

Transactional Atlas
as used by WorldForge games
version 0.2 DRAFT

Anders Petersson

November 2003

Contents

| | |
|---|-----------|
| 1 States | 3 |
| 1.1 Connected | 3 |
| 1.1.1 Serverinfo | 3 |
| 1.1.2 Account information | 4 |
| 1.1.3 Login | 4 |
| 1.1.4 Create Account | 5 |
| 1.1.5 Changing information on an existing account | 6 |
| 1.2 Logged in | 6 |
| 1.2.1 Logout | 6 |
| 1.2.2 Out of game | 7 |
| 1.2.3 Transition to in-game | 7 |
| 1.3 In-Game | 7 |
| 1.3.1 Entity information | 7 |
| 1.4 How to find out where an op is aimed | 7 |
| 2 General | 8 |
| 2.1 Notes on atlas operations | 8 |
| 2.1.1 Required | 8 |
| 2.1.2 Standard | 8 |
| 2.1.3 Optional | 8 |
| 2.1.4 Unset | 8 |
| 2.1.5 Ad-hoc | 8 |
| 2.1.6 Unspecified | 9 |
| 2.2 Concurrent operations | 9 |
| 2.2.1 Serialno/Refno | 9 |
| 2.3 Multicontrol | 9 |
| 2.4 Atlas hirearchy discovery | 9 |
| 2.5 Errors | 9 |
| 2.6 Administrator accounts | 9 |
| 2.7 Media | 9 |
| 2.8 Caching | 9 |
| A References | 10 |
| A.1 Bach syntax | 10 |
| A.1.1 Basic Atlas | 10 |
| A.1.2 Atlas Attributes | 10 |
| A.1.3 Codec specifications | 10 |
| A.1.4 Acorn Whitepaper | 10 |

| | |
|--------------------------------------|-----------|
| B Changes | 11 |
| B.1 Major version 0, DRAFT | 11 |
| B.1.1 Version 0.2 | 11 |
| B.1.2 Version 0.1 | 11 |

Chapter 1

States

The following states are stacked, that is most ops used in the various states are still viable during the following states. The most obvious exceptions being the one-shot ops used to change states such as login, the state discovering ops are always good such as the oog look discovery.

1.1 Connected

This is the state you enter directly after atlas codec negotiation, a client doesn't expect any ops to be sent and the server doesn't send any until the client takes some action.

1.1.1 Serverinfo

The serverinfo op is used to discover information about the server commonly used by metaserver clients.

This op is structured as an anonymous get, that is a get without from nor to being set.

```
{
  parents: [get],
  to, - Unset, see section 2.1.4.
  from - Unset, see section 2.1.4.
}
```

The returned op is an info op with a server argument.

```
{
  parents: ['info'],
  args: [{
    parents: ['server'],
    name: (string), - Standard, see section 2.1.2.
    ruleset: (string), - Standard, see section 2.1.2.
    clients: (int), - Standard, see section 2.1.2.
    uptime: (float), -- in seconds - Standard, see section 2.1.2.
    server: (string), - Optional, see section 2.1.3.
  ]
}
```

```

    version: (string), - Optional, see section 2.1.3.
    builddate: (string) - Optional, see section 2.1.3.
  }]
}
```

ruleset The rules the server is using (such as acorn or mason). This can be used by the client to provide specialized behaviour more appropriate for the game, and of course tell the user what game it is running.

uptime The number of seconds the server has been running.

server The name of the server hosting the game (currently cyphesis or indri).

version The version of the *server* running the game.

1.1.2 Account information

```

{
  parents: ['account'],
  id: (string), - Required, see section 2.1.1.
  character_types: (string list), - Ad-hoc, see section 2.1.5.
  characters: (string list), - Standard, see section 2.1.2.
  name: (string) - Standard, see section 2.1.2.
}
```

id used as from when account is doing anything

character_types the types of characters which can be created

characters the list of characters currently active in the account, this is the basis of what ids can be used to move in-game

name the name displayed when the account is taking a visual action, mainly out of game chat

- The password is probably considered a set-only operation by most servers, especially in the case of hashed passwords.
- The id cannot be changed this could introduce nasty incoherence in the clients and the server.

1.1.3 Login

The login op is a concurrent (do *not* rely on serialno/refno, look in 2.2.1) state transitional op, it is only allowed in the connected state, the op can be reversed by sending a logout (see 1.2.1) op when logged in.

The structure of a login op is as follows, currently there is no way of discovering password hashing supported by the server.

```

{
  parents: ['login'],
  args: [{
    parents: ['account'], - Optional, see section 2.1.3.
  }]
```

```

        username: (string), - Required, see section 2.1.1.
        password: (string) - Required, see section 2.1.1.
    }]
}

```

On successful login an info op with the account is sent, possibly with a type derived from account (commonly player or admin, see 2.4 for information on type discovery). It has lots of useful information used in the later states.

```

{
  parents: ['info'],
  args: [{
    (attr): (value) - See 1.1.2.
  }]
}

```

If the login was unsuccessful an error op (see 2.5 for details) will be sent with a reason and the info op as an arg.

1.1.4 Create Account

The login op is a concurrent (do *not* rely on serialno/refno, look in 2.2.1) state transitional op, it is only allowed in the connected state, the op can be reversed by sending a logout (see 1.2.1) op when logged in.

The create-account is the way of logging in if no account has been previously created. Be careful as the server can at will override any arguments you send, rely on the response and not on what you sent. (Clients can with great success use the same callback to handle both the response of both login ops and create account ops.)

