machines are not.

Grid

$$
\begin{aligned}
&regions : \mathbb{P}\ Region \\
&userService : UserService \\
&assetService : AssetService \\
&inventoryService : InventoryService \\
&gridService : GridService
\end{aligned}
$$

$$
\begin{aligned}
&\forall\ s, t : regions \mid s \neq t \\
&\quad \bullet\ s.avatars \cap t.avatars = \varnothing \land s.objects \cap t.objects = \varnothing
\end{aligned}
$$

GridInit

$$
\begin{aligned}
&Grid' \\
&regions? : \mathbb{P}\ Region \\
&userService? : UserService \\
&assetService? : AssetService \\
&inventoryService? : InventoryService \\
&gridService? : GridService
\end{aligned}
$$

$$
\begin{aligned}
&regions' = regions? \\
&userService' = userService? \\
&assetService' = assetService? \\
&inventoryService' = inventoryService? \\
&gridService' = gridService?
\end{aligned}
$$

The Client-Simulator Protocol

As we've seen, a major aspect of the OpenSimulator architecture is a client-server configuration where the graphical virtual environment client is the client and the simulator maintaining the authoritative VE state is the server.

There are many issues associated with maintaining consistent state between the simulator and its clients. However, we will not cover most of them here since we're concerned with the problem of creating an Internet-scale network of virtual environments rather than scaling a particular instance.

But even leaving these concerns aside, there are areas where OpenSimulator's client-server architecture does impact upon broader issues of scalability and transparency. These centre around the nature of the communication protocol between the client and the simulator.

In the OpenSimulator architecture, the communication link between the client and the simulator must carry a variety of different message types with different delivery requirements. One class of such messages is those which detail with user inventory. These are fairly time insensitive – a delay of 500ms between taking an object from the environment and having it appear in inventory has a negligible impact on the user's experience. But the messages must be delivered reliably - taking an object and failing to see it appear in inventory at all is a significant problem. Messages must be resent in the event of a delivery failure.

At the other end of the scale. messages that update object positions must be delivered rapidly. Failure to do so will result in jerky object movements, leading to degradation of the user's VE experience. But reliable delivery here is not critical. If one message is lost then the next - with a slightly updated position - would replace it anyway.

Because of the rapid update requirement of some messages, information exchange in the client-simulator protocol takes place over the User Datagram Protocol (UDP). UDP is commonly used in time-critical applications such as games and video streaming. Unlike the Transmission Control Protocol (TCP), it offers no message delivery guarantees. But this also means that UDP, unlike TCP, never pauses in the transmission of data for reliability purposes – lost packets are simply dropped rather than retransmitted. Hence, “UDP improves network delay by a factor of ten over TCP/IP” [Singhal and Zyda p48]**.**

The client-server protocol used by OpenSimulator overlays UDP with a simple message acknowledgement system.. Dropped UDP packets deemed particularly important, such as those containing the inventory manipulation messages described above, can be detected and resent.

Another aspect of the OpenSimulator protocol, and of VE client-server protocols in general, is that there is a constant information flow between the client and the server. In all but the most static of simulations, the server is pushing constant updates to the client about the movement of other objects and avatars, and the client is generating its own state change requests for consumption by the server.

This is going to be significant when we come to look at the range of possible Internet-scale VE architectures. The communication style of constant state updates is in marked contrast to that of the Web, as described by Roy Fielding in his 2000 dissertation “Architectural Styles and the Design of Network-based Software Architectures” [Fielding]. Web page requests are discrete, large-grained and occur in response to explicit client requests. Virtual environment updates, on the other hand, are constant and small grained, often pushed to clients that must be ready to process them.

Let's illustrate this by taking a look at the process of avatar movement in terms of the exchange of

messages between a client and a simulator..

When a user moves their avatar via the viewer's user interface, the client translates that into an AgentUpdate message that is sent to the simulator hosting the region that the avatar is in. Here are some of the fields of this message.

Diagram 6: Moving the avatar in the viewer

```
AgentUpdate
{
  AgentID : ID
  SessionID : ID
  BodyRotation : Quaternion
  HeadRotation : Quaternion
  ...
  Flags : Unsigned 8 bit Integer
}
```

AgentID is the user's unique identifier while SessionID is the ID for the session established between

the simulator and the client. BodyRotation and HeadRotation are quaternions that signal, respectively, the viewer's desired body and head rotation for the avatar. Finally, the desired direction of movement is encoded, slightly obscurely, in the Flags field. This can be none, forward, backward, left, right, up or down.

Changing the avatar's place in the simulation isn't as simple as just sending the agent update from the client to the simulator. After updating its central state repository the simulator needs to tell all observing entities and the user themselves about the avatar's new position.

Why doesn't the viewer simply authoritatively tell the simulator the avatar's new co-ordinates, hence eliminating the need for at least the return message? One problem is that any number of things may happen to the avatar in the simulation that the viewer cannot anticipate. For instance, a fast moving object (e.g. a vehicle), may have moved into the avatar's path since the last update, invalidating the movement request. Giving the viewer authority to decide its avatars position also enables it to lie and make movements that ought to be impossible. This is a particular problem if the virtual environment is being used for gaming.

So instead, when an avatar moves, both the client originating the movement and all the other avatars in that region receive a message called an ImprovedTerseObjectUpdate.

```
ImprovedTerseObjectUpdate
{
  Data : byte[]
}
```

In this case, all the important information is packed into a variable size Data variable. This includes the ID of the object (in this case, the user ID that we saw earlier), its position and velocity. Since ImprovedTerseObjectUpdate messages make up the vast majority of the messages transferred from the simulator to the client, the data is tightly packed in order to make the best use of the bandwidth available on the link.

As alluded to earlier, this constant exchange of fine-grained state-containing messages contrasts heavily with the Representational State Transfer (REST) architecture used for the Web. In REST, any session state is held on the client rather than the server [Fielding p78]. This gives REST the architectural properties of visibility, reliability and scalability. Visibility because each request is understandable on its own without extra content. Reliability because lack of state makes it easy to recover from partial failures. Scalability because extra servers can be brought online to handle extra load without any need to communicate state information between them.

The OpenSimulator client-simulator protocol has none of these properties. Each request is not understandable on its own – AgentUpdate takes place in the context of the current position of the avatar known to the simulator. Each message depends upon state held at the simulator, so a failure in the network or on the server side can often mean client disconnection. And scalability is very difficult at the simulation level - bringing extra servers online to support a heavily loaded region would mean constantly replicating state between them. Moreover, the load scales exponentially rather than linearly since every extra avatar needs to communicate with all the other avatars already present in the simulation.

Scaling OpenSimulator

Now that we've described the current OpenSimulator architecture it's time to look at how it could help form the basis of an Internet-scale VE network. I'll start off by stating what I think are the necessary characteristics of such a system. Then I'll look at the extent to which these can be met by the existing architecture and ways in which this architecture could be evolved to form alternative ways of satisfying the requirements. I'll finish with the conclusions from my analysis and a reflection on how helpful Z modelling and concepts from distributing computing have been in producing this project.

Requirements for an Internet-scale VE Network

So far, I've talked quite a bit about the idea of an “Internet-scale” VE network in this dissertation. But what does this term really mean? It's rather difficult to pin down - there's no cut and dried definition out there for the phrase “Internet-scale”. But one thing that we can be sure is that it's “considerably more than just geographical dispersion”, as Fielding states in his dissertation “Architectural Styles and the Design of Network-based Software Architectures” [Fielding, p69].

In fact, Fielding goes on to state two requirements for an “Internet-scale architecture” – those of “anarchic scalability” and “independent deployment”. Anarchic scalability is a general principle that a network design be resilient in the face of the unexpected, whether this comes in the form of extremely high load, malicious input, component failure or some other chaotic event. Independent deployment requires that it be possible to place new components besides old in a running system, allowing the new to take advantage of their advanced functionality without hindering existing operations.

I don't disagree with either of these requirements for an Internet-scale architecture. However, independent deployment is not something that I've focussed on in this project. As stated back in the introduction, I'm concerned here with investigating basic system architectures for a VE network, not with the inter-component communication standards that are crucial for enabling piece-wise network evolution.

Anarchic scalability, on the other hand, is much more relevant to my analysis. Some elements of this – such as dealing with malicious input, I will still ignore since these come down, again, to communication protocol design and message handling philosophy.

But other elements of anarchic scalability, such as robustness in the face of component failure or high load, are much more important to me and arguably form the basis of all the requirements of an Internet-scale VE network architecture that I'm now going to posit. In no particular order of importance, these requirements are:

- **Unbounded number of concurrent users.** We saw in the first section of this dissertation that the classic OpenSimulator architecture lacks scalability transparency at the simulation level. It's not possible to add extra simulators to meet the demands of extra users in a particular area of virtual space – the fixed size regions can only be simulated by one system process at a time.

  However, it's not individual simulation scalability that concerns me here but rather the capacity of the entire network of simulators. At this level, an Internet-scale network can have no hard upper bound on the number of concurrent sessions. Even if one particular simulator is overloaded this should not stop users from interacting with all the remaining operational simulators, much like an overloaded webserver does not stop users interacting with any other website.

- **Independent hosting.** Just as independent organizations and individuals can host webservers with only minimal co-ordination requirements, so it should be possible for independent entities to host their own VEs without any significant need for explicit co-ordination or trust between the different parties.

  If such trust is needed then I think that it becomes effectively impossible to build out an

Internet-scale network. If every existing VE in the system has to vet any new joiner, then the costs involved in adding another VE rise linearly, continually raising the entry barrier for joining as the system expands. The need for explicit co-ordination also smothers the potential for anarchic experimentation and evolution of the system itself.

- **No single point of failure or control.** It will always be possible for particular simulators to become unavailable, whether through high load, as we've already discussed, through network problems or by simple program failure. These problems should not affect other VEs. Again, this is something that we see on the Web – individual websites can fail but this does not make the Web as a whole unavailable.

