Technologies For Building Open-Source Massively Multiplayer Games

Al Riddoch alriddoch@zepler.org
James Turner james@worldforge.org

June 1, 2002

Open Source and Free Software have so far failed to make inroads into the games industry, where intellectual property is still seen as essential for successful business. The WorldForge project aims to break into the massively multi player portion of the games market, where subscription payments outweigh software sales revenue, and a development model based on Open Source or Free Software can work.

Through solid software engineering principles, reusable code, and Open Source peer reviewed development, WorldForge have developed protocols, tools and frameworks for developing large scale on-line game worlds. The Atlas protocol provides a solid, scalable, extensible, game independent protocol for on-line games, and aims to provide for generations of on-line games to come. The STAGE server framework provides a solid base on which games can be developed using the Atlas protocol. In introduction to the Atlas protocol is given here, with an explanation of its strengths for game development and demonstrations of its use in a working game, followed by an overview of the STAGE server framework's development. The challenges of large scale game servers are discussed, including database, networking and performance issues.

1 Introduction

Large scale on-line games have proved to be very expensive to develop in the proprietary games industry, and to ensure subscriber retention, new features and content are required on a regular basis. In a co-operative open source or free software project, it is often difficult to match the resources of a well funded proprietary project, so an approach to development must be taken that ensures that most effective use can be made of the resources available, and work is reused wherever possible. In order to achieve these goals, the WorldForge project has set out to design a protocol for on-line games which is sufficiently flexible that it can be used in many different types of game, and extensible enough that it can accommodate new game features and still be used in future generations of game. It is hoped that the successful implementation of this protocol will mean that game software will reusable in many games, rather than starting each game from scratch as is common in the industry. The Atlas protocol has now been in development for more than 3 years, and has seen successful deployment in the WorldForge demo game, Acorn. In part I the authors will give an overview of the protocol in its current form, with reference to its use in Acorn, and in part II a discussion of more advanced ideas under development in the next generation of game server software.

Part I
Protocols

2 The Atlas Protocol

2.1 Atlas Overview

Atlas is a multilayer protocol. At its top level it is an object oriented remote procedure call system; at the middle level it provides a data format for arbitrary structured data, and at its lowest it provides transport independent communication and codings. The core driver of Atlas development is Aloril, with the help of numerous members of the WorldForge project. It has been defined for use in many types of system, including virtual world systems used for non entertainment purposes, however the main goal of WorldForge is to build character oriented large scale multi player games. Typically the player controls an avatar in the game world, which is represented to the player via a client as a graphical scene. The game world itself is held on one or more servers, and clients are connected to the servers via a network.

2.2 High Level Atlas

At the top level, Atlas defines objects and a mechanism to transmit them across a network connection. These objects can be of two fundamental types, operations and entities. An entity type object represents any kind of entity, both in game entities such as characters and out of game entities such as user accounts. An operation type object represents an event or message. The objects are defined as inheriting from a definition object, which is known as a class in the case of entities and an operation definition in the case of operations. Definitions exist in an inheritance tree and all inherit from a root definition, which is abstract. The definition objects can be represented in the same form as their instances and can be transmitted and stored in the same way. Client server communication is achieved by sending operations between the client and server along the network connection, using a codec to serialise and deserialise them.

An operation has an argument list which usually contains one or more high level Atlas objects i.e. entities or operations. An example of an operation with an entity argument is a move operation, sent to an in game entity to indicate that it is moving. The argument would be an entity with the id of the in game entity that is moving and the attributes necessary to describe the movement. An example of an operation with an operation as its argument is a sight operation, sent to all in game entities when the move operation above occurs. The argument would be the move operation described above, which in turn has the entity that is moving as its argument.

2.3 Low Level Atlas

At the lower level, Atlas uses data in the form of messages to represent the high level objects described above. A message can be one of five types: an integer, a float, a string, a list or a string mapping. A high level object is represented by a message which is of type map, which contains various messages keyed on strings as the operations attributes. In addition to being transmitted, these messages can be used to store and manipulate data such as object properties, arbitrary procedure arguments and persistent structured data of any kind. A convention has been adopted in Atlas documentation, which will be used here to represent low level Atlas data structure using javascript like syntax, as it is well suited to representing the data types involved clearly. Here are some examples of simple messages represented using the javascript like format produced by the Bach codec.

```
42                             An integer
32.85                          A float
"foo"                          A string
[69, 24.8, "bar"]              A list
{name: "foo", height: 5.4}     A map
```

This encoding has been designed for maximum readability, and is used for examples, and human editable data. Other codecs are preferred for efficient network transport, but Bach can be used when snooping the network traffic for debugging purposes is desirable. Other codecs that have been implemented include packed ascii, xml and binary.