The op is structured as a create op with an account argument. But note that id will most likely be overridden (many servers will probably keep an username, id equivalence). Also character_types and characters will be ignored (unless you connect to a very strange server).

```

{
  parents: ['create'],
  args: [{
    (attr): (value) - See 1.1.2.
  }]
}

```

The response on a successful create account is an info op with the account argument (exactly the same you would expect from a successful login).

```

{
  parents: ['info'], args: [{
    (attr): (value) - See 1.1.2.
  }]
}

```

In case of an error (could be anything ranging from username collision to requiring some kind of authorization) you will be sent an error op see 2.5 for details.

1.1.5 Changing information on an existing account

It is possible to change information on an existing account. It is done by sending set ops to the account id with the arguments to set as the argument.

```
{
  parents: ['set'],
  to: (id), - Required, see section 2.1.1.
  args: [{
    (attr): (value) - See 1.1.2.  }]
}
```

Usually you can set any attributes not derived by the server state (ie not id, characters, character_types and such).

1.2 Logged in

This is the state the connection enters when a successful login or create account operation has been performed. It can be reversed by a logout (1.2.1) operation.

1.2.1 Logout

The logout operation is used when a client wants to login using another account or wants to end the connection in a clean way (dropping the connection will always work of course).

If the logout op has no argument then the account to be logged out is the value of the from attribute. If it does have arguments then the id attribute of the arguments is used. *However* servers which do not support the logging out of other accounts might not bother to check the arguments and use the from directly (cyphesis-c++ most notably).

```
{
  parents: ['logout'], - Required, see section 2.1.1.
  from: (string), - Required, see section 2.1.1.
  args: [{
    id: (string), - Optional, see section 2.1.3.
  }]
}
```

1.2.2 Out of game

1.2.3 Transition to in-game

Character creation

Character activation

1.3 In-Game

1.3.1 Entity information

1.4 How to find out where an op is aimed

Chapter 2

General

2.1 Notes on atlas operations

Unless explicitly noted otherwise parents are always required to be sent.

2.1.1 Required

That an attribute is *required* means that it must always be sent across the network to enable the client and/or server to decode the object correctly.

2.1.2 Standard

A *standard* attribute must always be available for retrieval from the server, this is to ensure that the client can do some sane assumptions in their internal data model.

2.1.3 Optional

An *optional* attribute may or may not be present in the servers data model, the client can safely ignore these but will then lose some functionality should the server support it.

2.1.4 Unset

The *unset* attributes must not even be sent across the wire for the operation to work correctly, they are used to distinguish when the target is unknown (such as discovering the out of game lobby or the in-game world).

2.1.5 Ad-hoc

Ad-hoc attributes are temporary measures to provide functionality which should really be provided another way, be prepared for them to disappear (character_types is the most notable one, it will probably use the type-tree later on).

2.1.6 Unspecified

The *unspecified* attributes are nothing but functionality provided by clients which can be useful to know but they are not defined and you shouldn't depend on their presence, their type nor that they actually mean what you think they do (one example could be that `silence-py` specified a `style` attribute to chat operations which determined how they were to be displayed, this degraded gracefully on all clients who didn't know about it).

2.2 Concurrent operations

2.2.1 Serialno/Refno

Don't depend on them!

2.3 Multicontrol

Multiple logins and characters using the same connection are allowed (if supported by the server) but require some care in the client library for successful decoding. More specifically `from/to` become exorbitantly important.

2.4 Atlas hierarchy discovery

2.5 Errors

```
{
  parents: ['error'],
  args: [{
    message: (string) - Required, see section 2.1.1.
    -- should we perhaps have some machine readable error?
  },
  {
    (the operation this error is a response to) - Required, see section 2.1.1.
  }
  ]
}
```

2.6 Administrator accounts

2.7 Media

2.8 Caching

Appendix A

References

A.1 Bach syntax

Expressing an Atlas object in the common Bach encoding form:

```
{
  parents: ["info"],
  objtype:"op",
  arg: {
    name: "Joe",
    pos: [4.5, 6.7, 2.3],
    meaning_of_life: 42
  }
}
```

In above example “parents” is list attribute containing one string value “info”.

Arg is mapping attribute containing 3 attributes:

- “name” is string attribute with value “Joe”.
- “pos” is list attribute with 3 float values [4.5, 6.7, 2.3].
- “meaning_of_life” is an integer attribute with value 42.

Above example also “specifies” Bach encoding for most purposes. List is enclosed with [], mapping is enclosed with {} and string is enclosed with “”.

A.1.1 Basic Atlas

A.1.2 Atlas Attributes

A.1.3 Codec specifications

A.1.4 Acorn Whitepaper

Appendix B

Changes

B.1 Major version 0, DRAFT

B.1.1 Version 0.2

- New chapter: Changes
- New section: Bach syntax (Stolen from http://purple.worldforge.org/aloril/atlas/simple_core.html)
- Changed all atlas specifications to bach
- Changed all bach ops to use tabbing typewriter mode
- New section: Notes on atlas operations
- Added `\required`, `\standard`, `\optional`, `\unset`, `\ad hoc` and `\unspecified` specifications to atlas attributes
- Extended and corrected the error specification
- Cleaned up the account specification
- Extended the serverinfo specification
- Moved the sections beneath “Account information” up a level

B.1.2 Version 0.1

- Initial Draft