  More broadly, an Internet-scale VE network must not rely on any single component that could act as a point of failure. In the cases where such reliance is unavoidable, the system being used should itself be distributed, fault tolerant and as simple as possible. One example of such a system that exists today is the Domain Name System (DNS) used to associate IP addresses with domain names [Mockapetris and Dunlap].

  If we don't avoid single points of failure in our system then we continually risk considerable disruption in the face of the inevitable accidents and hardware failures. Single points of failure can also be single points of control, giving a single organizational entity leverage over the entire system. Such a situation exposes the system to many non-technical problems in additional to the technical ones, organizational bankruptcy and a lack of competition to spur innovation, being two examples.

- **Portable identity**. Just as a user can seamlessly navigate via hyperlink between websites hosted by independent operators, so too should it be possible to seamlessly navigate a client between independently hosted VEs in the network.

  In the context of VEs, I believe that such navigation requires that the user's identity be preserved over different systems. This requirement comes from the multi-user and social aspects of virtual environments. If I'm called “Justin Clark-Casey” on one system but then get assigned the name “Purple Guest” when I navigate to another system there's no persistent identity that other places can detect and interact with – I need to be “Justin Clark-Casey” on both systems. More sophisticated scenarios could see the preservation of my user profile information and my avatar's appearance in addition to my name.

  There is some parallel here with the the current situation on the Web. As the Web has matured, the need for websites, in particular shopping and social networking sites, to gather information about a user has grown enormously. Historically, each site has requested, stored and secured this data independently. It's extremely tedious for the user to fill out the same details separately for each website.

  Initiatives such as OpenID[6] have come about in response to allow users to log on to multiple services with the same digital identity. Our distributed network of VEs has a similar requirement, though in this case I consider it to be essential rather than optional as it is on the Web.

6 http://openid.net/

- **Portable inventory.** In the classic OpenSimulator architecture each user has an inventory in which they can store objects for later use. Unlike identity, the notion of making this inventory portable between environments doesn't have much of an equivalent on the Web – one doesn't take programmable objects from one website and instantiate them on another.

  Yet for an Internet-scale VE network, I'm going to argue that portable inventory is essential. Creation of useful objects or systems of objects that make up VE applications involves a considerable amount of work. Being able to move this content between different environments significantly enhances the overall utility of the system and encourages the creation of more such objects and applications by independent developers.

  This wouldn't matter so much if items were copied between systems in the same way that Web application code is today. After all, there are a million different ways in which a Web application can be created, everything from good old-fashioned hand-coded C-based Common Gateway Interface (CGI) programs up to full Web application stacks such as Ruby on Rails. This variety hasn't certainly hasn't held back the Web.

  But in the case of virtual environments, a large proportion of objects are personal to the user rather than part of the VE that they are visiting. Clothing and other appearance items are one example in the social realm. Process and building models are examples in the business realm.

  Therefore, it should be possible to create inventory on one system and use it on a completely independent system subject to the appropriate authorization and permissions.

  This requirement has considerable ramifications. It implies that different VEs understand a common format for object description and content. It also implies a standardized execution environment so that the scripts that drive object behaviour can be written once to run anywhere. This environment must be controlled so that object instantiation and execution is subject to proper authorization.

One requirement that I haven't put on the list is that the VE network be a single contiguous environment, such that all simulators in the entire world-wide system occupy the same 2D grid and simulators next to each other can be seamlessly navigated by simply moving one's avatar between them. This is a property possessed by both Linden Lab's Second Life and the classic OpenSimulator architecture and comes from the idea that the virtual environment is a 'world' that closely models the real one.

However, such an arrangement requires additional architectural complexity, particularly when the environments are spread over the entire Internet rather than concentrated in centralized locations. Therefore, I'm not going to consider this requirement, particularly as I don't believe that the hypothetical benefits of a possibly more immersive virtual experience are essential for a useful Internet-scale VE system.

I'm also not going to consider how infrastructure and VE applications could be funded under different network architectures. Financial transfers are important in many different contexts, whether it's the need for developers to be able to charge for the VE objects that they create, the need

for entities to pay for simulation hosting or the need to fund the storage of data on VE services. And without viable business models no practical Internet-scale VE architecture is likely to be successful. But the issues here are extremely complex and speculative, and I believe that at this stage a useful analysis can be performed without explicitly considering them. So for this project I am treating them as out of scope.

Scaling the Classic Architecture

It's already known that the classic architecture can handle a certain number of concurrent users. Linden Lab's own Second Life installation accommodates over 70,000 simultaneous users at peak periods[7]. The OpenSimulator grids themselves, though much smaller, are still known to accommodate hundreds of people simultaneously.

In the context of this architecture, scaling up means adding more simulators to a single grid. Thus, all simulators on the Internet would need to have a location (i.e. an address) on the same 2D map.

Is it possible to grow the classic architecture to the point where it could potentially accommodate the 1.5 billion people using the Internet as of the end of 2009[8]? We can raise a number of difficulties.

Problem 1 – No Independent Hosting

In the classic architecture, all simulators are operated by a single entity, an entity that also provides the central VE services for those simulators. This is in direct contradiction to our requirement that people can independently operate their own simulations within the network. One answer here is simply to allow this independent hosting within the classic architecture. We'll explore the consequences of this later on in the alternative architecture known as the "open grid".

Problem 2 – Centralized VE Services : Storage and Capacity

The classic architecture routes all asset and inventory requests to the same centralized asset and inventory services. As the number of simulators and users increases, the load on these back end services increases proportionally.

How much of an issue is this? Since the central services are replication transparent, it should be possible to indefinitely expand their capacity. The stateless nature of the services means that it's easy to create extra service instances and load balance service requests between replicas.

However, this expansion would push capacity problems further back to the underlying data storage mechanisms, such as databases, on which the service instances depend. Scaling these through replication may also be possible, though that discussion is outside the scope of this project.

A large and growing number of service requests is not the only issue. As we discussed in our analysis of the classic architecture, OpenSimulator's tenet of asset immutability makes it extremely hard to perform garbage collection on assets without risking the referential integrity of objects and inventory.

To perform garbage collection one would have to identify unreferenced assets by performing a complete inspection of all the possible places where assets could be referenced. This includes direct referencing of underlying assets from user inventory items, referencing by serialized object assets and referencing by objects instantiated on regions.

Inventory and serialized object reference analysis is possible since all the necessary data exists in the centralized VE services. Inspection of asset referencing by regions, on the other hand, is more complicated. Although region data can be kept in a centralized database, it is more common to

7 http://nwn.blogs.com/nwn/2008/09/second-life-con.html
8 http://www.internetworldstats.com/stats.htm

store it on the distributed systems running the simulators themselves. In this case, inspection has to be carried out remotely and machines and their CPU time made available. The situation becomes even more complicated, possibly intractably so, if simulators (or even VE services) are operated by third parties, a point that we'll discuss further when we talk about the “open grid” architecture. Even in the classic architecture, co-coordinating the availability of databases for inspection is a big problem.

Furthermore, even if one entity does own all the services and simulators, the amount of time required to perform an inspection of the system will continue to increase as new assets, inventories and simulators are added.

If we discard the possibility of actually removing data then any system that consists of more than a few simulators will likely face an asset storage issue over time. In terms of physical storage this might not actually be a big problem – magnetic disk storage density continues to improve exponentially over time, outpacing even Moore's law[9]. Organizing, manipulating and retrieving such a large store of data may be a bigger issue.

Problem 3 – Centralized VE Services : Single Point of Control

Even if it is technically possible to scale the VE services, issues of control and evolution come into play if one tries to expand the classic architecture up to an Internet level.

Under the classic OpenSimulator system, all the VE services are controlled by a single entity. In some domains this is considered acceptable – the Internet Corporation for Assigned Names and Numbers (ICANN), for instance, is the only organization that manages the assignment of domain names and IP addresses for the Internet. However, not only is ICANN a non-profit organization but it also performs a purely administrative function in a system that is well established. Central Opensimulator-style VE services, on the other hand, are extremely new, untested even over the short-term and they manage data services that deal with commercially and culturally sensitive intellectual property and user information.

A single set of central services would give the operator a monopoly over all these functions, severely restricting their incentive to innovate. Even if they were to innovate, they would only be able to concentrate on a limited number of development approaches, whereby a more distributed model would allow a much greater degree of competition and innovation.

Moreover, a single service point forms a weak point in any Internet-scale system. Even when we discount technical reasons for outages, there are many non-technical causes of failure, ranging from organizational failure to disputes between the central organization and client entities. Indeed, Linden Lab's implementation of a centralized architecture has not been bulletproof. There have been many instances in their environment of outages where asset and inventory services have failed.

Having a single central set of services for an Internet-scale architecture would be in marked difference to the philosophy prevailing for the Web and the Internet as a whole. Both the Web and the Internet are highly distributed architectures. Individual websites can fail and cause content to be unavailable but there is no single centralized system failure that would make the Web as a whole unavailable. Equally, in a virtual environment network we cannot have a situation where the failure of a single services makes all environments unavailable or severely degraded.

For all these reasons I don't believe that expanding the current classic architecture is a viable

9 http://www.scientificamerican.com/article.cfm?id=kryders-law

solution for producing an Internet-scale network. So we'll now explore some alternative approaches which can meet some or all of our requirements for such a system.

Alternative 1 – Multiple Classic Architectures

One response to the problem of scaling the classic architecture is not to change anything as all. Rather, we could simply retain the same system but establish many entirely separate grids, each with their own private VE services. This is illustrated in diagram 7.