The following Bach represents a high level object.

```
{
 objtype: "obj",
 parents: {"pig"},
 id: "pig_42",
 loc: "world_0",
 pos: [0.5, 3.4, 9.7]
}
```

The opening { indicates that this is the start of a map message. The first attribute **objtype** indicates that this is an entity object, the second **parents** attribute indicates that it inherits from an object called pig, which in this case is an entity class. This means that this entity represents a pig in the game world. It is possible for an object to be derived from more than one definition or class, but this feature is not exploited in current implementations. The **id** attribute is an identifier which uniquely identifies this entity on a given server, and the format of the id in the above example is one used for easy debugging in development code. The other two attributes describe the location of the entity in the game world. The **loc** attribute gives the id of the reference relative to which this entities position is described, and the **pos** attribute specifies the coordinates relative to that reference entity. The ordering of attributes is arbitrary; the order given above is chosen for clarity of explanation.

2.4 Introducing Atlas Operations

The following Bach fragment represents an operation.

```
{
 objtype: "op",
 parents: ["move"],
 serialno: 12345,
 refno: 0,
 args: [{objtype: "obj",
         id: "pig_42",
         parents: ["pig"],
         loc: "world_0",
         pos: [0.5, 3.4, 9.7],
         velocity: [0.0, 1.0, 0.0]
        }
       ]
}
```

The basic format is the same as that of the entity example however in this case the first attribute **objtype** indicates that this is an operation object and the second indicates that it inherits from an object called move, which in this case is an operation definition. This means that this is a move operation, the effect of which is to change the location data of an entity. The move operation definition is one of a group of definitions referred to as actions. One of the most important attributes of any operation is the **args** list attribute. This can be considered analogous to the arguments of a procedure or method call, and usually contains one or more Atlas objects. In the case of a move operation, the first and usually only argument is a description of the entity to be moved, with at very least the id of that entity provided. In the example above, the reference entity **loc**, the coordinates relative to that entity **pos**, and the **velocity** of the entity are given. In addition, the **parents** attribute is included, to indicate that this entity inherits from the class called pig.

The result of this operation will be that the pig indicated is moved. In order for any other entities within the game to observe this movement, a sight operation indicating that nearby entities have seen something happen is sent to appropriate entities. The sight operation may look something like this in Bach.

```
{objtype: "op",
 parents: ["sight"],
 serialno: 12346,
 refno: 12345,
 args: [{objtype: "op",
         parents: ["move"],
         serialno: 12345,
         refno: 0,
         args: [{objtype: "obj",
                 id: "pig_42",
                 parents: ["pig"],
                 loc: "world_0",
                 pos: [0.5, 3.4, 9.7],
                 velocity: [0.0, 1.0, 0.0]
                }
               ]
        }
       ]
}
```

This operation inherits from the sight operation definition, which is one of a group of operation definitions known as perceptions. The attributes are very similar to those of the move operation, with some notable exceptions. A **refno** attribute is provided to indicate that this operation is dispatched with reference to the move operation. The first and only argument of this sight operation is the move operation given in the previous example and this is how the information about the movement is conveyed to the other entities. Sight operations with entity data as their arguments are requested by issuing a look operation, the definition for which is one of a group of perceive operation definitions.

A look operation typically provides a minimal entity description as its argument, to specify the id of the entity it is looking at. In order to bootstrap examination of the entities in the world, a look operation with no argument is issued. The result is a sight of the top level entity in the world, which can be examined to determine the ids of the rest of the visible entities in the world.

Two remaining operation attributes are **to** and **from**, used to specify the source and destination of operations. The use of the attributes internally to systems which use Atlas varies from system to system. In general **from** is almost always used when sending an operation from a client to a server, to indicate what server side object the client is using to invoke this operation. **to** is almost always used when sending an operation from the server to a client to indicate which server side entity the operation was received by. For example, when the client sends a look operation, it sets the operation **from** the character which it is currently controlling and it will expect the resulting sight operation to be **to** that character.

2.5 Atlas Implementations

The development work and experimentation for most of Atlas was done using a python implementation of the basic functionality, as python lends itself very well to the task. In order to provide an implementation for production use, the Atlas-C++ implementation was written. Atlas-C++ features vastly better performance than the python prototype, and has been designed for good portability and reliability. This implementation was used as the basis for the java implementation Atlas-Java.