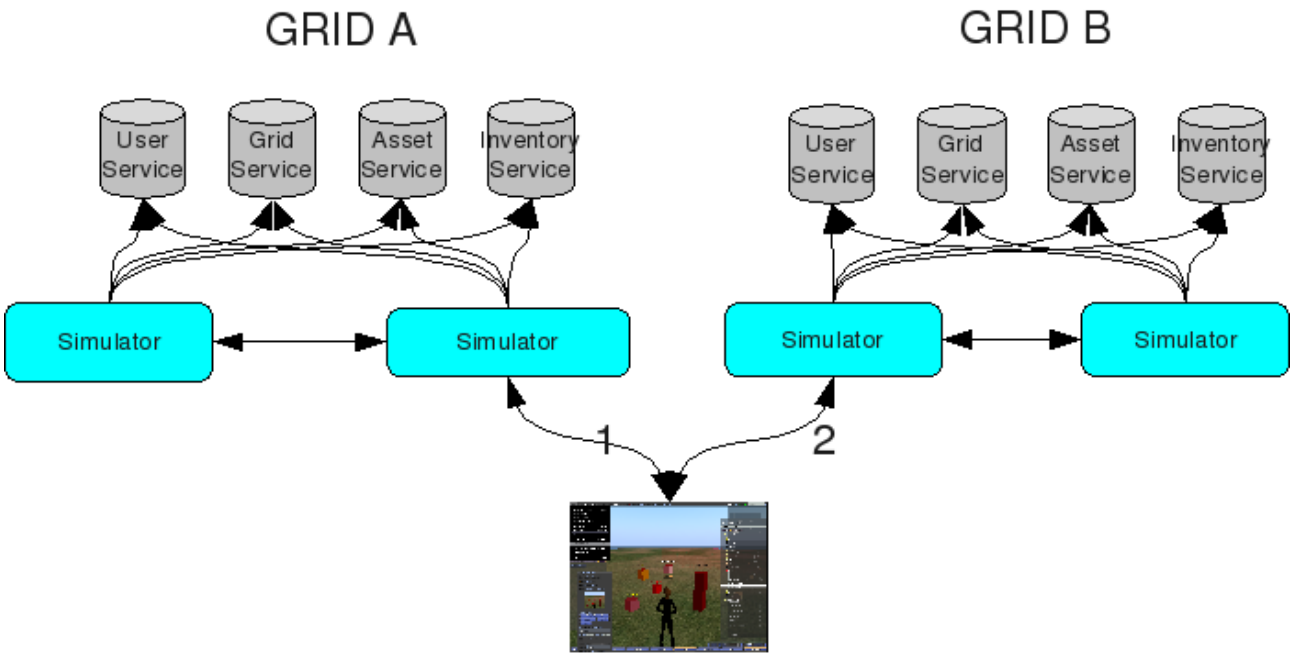


Diagram 7: A user moving between two separate classic architectures

The arrangement does solve some of the problems with simply scaling a single classic grid instance. Different organizations and individuals can own separate grids, removing both the independent hosting problem and the possibility of single points of control. Since each grid has its own set of back-end services, storage and capacity problems for asset data is spread out among OpenSimulator installations rather than causing scalability problems in one place.

Unfortunately, this isn't a solution that meets our requirements for portable identity and inventory. As you can see from the diagram, grid A and grid B are entirely separate from each other. If the user is located in a region simulator in grid A (connection 1) and wants to move to a region simulator in grid B (connection 2), they have to log out of grid A and separately log in to grid B. Grid B has no knowledge of the identity that the user has established on grid A – the user must manually recreate their identity and keep it in sync across all the VE installations that they visit.

Much the same applies to the user's inventory. When in grid B, the user has no access to the inventory available to them on grid A and vice versa.

Nonetheless, I think that the idea of establishing entirely separate systems to avoid scalability problems is a good approach. Many of the architectures that we consider later on revisit this idea with solutions that attempt to satisfy the requirements of portable identity and inventory.

Alternative 2 – The Open Grid

A different approach to scaling a VE network is to maintain a single set of central services but allow third parties to hook up their own simulators. This is illustrated in diagram 8.

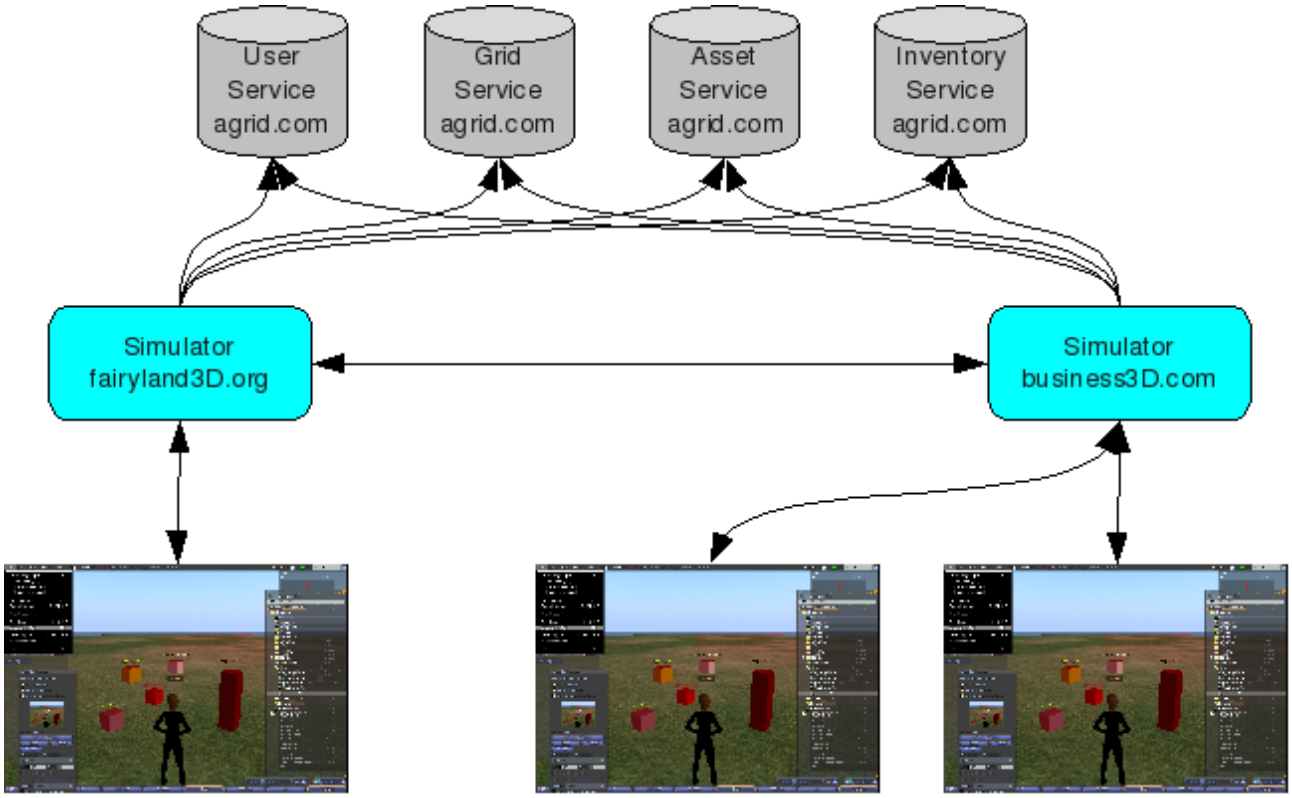


Diagram 8: Third party region simulators using a common set of central services

In the diagram there are two simulators. One is owned and operated by fairyland3D.org while the other is owned and operated by business3D.com. However, both are hooked up to the a single set of services owned and operated by another organization, agrid.com.

This allows independent hosting of simulators while retaining the benefits of portable identity and inventory provided by the classic architecture. When users travel between simulators they are always access identity and inventory information from the same set of data services.

However, this approach does continue to incur all the problems that we've associated with centralized VE services – the services need to be able to handle the entire Internet-scale system and they form a single point of control. In addition, there are some other issues that are thrown up by this architecture.

1. **The grid operator trusts all simulators connected to the services**. In the classic architecture, simulators and services can trust each other completely since they are operated by the same entity. Any simulator can manipulate any user's inventory as long as they know that user's name or ID. Any simulator can retrieve any asset data as long as they know its ID. And there are no limits to how many requests a simulator can make on a grid service or how much data they can store in the asset service. If the services are opened up to third party simulator operators then there is a very real danger that they will abuse these facilities.

One could try to counter this problem in various ways. For instance, the services operator can put limits on how much asset data can be uploaded from a particular simulator and monitor the service requests to prevent abuse or denial of service attacks.

Perhaps the most awkward issue is the unfettered access that simulators have to user inventories. This access is necessary since all interaction by the client program with the classic OpenSimulator architecture passes through the simulator system. Thus, simulators can always to add, retrieve and delete items for any user. We'll look at an attempt to address this security problem in a later alternative architecture that I've labelled "Session ID Security".

A final move that the services operator could make is to heavily vet potential simulator operators before they are allowed to hook their systems up to the services. However, this violates my earlier requirement that independent hosting be possible without any significant need for trust between the parties. Requiring such trust would hand a lot of control over to the services operator and massively increase the cost of building out the network.

2. **Heterogeneous simulator quality.** Another consequence of opening up the classic architecture to third party simulators is that the grid operator is no longer able to guarantee service quality across the entire VE.

   In the classic architecture, the single operating entity hosts simulators on machines of known specification and such machines can be hosted in the same data centre to guarantee high quality network links between them.

   But on an open grid, simulators can be hosted on machines of varying performance – scripts that execute quickly in one simulation may execute very slowly on another. The network links to simulators may also vary considerably – on one region clients may received object textures from the simulator quickly but on another these downloads might be extremely slow. Simulator's whose network links that fail completely may disappear from the grid map or, even worse, remain but be the source of frustrating teleportation and movement failures for users.

   Yet in many ways this kind of variable performance is no different from the Web, where individual Web servers can be overloaded or have network connectivity problems. The percentage of webservers that suffer this is sufficiently tiny for users to ascribe difficulties to individual websites rather than to the Web as the whole.

   The grid operator can take some steps to improve the situation in an open grid – notably by not locating simulators operated by different organizations next to each other in virtual space. This eliminates any network and simulator related issues that occur when a user tries to fly directly between regions hosted at different places on the Internet.

   But what the operator of a genuinely Internet-scale open grid could not do is refuse to provide service access on the basis of simulator hardware or network connection. Though it might be tempting to allow only fast machines with high bandwidth on to the network in order to improve the perception of the entire VE, to do this would be to exercise a monopoly

power that may well end up holding back growth and evolution.

Alternative 3 - Session ID Security

In our Z model of the classic architecture, the user logout operation only proceeds if the correct session ID is given. We can extend this session ID check to all the inventory operations in order to try and secure them against arbitrary use by 3rd party simulator operators. If this can be done then it would eliminate some of the significant security problems associated with the open grid architecture.

We can express this extra check in Z by reusing the existing CheckSessionId schema. Thus, the user inventory operations become.

$$CreateItemSID \triangleq (CheckSessionId \land CreateItem) \lor SessionIdIncorrect$$

$$GetItemSID \triangleq (CheckSessionId \land GetItem) \lor SessionIdIncorrect$$

$$DeleteItemSID \triangleq (CheckSessionId \land DeleteItem) \lor SessionIdIncorrect$$

$$UpdateItemNameSID \triangleq (CheckSessionId \land UpdateItemName) \lor SessionIdIncorrect$$

$$UpdateItemAssetSID \triangleq (CheckSessionId \land UpdateItemAsset) \lor SessionIdIncorrect$$

For clarity, let's graphically step through the GetItemSID operation, beginning with diagram 9. In step 1, the user requests a login via the user service, giving their user ID and password. If the login is successful, the user service creates and tracks a random session ID for that user (step 2). This session ID is passed back to the client (step 3).

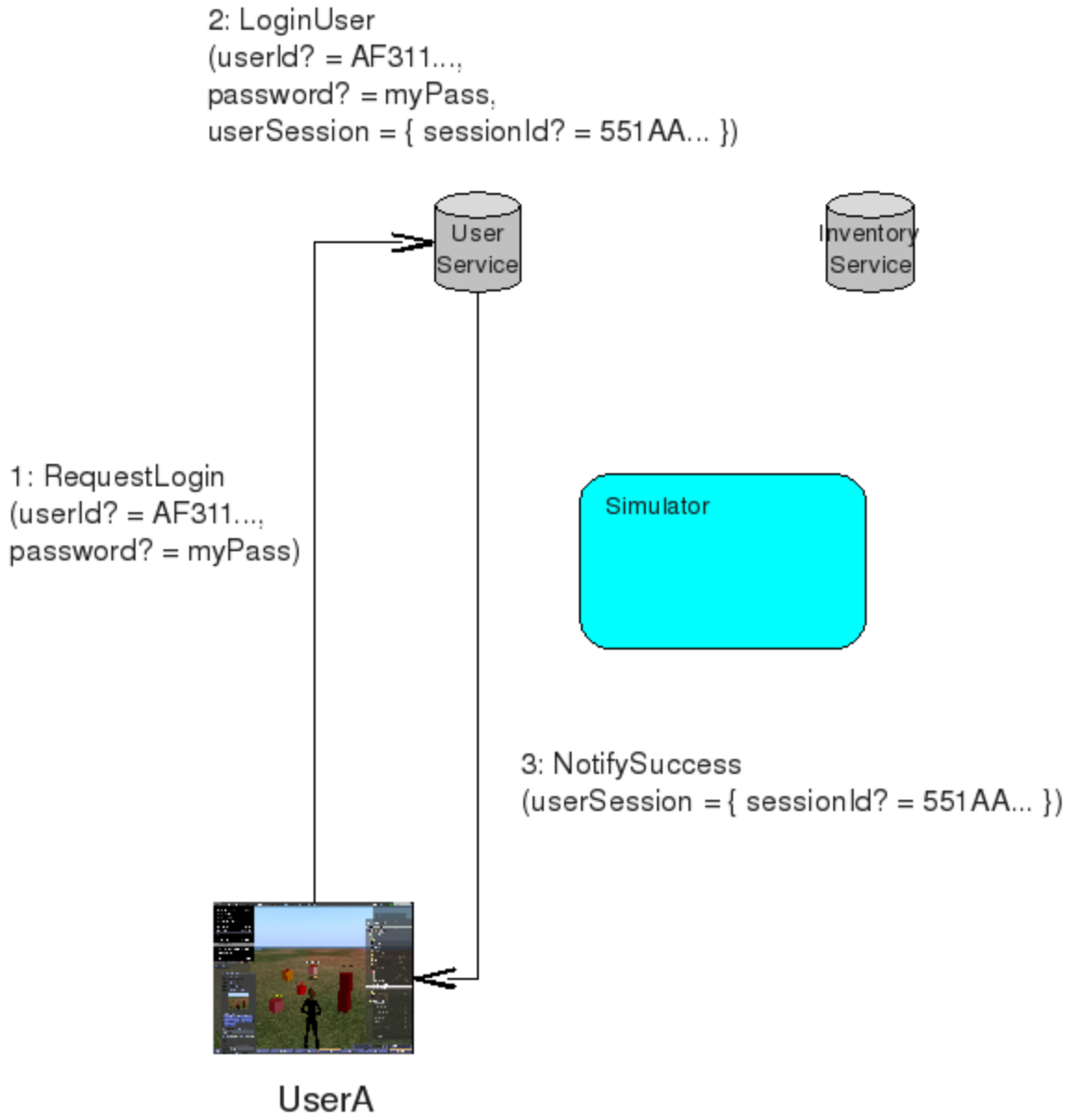


Diagram 9: User login

Now the inventory service will check the session ID every time an inventory operation is attempted. The session ID supplied by the viewer must match the one previously assigned and stored by the user service.

This is portrayed in diagram 10. First, the user makes a request to fetch the details of an inventory item. The client passes this request to the simulator via the GetItem() message, attaching the user ID, item ID and session ID (step 1). The simulator relays this request to the inventory service (step 2). The inventory service uses the CheckSessionId() method to check with the user service whether the session ID is valid (step 3). In this case it is, since it matches the session ID that the user service previously stored.

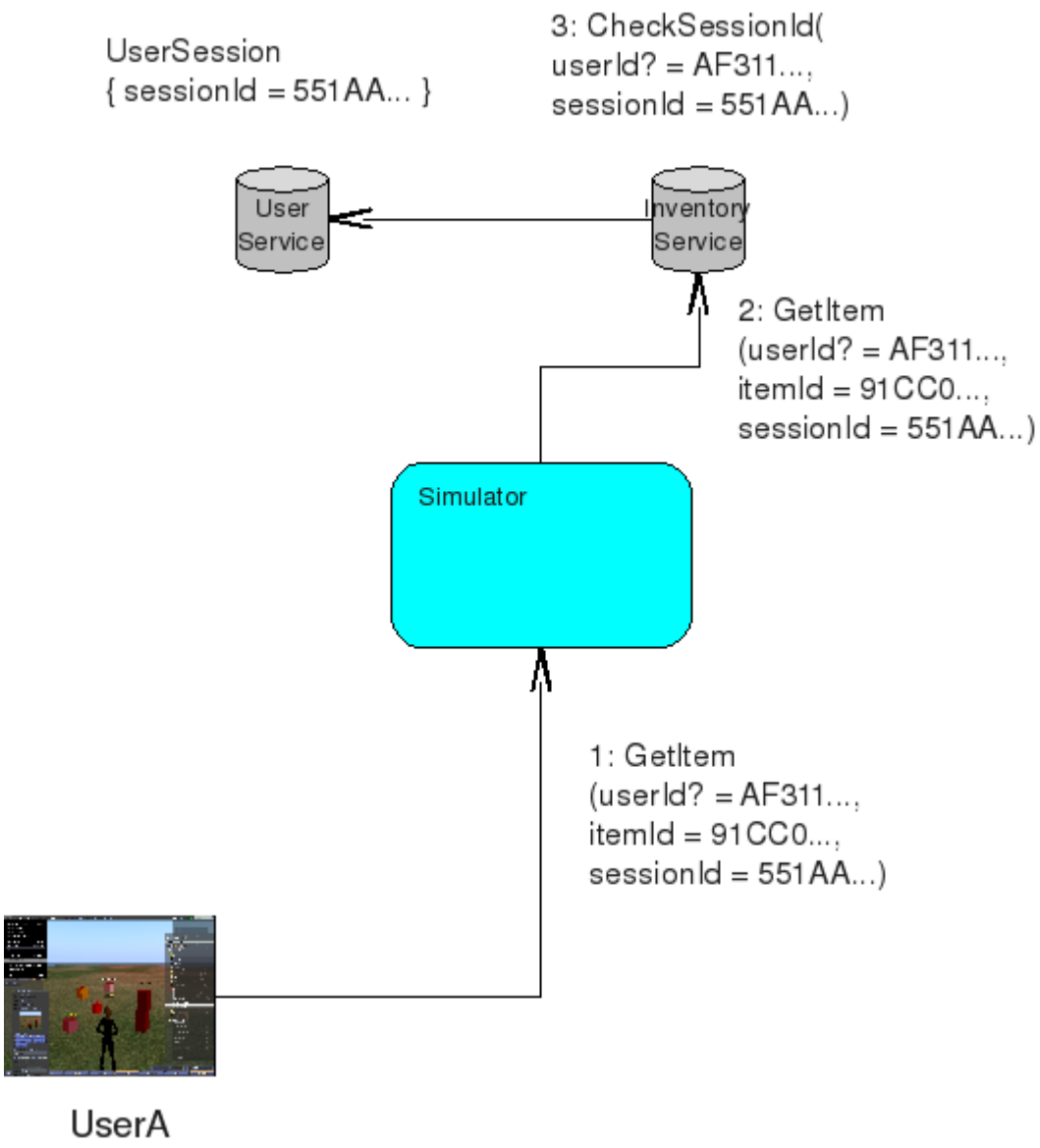


Diagram 10: Checking the session ID for an item request

Diagram 11 shows the return path. Since authentication was successful (step 1), the inventory service passes the details of the requested item back to the simulator (step 2). The simulator in turn passes this back to the client (step 3).