2.6 Atlas Applications

The game server Cyphesis has been in development longer than Atlas and was the first application to use Atlas both internally and for communication. It was developed by Aloril in python as a prototype game engine, with an AI and Alife engine. It was then engineered in C++ as a game server for performance reasons. It supports access control using Atlas account entities via login, logout and create operations. The game world functionality provided by the core software supports perception of the world through look, sight and touch operations, as well as talk and sound operations for communication. World modification is supported via move, set, create and delete operations. In addition, scripts can be added to extend the functionality using essentially arbitrary operation types.

Cyphesis was used as the platform for the Acorn demo game. The game required players to create a character and take on the role of a pig farmer. The pig farmer would buy piglets from a trader, fatten them up using food discovered in the game environment and sell the pigs for slaughter at a profit, while avoiding obstacles placed in their way.

Scripts were written to define the nature of the entity classes required for the game and operation definitions were created for the necessary interactions. Information about these entity classes and operation types was integrated into the client software where necessary.

Acorn gameplay relied almost entirely on features and operations supported by the server core to implement its gameplay. Players controlled pigs by the client sending touch operations which drive the pigs through the forest. Predators relied on sight operations to identify pigs and move operations to chase them. Trading of pigs in the market place relied on talk and sound operations to communicate between the players and the market traders.

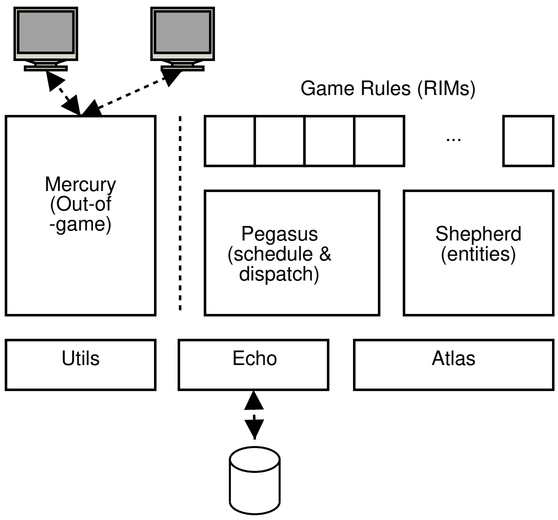
The game, developed in the most part by Karsten-Olaf Laux, Uta Szymanek and Alistair Riddoch was successfully release at LinuxTag 2001, and has been WorldForge's flagship product for the past year.

Part II
Development

3 STAGE

3.1 Why Another Server?

STAGE is intended to provide several orders-of-magnitude increase in the size of world that can be supported, and it's complexity. In particular, many issues which are solved in Cyphesis using brute force approaches are (or will be) given a much more thorough treatment in STAGE. Most notably, a full rigid body physics model is being incorporated, which alone offers the potential for a



far richer game play experience. The STAGE design also anticipates wide-scale distribution of the server across commodity hardware, though this will not be implemented in the first versions.

3.2 Overview

The initial design of STAGE was undertaken by Bryce Harrington in late 2000, in parallel with the ongoing development of Acorn as discussed earlier. The roles of the basic functional units have evolved somewhat, but in general remain fairly consistent with the original design.

The critical components of STAGE are laid out here:

- Echo provides a database layer that underpins both the in-game and out-of-game systems

- Mercury provides a public network interface that clients connect to, implements account control and also the out-of-game (OOG) chat system.

- Shepherd controls game entities, which the entire world is built from. This includes in-memory storage, persistence and a caching layer that retrieves entities from the DB layer (or, in the future, a remote server).

- Orion provides a simulation of forces and velocities on entities

- Pegasus locates and schedules rule implementation modules (RIMs) that implement game logic, in response to Atlas commands (either from a client or some other source)

In a distributed model, one OOG server and a cluster of game servers collaborate to simulate the world. The inter-server communications are also achieved via Atlas.

3.3 Observers and the Broadcast model

One of the primary functions of any simulation server is to distribute changes to the world to clients efficiently. This includes restricting the visibility of changes based on distance and perception, to avoid exposing extra information to the client (which we cannot trust).

In Cyphesis, changes to the world are simply sent to every client within a fixed, server determined radius; this model is obviously unsuitable as the number of clients and changes to the world increases. Also, it is desirable to enable entities not associated with a client to respond to events in the world.