If the session IDs didn't match, then either an error would be sent to the viewer or more likely the request would be silently dropped, since non-matching session IDs probably indicate an attempt at unauthorized access and it would be better not to provide feedback to an anyone attacking the system.

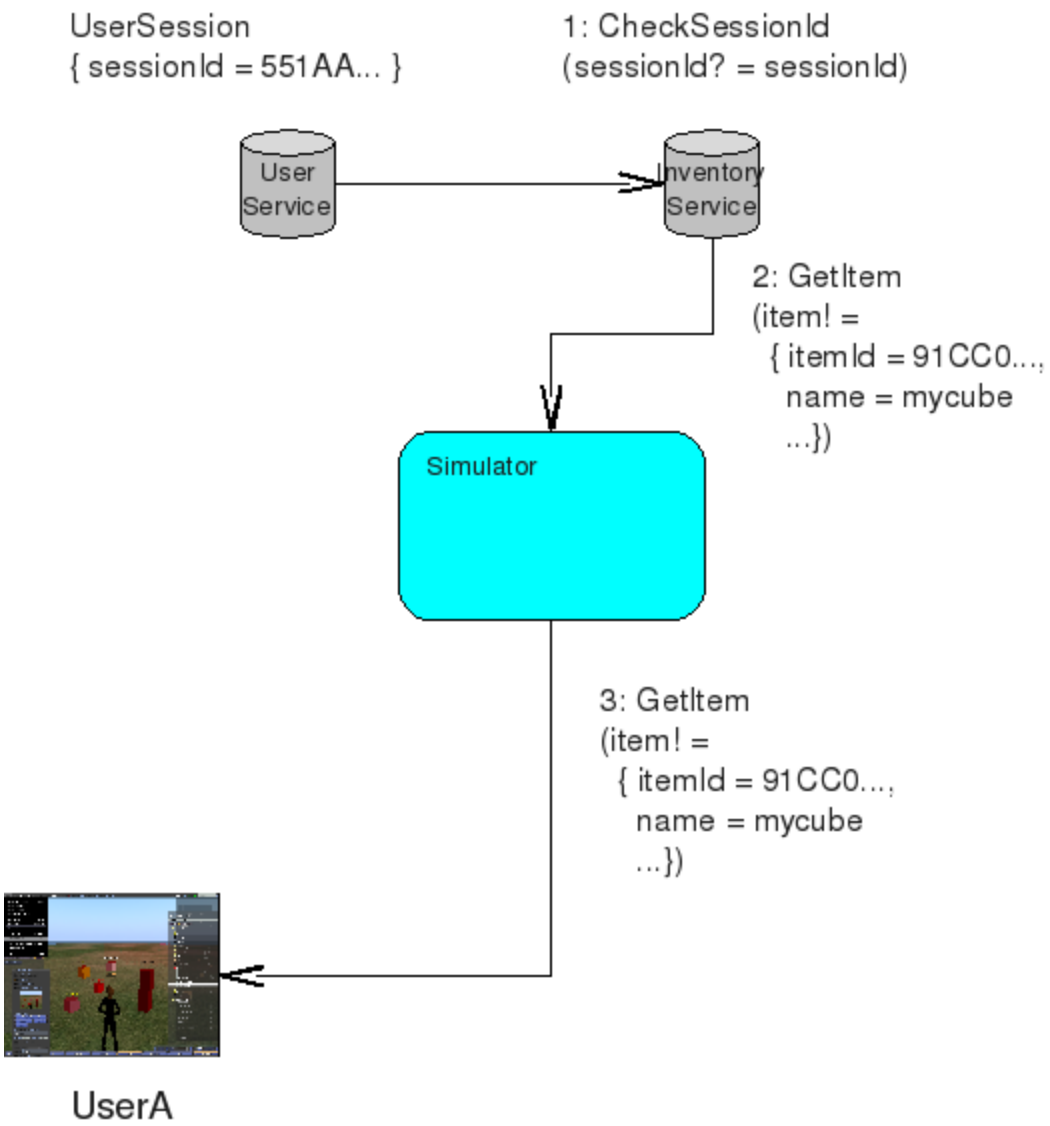


Diagram 11: Return of item data on successful session ID validation

This extension of session ID checks to inventory operations makes it impossible for simulator operators to carry out inventory operations for users that aren't logged in. Even if a user is logged in, the checks ensure that inventory operations cannot be carried out from regions that they have not visited during that session.

These checks are not cost free. Under the session ID architecture, the actions required to carry out an inventory operation are no longer confined purely to the inventory service. Instead, the inventory service has to contact the user service in order to check the session ID for every inventory request. Extra network operations increase the time of response and increase the fragility of the system, though much of this could be avoided by caching the user session IDs on the inventory service after any initial inventory operation for a particular user.

A more important problem with this scheme is that it doesn't offer any protection if a user accidentally goes to a simulator operated by a malicious entity. The simulator will receive the user's current session key just like all the others visited by that user. Even when the user leaves, the malicious simulator can still perform actions on the user's inventory until that user logs out.

Is this any different to the problem of visiting malicious websites with a Web browser – sites that attempt to comprise the security of the visiting computer by exploiting security weaknesses in the client? Users are advised to avoid such sites and regularly update their software to close security exploits. Malicious sites are an issue but they are not considered a terminal threat to the general architecture of the Web.

I would argue that in an Internet-scale VE network the problem of malicious simulators is worse than on the Web, for the following reasons:

1. **Maliciously obtaining a person's inventory merely requires them to visit the region** - it does not rely on the presence of a security hole on the user's local system. Therefore, the user can only protect themselves by avoidance, rather than by keeping their client software up to date.
2. **A user cannot directly detect when their inventory has been compromised.** Whereas a the compromise of a PC through a browser exploit can often be detected by the scans of good up-to-date security software, it would be much more difficult for a user to detect whether their inventory has been compromised in the session ID architecture, as long as the compromiser does not insert new inventory items or remove existing ones.
3. **Accidentally visiting a malicious simulator is much easier than accidentally visiting a malicious website**. On the Web, malicious websites are relatively unlinked compared to non-malicious ones. In this OpenSimulator architecture, the regions of malicious simulators are just as visible as any other region. Therefore, a user is less likely to avoid them just through their relative obscurity.

The only apparent way to counter the problems within the terms of this architecture is to increase the level of trust that one has in a simulator operator before they are allowed to join the VE network. But this again brings back the very problem of required trust that this architecture aimed to relieve in the first place.

In any case, even if the session ID architecture did completely solve the problem of the need to trust all simulator operators, the fact that it takes place within the context of a centralized VE architecture means that the other problems associated with that system – the scalability questions and single point of control issues – remain. So we'll now look at an architectural scheme that revisits the idea of multiple grids with independent data services but looks to allow users to move between them while retaining their identity and inventory.

Alternative 4 - The Hypergrid

The Hypergrid is the name of an alternative OpenSimulator architecture put forward by Christa Lopes of the University of California at Irvine[10]. It aims to allow the direct teleport of a user from one OpenSimulator installation to another while allowing them to retain a single identity and inventory. This is illustrated in diagram 12.

In this architecture, each simulator in a particular OpenSimulator installation retains a connection to that installation's asset, inventory and grid services. But the user service connection is now associated with the user rather than the simulator. The user also retains separate connections to the asset and inventory services. These services we'll call the user's home services – they'll continue to store any objects he takes into his inventory as well as his identity.

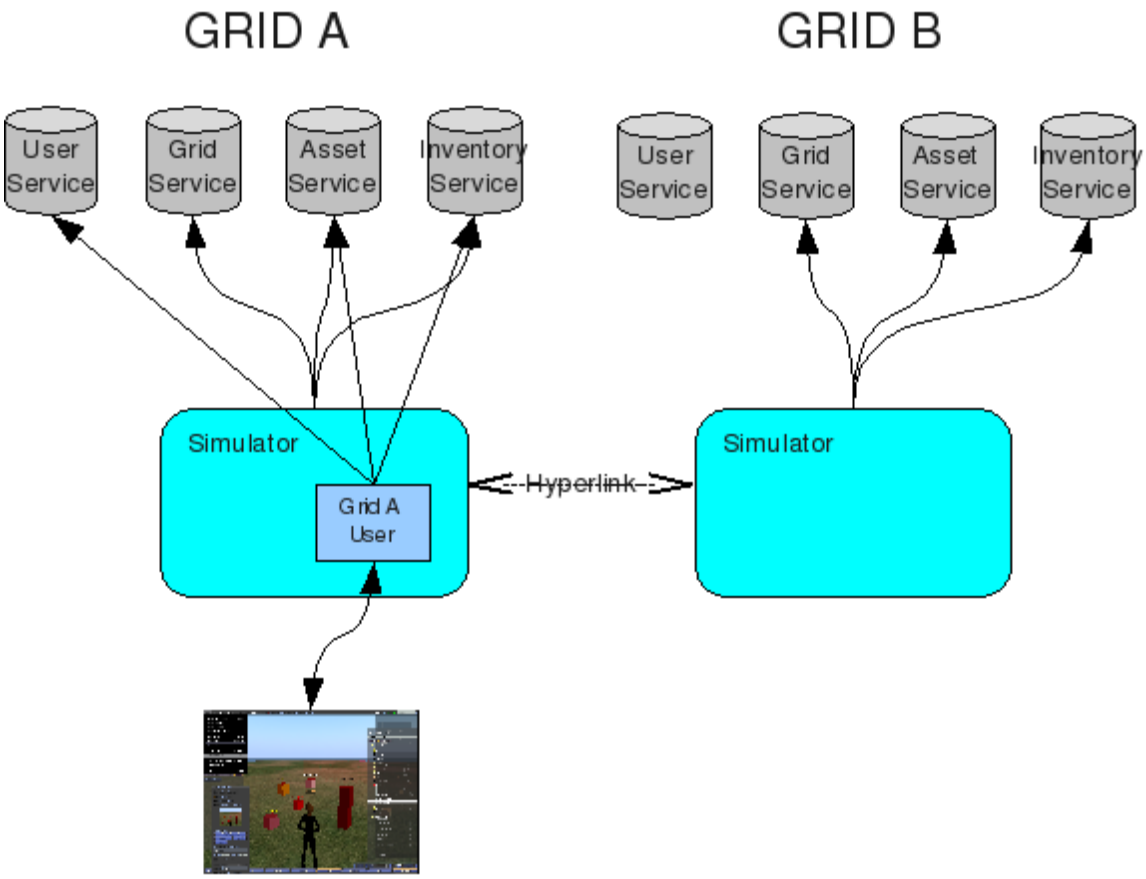


Diagram 12: A user in grid A with a link to a region in grid B

So let's suppose that a user has created an account on grid A and has logged in to a region hosted by a simulator on that grid. This is shown on the left half of diagram 12 – the simulator has connections to that grid's grid, asset and inventory service while the user has connections to the user, asset and inventory services.

The diagram also shows that a hyperlink has been established with grid B, an OpenSim installation that hosts simulators with connections to grid B's services. When the user on grid A accesses a grid map, they can see grid B's regions as well as grid A's. The user can teleport to a region in grid B

10 http://opensimulator.org/wiki/Hypergrid

either by using this map or by entering a URL for the region.

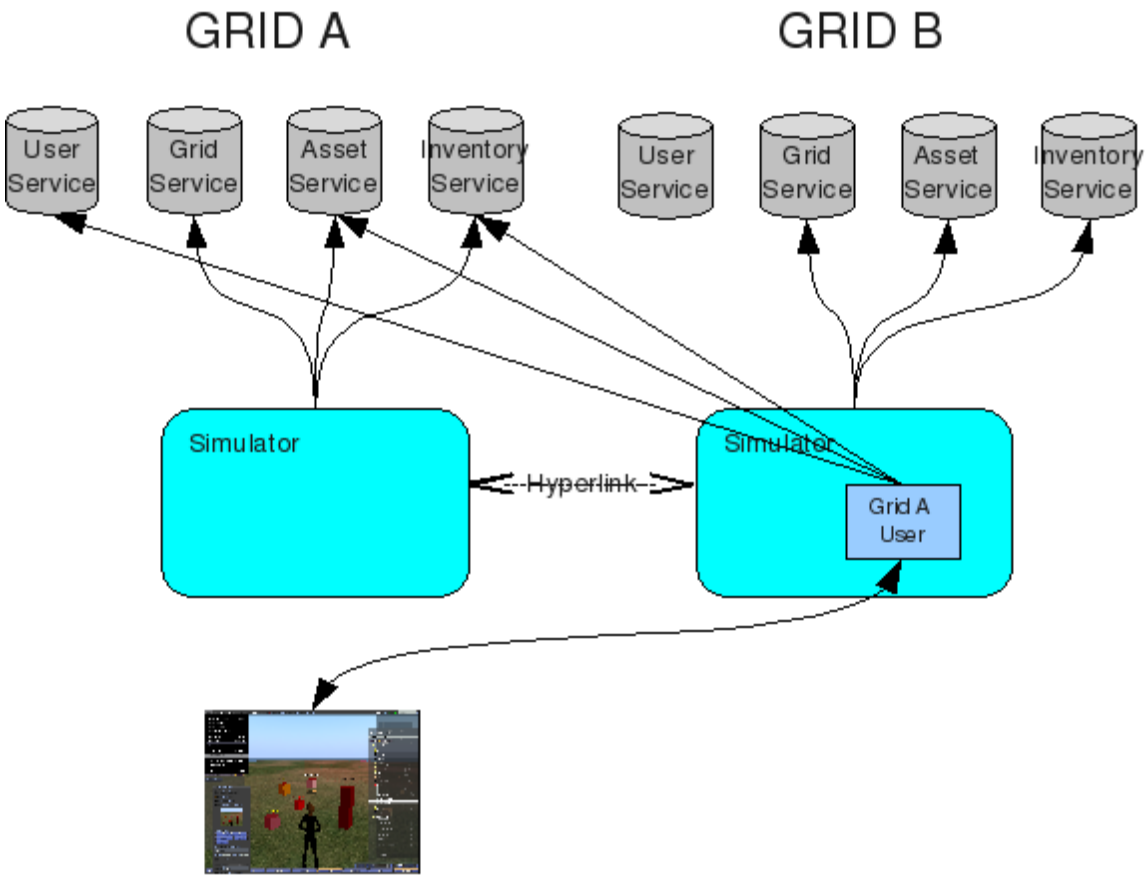


Diagram 13: A successful teleport of the user to grid B

Diagram 13 shows the result of such a teleport. When the user arrives at the grid B region they retain their connections to their home user, asset and inventory services – from grid B's point of view they are a foreign user. If a foreign user rezzes an object in a simulator hosted by grid B, the simulator contacts the user's home inventory and asset services rather than its own. The asset data necessary to support the object - such as sounds, scripts and textures - is copied from the home asset service into grid B's local asset service. This allows the data to be sent to other users when the original object instantiator is no longer online.

Maintaining the connection to the home user service allows a user to make changes to their profile data no matter what region they are in. The simulator can also obtain the most up to date profile information to show to other users.

Like the strategy of establishing completely separate grids, the Hypergrid enables the use of multiple VE services - each grid uses its own set of services for its associated simulators and users. If a particular service fails then only simulators and users who are using that services as their home service will suffer problems – other grids will be unaffected.

However, unlike the separate grids strategy, by maintaining home service connections the Hypergrid architecture enables user identity and inventory portability.

We can formulate the Hypergrid as a Z schema that would replace the Grid schema from the classic architecture.

$$
\begin{array}{l}
regions : \mathbb{P}\ Region \\
userServices : \mathbb{P}\ UserService \\
assetServices : \mathbb{P}\ AssetService \\
inventoryServices : \mathbb{P}\ InventoryService \\
gridServices : \mathbb{P}\ GridService \\
\hline
\forall\ s,\ t\ ;\ regions \mid s \neq t \\
\quad \bullet\ s.avatars \cap t.avatars = \varnothing \land s.objects \cap t.objects = \varnothing \\
\forall\ u,\ v\ ;\ userServices \mid u \neq v \\
\quad \bullet\ u.profiles \cap v.profiles = \varnothing \\
\forall\ i,\ j\ ;\ inventoryServices \mid i \neq j \\
\quad \bullet\ i.inventories \cap j.inventories = \varnothing \\
\forall\ g,\ h\ ;\ gridServices \mid g \neq h \\
\quad \bullet\ g.regionRecords \cap h.regionRecords = \varnothing
\end{array}
$$

Here, we capture the fact that in the Hypergrid a user has a single set of home services in which their identity and inventory is maintained – this information is not duplicated to other locations. However, there is no such restriction on asset services – data must be replicated between them in order for the system to function.

The Hypergrid schema also enforces the restriction that a particular region record cannot occur twice, otherwise navigation between regions would become a problem. Avatars and objects can only be at one location on the Hypergrid at a time since all simulators are joined into a single network.

The biggest issue with the Hypergrid, as with the open grid, is the potential exposure of sensitive service operations. Service connections exposed to foreign simulators can be abused to perform arbitrary operations in just the same way as they independent simulator operators can abuse them on an open grid. When the user travelled from grid A to a simulator on grid B, the simulator on grid B gained access to the user's entire inventory, asset collection and profile.

One could respond to this with the same user session ID checks and trust restrictions that we've previously discussed. But once again, these are either of very imperfect effectiveness or severely restrict the Internet-scale independent hosting goal that we are trying to achieve in the first place.

So where do we go from here? Many of the problems we've seen arise from the fact that all service operations are carried out via the simulator in which the user's avatar happens to be situated. This makes it very hard, if not impossible, to secure these operations against malicious operators. The next architecture that we'll consider will look to address this by moving service operations away from the simulator altogether.

Alternative 5 - Direct Client Services

We've seen that, while interesting, both the Session ID and the Hypergrid approaches are flawed solutions to the problems associated with scaling OpenSimulator to the Internet level.

Both try and fix deficiencies in the classic architecture purely at the server end. But both fall down because they can't eliminate the need for services to trust simulators with private user and inventory data.

So another tack is to shift the responsibility for sensitive service interaction from the server directly to the client. This is shown in diagram 14.