STAGE calls visible world changes 'events', and any object interested in receiving them an 'observer'. Clients are observers, but other code can be too (for example, a door that opens in response to a **talk** event). To manage the transmission of events, each container in the world hierarchy tracks the observers it contains, and it's relation to other containers. For example, two rooms (each a container) may be joined by a doorway. When the door is open, events occurring in one room will be propagated through to the other. When the door is closed, visual events will be blocked (assuming the door is not made of glass) and audible events may be dampened. In an outdoor environment, the relations between containers are more arbitrary, but the basic principle holds. Thus the container model can be considered as a graph, with relations between nodes forming the edges.

It is possible (with some additional complexity) to make this model very sophisticated, for example giving sound events a direction, and modelling propagation through doorways, windows and solid materials accurately. While this level of simulation is some way off, the underlying model is general enough to support it, and the potential for increasing realism in the world is very high (for example, hearing the foot-steps of a character walking around in a room above another). Also, there is nothing in the design limited to audible or visual events : modelling psychic or magic events in the same way is intended.

3.4 Entity caching and re-location

Since STAGE is designed to work with tens or hundreds of thousands of entities, the management model becomes considerably more complex. The ability to support multiple game servers also complicates the situation. The basic solution is a caching system, which aims to keep only a small percentage of entities in memory at any given moment: typically those which are in motion, being modified, or so on. The ability to lazily update world state at load time allows background updates such as plants growing.

The lowest level of the cache is the database system, which can be any SQL database; presently, MySQL and PostgreSQL are used for testing, and SQLite (an in-process, file based DB) is used for development purposes. Communication with the database is handled via Echo, which manages the serialisation of entities into SQL data.

Echo is a passive layer that responds to requests from the entity caching layer built into Shepherd. This layer maintains a reference count for 'active' entities, and the primary in-memory cache of inactive but recently used entities. The caching layer also manages preload requests to normalise the cache's use of Echo, and ultimately the database. This functionality is currently not well developed but is expected to be important as the number of entities rises. Fortunately, there is a very good spatial locality of access which makes various preload schemes viable.

In a distributed model, each server has it's own local entity cache. Similar to a distributed memory model, a given cache, when fulfilling a request for a specific entity ID, can query other caches in the network in addition to the database layer. A location based 'affinity' scheme will almost certainly be necessary to ensure that spatially related entities tend to reside on the same game server. Unlike distributed memory systems, where the pattern of access cannot be determined in advance by the operating system, we have the advantage of being able to locate 'busy' areas of the world on a single server, and ensure that transitions occur in places that are seldom visited. Thus, server allocation can be placed so that a city or town never spans multiple servers.

Note that the inter-server protocol for exchanging entities is also Atlas; however, the actual objects being sent contain server-specific data in addition to the publicly visible properties the client sees.

3.5 Concurrency

The concurrency model for STAGE was not specific as part of the initial design, resulting in a rather fragmented approach. At present, pthreads are used in a limited fashion to provide non-blocking database I/O in a generic manner, and the entity cache layer contains a maintenance thread. The overall model is roughly as follows:

- the entire out-of-game system runs on a single thread, and takes care never to block
- the in-game system contains a main thread, and some number of worker threads which process game logic. The main thread should never block, but the worker threads can and will block frequently.

In addition, the entity cache has it's own thread for processing with entity requests. This is a convenience that allows use of the main thread as a worker thread for development purposes, and needs to be modified in the future.

In the longer term, there is an alternative model for dealing with worker threads, which is to turn them into worker *processes*. This models is akin to how CGIs are implemented in a web server, in it's simplest form. A number of possible designs exist, such as maintaining a persistent process for each rule implementation, to avoid process creation overhead every time a rule is used. The IPC between the game server and it's rule processes would initially be Atlas messages, with the option of using shared memory in the future, though this reintroduces many of the synchronisation issues we hope to avoid by not using threads.

Certainly, even the existing threading code has had a number of complex bugs, and the out-of-process model naturally gives rule developers a sand-box, whilst protecting the server core from them. At present, the trade-offs in isolation, data-copying overhead and so on have not been investigated enough to decide whether an in-process solution based on threads is better or worse than an out-of-process one.

3.6 Status

The current implementation of STAGE includes working implementations of all the points discussed above (albeit in several cases with interim code). It supports a very complete out-of-game environment, and a gradually improving, persistent in-game world. At present the range of in-game features is very limited (essential basic movement and speech), but this already uses the physics modelling that has been developed. As the core persistence and caching systems stabilise, development on game rules and logic will add more user-visible features at an increasing pace.

STAGE is currently developed by a core team of five developers, with various contributions from many more.