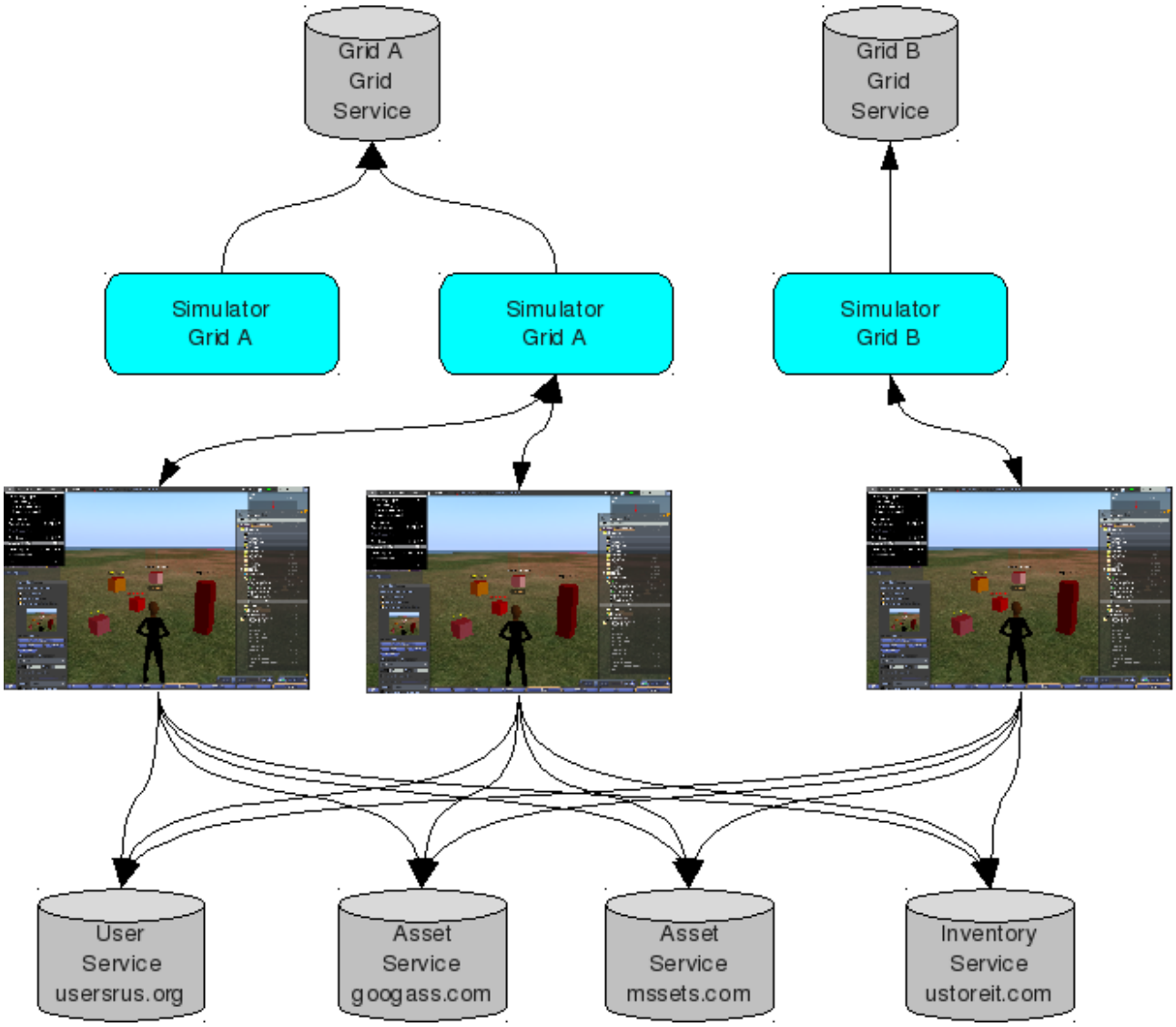


Diagram 14: Direct use of asset, inventory and user services by clients

As you can see, instead of going through the simulators for asset, inventory and user requests, clients make these requests directly to the services themselves, as shown at the bottom of the diagram. Simulators no longer have any need to connect to asset, user or inventory services.

So under this scheme, user, asset and inventory services can now be operated completely independently of the simulators. Moreover, any number of instances of each service can exist – the diagram shows two asset services from which assets can be drawn depending on where the original asset creator chose to store the data. The location of inventory and user data also depends on where the user chose to establish their persistent identity and inventory.

To make direct service interaction possible, the unique UUIDs used to refer to assets, users and inventory in the current architecture need to mutate into something more expressive. They need to remain unique but also encode a service location so that a client that receives one knows from where to fetch asset, inventory or user data. One approach to achieve this would be to incorporate a UUID into a URL. For example,

`http://googass.com/1a1fd6d0-7d30-11de-8a39-0800200c9a66`

could denote a VE object texture stored by the asset service operating on googass.com.

When a user visits a simulator and the object comes into view, it receives this URL rather than just a UUID. Instead of requesting the data from the simulator, it requests the data directly from the URL.

Just as with the Hypergrid, clients are free move between arbitrary simulators, possibly by typing URLs into an address bar in a manner similar to a Web browser. But unlike the Hypergrid, assets for avatar appearance or item rezzing are not copied between asset services. Rather, when the client receives the ID for an asset, it always fetches the data directly from the relevant asset service. It should still be possible for the client itself to cache this data, provided that the immutability property for assets is maintained.

Let's compare this architecture against our original requirements for an Internet-scale VE network.

- **Unbounded number of concurrent users.** There's no single service in this architecture that holds all the virtual environment data. Client and simulator data can be spread out amongst multiple services so there is no restriction at the network level on the number of concurrent users.
- **Independent hosting**. As with the open grid and hypergrid architectures, independent entities can host their own simulators. But in this architecture, simulators do not proxy sensitive user and inventory requests, hence cutting out the security issue suffered by the previous two approaches.

  The direct client services architecture, like the hypergrid, also supports multiple independently owned services.
- **No single point of failure or control**. As there is no bottleneck service in this architecture there is no single point of failure or control.
- **Portable identity.** In this architecture, user IDs would be URLs. Any client can obtain these from a simulation and fetch profile data by dereferencing the URL. Only the user that owns the data will have the credentials required to change it.
- **Portable inventory.** The client always has direct access to its inventory service at all times (which in principle could simply be part of the client). When a client rezzes an inventory item to a region, only the basic object data and the URL's for it's data components (such as textures and sounds) are sent.

What are the downsides to the direct client services approach? One issue concerns the introduction of undesirable asset URLs into regions, such as pornographic textures or malicious scripts. In the classic centralized architecture this can be countered by deleting the asset in question from the centralized asset service. The same is true of the Hypergrid – even though there are multiple asset services, the asset actually used by a simulator is always a copy that has been transferred to its home service. This service remains under the control of a grid administrator who can simply delete malicious assets.

However, in the direct client services architecture, the asset comes from a service chosen by the user rather than the grid. Therefore, offensive images cannot be removed at source – references to them have to be removed manually.

In addition, the use of URLs rather than plain UUIDs opens up a new attack vector. For instance, an attacker controlling their own asset service could infiltrate innocuous images into various regions around the VE network. Once this is complete, he could change the image from something innocent to something malicious. Subsequent clients would receive the malicious data when they fetched assets from those URLs.

I don't think that these problems are fatal for this architecture. In particular, the issue in the last paragraph could be countered by taking copies of the assets that are rezzed to a region, pointing subsequent client requests to the home asset service rather than the one that was originally encoded at rez-time.

So, to me, the direct client services approach appears to be very promising. However, there's one final alternative architecture that I'd like to consider that takes a very different view of VE networking.

Alternative 6 - Live Entity State Stream (LESS)

Up until now we've effectively assumed that all systems in an Internet-scale VE network are OpenSimulator instances. We made this simplifying assumption in order to avoid questions over such things as differeringscripting environments, different graphical rendering approaches and different simulated object serialization formats between VE systems.

But a healthy Internet-scale network will not consist solely of OpenSimulator installations. On the Web, though the Apache project has the largest installed base of webservers today, other packages such as Microsoft Internet Information Server (IIS) also have a significant market share[11].

While there are different webserver implementations, they all conform to the Hypertext Transfer Protocol (HTTP) standard for retrieving hypermedia. The hypermedia itself is also written in terms of standards such as Hypertext Markup Language (HTML) and Javascript so that it can be correctly interpreted by different Web browsers.

Will an Internet-scale VE network have the same standards-based universality, allowing a wide-range of different clients, server and service implementations to interoperate? Naturally, there are great advantages if this is the case – visiting different simulators, for instance, is far easier using a single client than if different clients need to be used for different systems. Common standards allow content to be easily exchanged between many buyers and sellers, promoting the development of increasingly valuable applications.

However, not everybody would agree that this level of interoperability is a certain. There's a countervailing argument that various vendor's virtual environment systems differ so much from each other that common connection and content standards are impossible. For instance, there already exist significantly different approaches for modelling the geometry of VE objects. As of early 2010, Second Life has an exclusively primitive-based approach to this while Blue Mars, another virtual world, uses mesh-based objects. There are also competing approaches to other fundamental VE features such as maintenance of state between servers and clients (or whether there are even any servers at all).

I'm not going to take an explicit position on the likelihood of an interoperable future in this project. But if one supposes, for whatever reason, that standardization is not possible, then there is still an approach to VE networking that circumvents the need for wide-ranging agreement on VE standards and protocols.

This approach is called the Live Entity State Stream, formulated by Jon Watte, the CTO of Forterra Systems[12]. Unlike the other architectural approaches that we've seen, it does not assume agreed standards of communication and content between different VE implementations. For rather than seeking to enable movement of a user between systems, it concentrates on replicating and continuously synchronizing a bounded space between VEs over a point to point link. This is the virtual environment equivalent to setting up a teleconference with somebody in another country rather than flying out to meet them in person.

To illustrate, let's suppose that an appropriately bounded space has been set up on VE A and VE B linked with the LESS protocol, as portrayed in diagram 15. In this situation, a cube (cube1) that exists in VE A at point (2, 2, 2) in the bounded space will have a replica in the bounded space of VE B at (2, 2, 2). State changes between the spaces are replicated. If cube1 is moved to (4, 4, 4) in VE

11 http://news.netcraft.com/archives/2009/06/17/june_2009_web_server_survey.html
12 http://www.interopworld.com/mmox-less-protocol

B, then the cube in VE A will also move to (4, 4, 4).

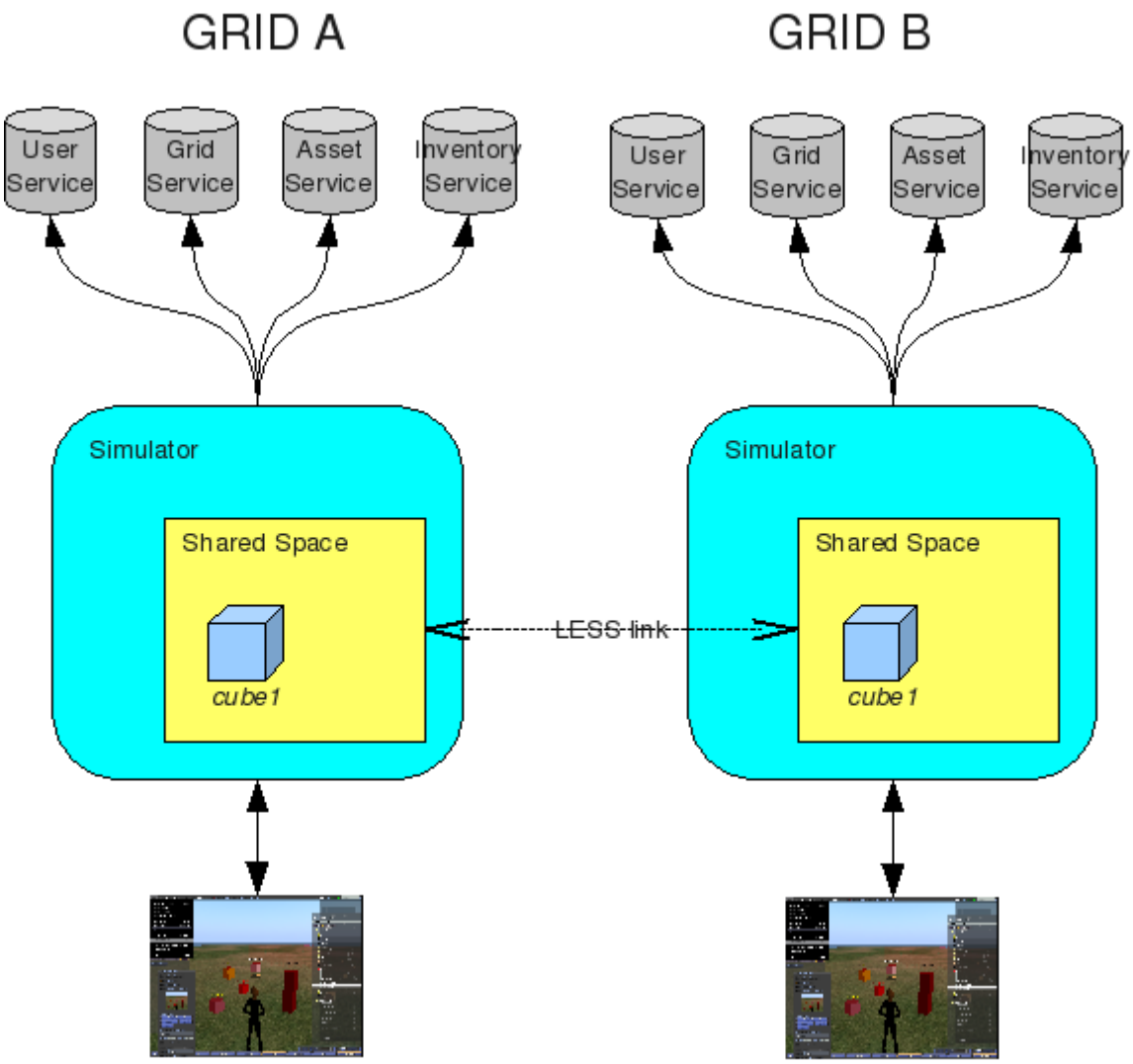


Diagram 15: LESS link between grid A and grid B

As well as position information, all the necessary data to replicate an object's appearance, such as texture assets, is transmitted over the link.

When a user avatar enters the bounded space a suitable avatar 'double' would be generated on the remote system. As soon as they move out of the bounded space the double will disappear from the remote end of the link.

One issue is whether there can really be a synchronization relationship between two replicas of the same object. What happens if both ends attempt to move the same replicated object at the same time? If an object has scripted behaviour on which system does its program execute? Is any form of physics simulation possible on replicated objects? These and other questions may make it simplest to institute a master-slave relationship between object replicas, though that will lose some of the benefits of interactivity.

Nonetheless, the LESS approach is interesting because it might allow existing VE systems built on

different principles to interoperate without having to harmonize on extensive common standards. Such existing alternative systems include Sun's Project Wonderland, Croquet and Forterra System's own On-Line Interactive Virtual Environment (OLIVE) platform. All these already have their own software ecosystem and customers which makes radical change on their part to meet a common standard difficult.

The major disadvantage with the LESS approach is that, just like a real-life teleconference, the interaction is limited and confined to a very narrow location. Step outside the bounds of that location and communication ceases. Communication would not be very rich if there are difficulties sharing intricate objects. And the LESS approach requires time and effort every time a link needs to be set up.

Moreover, while it meets the requirements of an unbounded number of concurrent users, independent hosting and the removal of single points of failure, the LESS approach completely gives up on the ideals of portable identity and inventory.

Conclusions

There are good reasons to suppose that the classic OpenSimulator architecture is not suitable for expansion to an Internet-scale virtual environment network, according to the criteria that we have put forward. It doesn't allow independent hosting of simulators, necessary for competition and innovation. It also has storage and control bottlenecks in the form of a centralized control over VE services.

Of the various alternative architectures, multiple classic grids are not suitable because they fail to provide any portable identity or inventory. Nor is the open grid suitable architecture, since every independent simulator operator would need to be strongly trusted by the service provider (and transitively by all other simulator operators). The Session ID approach to securing inventory doesn't work because it only addresses a very narrow range of security issues associated with server side inventory, while leaving other problems untouched. And hypergrid is not a complete solution because, like the open grid, it has security problems associated with exposing services to arbitrary VE installations.

LESS, though also an interesting approach, ultimately does not address the core issues of an Internet-scale virtual environment network as we've defined them, chiefly the possibility of maintaining a user's identity and inventory access between multiple different virtual environments.

The direct client services architecture appears the most promising for an Internet-scale VE system since it allows many independently operated simulators and services to exist in the same network. It also resolves the security problems inherent in proxying user and inventory service access through the simulator.

Reflection

In this dissertation I applied ideas from both distributed computing and Z-schema based specification and design.

On the distributed computing side, part of my analysis of the classic Opensimulator architecture was conducted within the framework of the dimensions of transparency. This was extremely useful in illuminating the design of the current system and the areas in which it falls short of transparency ideals. For instance, it became clear during the analysis that while the VE services met many transparency conditions, the simulators themselves can only achieve the very lowest levels of transparency.

In my analysis of the classic architecture I also made considerable use of the design ideas of the Web as laid out by Fielding in his “Architectural Styles and the Design of Network-based Software Architectures” dissertation. This was particularly important when thinking about the broad features of OpenSimulator's client-server protocol as its features contrast considerably with those of the protocol operating on the Web.

The dimensions of transparency and Fielding's dissertation were also tremendously useful when it came to thinking about scaling virtual environments. The ideas informed the Internet-scale VE network features that I posited and came into play when analysing the extent to which these were met by the current architecture and alternative approaches.

My experience with my use of Z modelling was more mixed. Explicitly formulating schemas was very helpful in paring down the classic architecture to its most crucial features. The process was iterative – I would formulate a schema, carry out some (non-formal) analysis and then go back to eliminate aspects of the model that were not important for the purposes of the project. This would also work the other way around - alternative architectural analysis would reveal some feature that I previously hadn't thought as important and I would go back and adapt the Z schemas to include it.

This meant that the schemas ended up as a precise description of the most salient points of the classic architecture. I think that this is very helpful in conveying to the reader information about the system, providing that they understand Z schemas! In this, I think that the schemas played a very similar role to UML diagrams.

However, I didn't end up using Z for more than description - the vast bulk of my analysis in this project has ended up being prose-based and informal. My original hope was that I could use the Z model to explore extending the current architecture and creating alternative architectures in a formal way. Although I did end up formulating some extensions in Z for the session ID and Hypergrid architectures these do not employ formal reasoning to justify conclusions or arrive at new ideas.

This was something of a disappointment. But on reflection I believe that it was inevitable. The thrust of this project was very broad, taking an overview of the problem of generating an Internet-scale VE network and trying to pick out future directions rather than tightly analysing or specifying future systems. Generating enough concrete detail to employ formal reasoning would require a vast amount of speculative work that I feel would not be useful given the current embryonic state of the field.

Appendix A - Glossary

Term	Definition
Asset	Media data. This includes textures, sounds, scripts and notecards.
Avatar	A representation of a user in the simulation.
Classic Architecture	The existing OpenSimulator VE architecture where a set of distributed simulators all connect to a centralized set of services.
Client	The program run on the user's computer that is responsible for allowing them to examine and interact with the simulation.
Grid	See Classic Architecture.
Object	An object, such as a cube, represented in the simulation.
Rez (verb)	The act of instantiating an object in the simulation, whether directly from pre-existing primitives or from an existing object stored within a user's inventory.
Service	A system that provides data to one or many simulators. Examples include asset services (that provide data objects such as textures and scripts) and inventory services (that provide inventory organization for a user).
Simulation	A simulation of a place. Can be in 2D or 3D. Contains objects, avatars and other rules and elements such as physics and weather. Operated by a simulator.
Simulator	A server that operates the simulation and interacts with a client. Part of the VE.
Teleport	A movement of an avatar between simulations that involves a discontinuity rather than smooth movement through a 3D environment. This discontinuity often takes the form of a black 'loading' screen.
Virtual Environment (VE)	The set of servers necessary to provide simulated environments to users. This consists of one or many simulators and one or many services. Users connect with client programs.

Bibliography

[Elson and Howell] Jeremy Elson and Jon Howell. *Handling Flash Crowds from your Garage*, Microsoft Research, 2008, http://www.usenix.org/event/usenix08/tech/full_papers/elson/elson_html/index.html

[Emmerich] Wolfgang Emmerich. *Engineering Distributed Objects*, Wiley, 2000.

[Fielding] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*, University of California, Irvine, 2000, http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[Mockapetris and Dunlap] Paul V. Mockapetris and Kevin J. Dunlap. *Development of the Domain Name System*, Proceedings of SIGCOMM '88, Computer Communication Review VoL 18, No. 4, August 1988, pp. 123-133, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.116.9920&rep=rep1&type=pdf

[RFC 4122] P. Leach, M. Mealling and R. Salz. *A Universally Unique IDentifier (UUID) URN Namespace, RFC 4122*, Internet Engineering Task Force, 2005, http://www.ietf.org/rfc/rfc4122.txt

[Singhal and Zyda] Sandeep Singhal and Michael Zyda. *Networked Virtual Environments Design and Implementation*, ACM Press, 2000