

Atlas as used by WorldForge games
version 0.3 DRAFT

Anders Petersson

November 2003

Contents

1	States	3
1.1	Introduction	3
1.2	Connected	3
1.2.1	Server notices	3
1.2.2	Serverinfo	4
1.2.3	Account information	4
1.2.4	Login	5
1.2.5	Create Account	5
1.2.6	Changing information on an existing account	6
1.3	Logged in	6
1.3.1	Logout	6
1.3.2	Out of game chat	7
1.3.3	Transition to in-game	11
1.4	In-Game	11
1.4.1	Entity information	11
1.5	How to find out where an op is aimed	11
2	General	12
2.1	Notes on atlas operations	12
2.1.1	Required	12
2.1.2	Standard	12
2.1.3	Optional	12
2.1.4	Unset	12
2.1.5	Ad-hoc	12
2.1.6	Unspecified	13
2.1.7	Read-Only	13
2.2	Concurrent operations	13
2.2.1	Serialno/Refno	13
2.3	Multicontrol	13
2.4	Atlas hirearchy discovery	13
2.5	Errors	13
2.6	Administrator accounts	14
2.7	Media	14
2.8	Caching	14

A	References	15
A.1	Bach syntax	15
A.1.1	Basic Atlas	15
A.1.2	Atlas Attributes	15
A.1.3	Codec specifications	15
A.1.4	Acorn Whitepaper	15
B	Changes	16
B.1	Major version 0, DRAFT	16
B.1.1	Version 0.3	16
B.1.2	Version 0.2	17
B.1.3	Version 0.1	17

Chapter 1

States

1.1 Introduction

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.

When an atlas connection is established it enters the state Connected, from here it is the clients responsibility to take whatever action it considers appropriate. Usually it is to login or create a new account. It then enters the Logged In state.

In the Logged In state a client commonly discovers the state of out of game communication, begins using the same and eventually moves In-Game by either activating an existing character or by creating a new character.

The In Game state is where all gaming activities take place and usually begins by discovering the characters surroundings and it's own status.

1.2 Connected

This is the state you enter directly after atlas codec negotiation. The only operations not specifically requested are server notices.

1.2.1 Server notices

Warning: This has not yet been reviewed and is likely to change!

Server notices are info operations containing a notice from the server, automated or otherwise. They can be identified by having unset "to" and "from" attributes and a "notice" argument.

```
{
  parents: ['info'],
  to, - Unset, see section 2.1.4.from, - Unset, see section 2.1.4.args: [{
    parents: ['notice'],
    description: (message), - Required, see section 2.1.1.
  }]
}
```

1.2.2 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.

1.2.3 Account information

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

1.2.4 Login

The login op is a concurrent (see section 2.2.1) state transitional op, it is only allowed in the connected state, the op can be reversed by sending a logout (see section 1.3.1) 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 operation with an argument inheriting from account is sent (commonly player or admin, see section 2.4 for information on type discovery). It has lots of useful information used in the later states.

```
{
  parents: ['info'],
  args: [{
    (attr): (value) - See section 1.2.3.
  }]
}
```

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

1.2.5 Create Account

The login op is a concurrent (see section 2.2.1) state transitional op, it is only allowed in the connected state, the op can be reversed by sending a logout (see section 1.3.1) 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.

```
{
  parents: ['create'],
  args: [{
    (attr): (value) - See 1.2.3.
  }]
}
```

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 1.2.3.
  }]
}
```

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

1.2.6 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 1.2.3.
  }]
}
```

The response is either an error operation (see section 2.5) or a sight of the set operation.

1.3 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 (1.3.1) operation.

1.3.1 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.
  }]
}
```

1.3.2 Out of game chat

Out of game is a code-name for all communication (in the social sense, not the technical) which is not happening in the game, the main reason for this is to not ruin the game experience for anyone. This is intentionally not as powerful as in-game.

Neither clients nor servers are required to support out of game chat.

Out of game is structured as a tree with one root room (node) which contain further rooms (nodes) and a list of people contained in each room. This is not entirely compatible with in game since an account can be present in any number of room.

Room information

```
{
  parents: ['room'],
  id: (string), - Required, see section 2.1.1.
  people: (string list), - Standard, see section 2.1.2.- Read-Only, see section 2.1.7.
  name: (string) - Standard, see section 2.1.2.
}
```

Discovering out of game chat

Out of game is structured as a tree with exactly one root node with any number of subnodes (which in turn can contain subnodes etc).

To discover the root node an anonymous look from the account is made.

```
{
  parents: ['look'],
  from: (account id), - Required, see section 2.1.1.
  to - Unset, see section 2.1.4.
}
```

The response is either a sight of exactly one room or (if the server doesn't support out of game chat) an error operation (see section 2.5).

```

{
  parents: ['sight'],
  to: (account id), - Required, see section 2.1.1.
  args: [{
    (see section 1.3.2)
  }]
}

```

You can then issue further looks recursing on rooms to discover the rest of the out of game rooms.

Joining/departing a room

To enter a room you send a move operation with an extra mode argument set to *join* and to leave a room you set it to *part*.

```

{
  parents: ['move'],
  from: (account id), - Required, see section 2.1.1.
  args: [{
    mode: 'join' or 'part', - Required, see section 2.1.1.
    id: (account id), - Required, see section 2.1.1.
    loc: (room id) - Required, see section 2.1.1.
  }]
}

```

Ponder and comment:
Should “to” be required? And perhaps “id” and “loc” inferred if not present (they can differ in case of for example a kick)?

The response is either an error operation (see section 2.5) or an info with the room as an argument and the attribute “action” set to either join or part.

```

{
  parents: ['info'],
  to: (account id), - Required, see section 2.1.1.
  args: [{
    action: 'join' or 'part', - Required, see section 2.1.1.
    (see section 1.3.2)
  }]
}

```

Kicking accounts

To remove an account from a channel you simply send a part message as shown in the previous section coming from your own account but with a different id attribute in the argument. Not all accounts are allowed to do this.

Appearance and disappearance

When you are in a room you will be notified when an account enters and/or leaves that room this is done by sending appearance and disappearance operations.

Ponder and comment:
Do we have (should we have) anything equivalent when rooms are created/deleted?

```

{
  parents: [‘appearance’] or [‘disappearance’],
  to: (account id), - Required, see section 2.1.1.
  from: (room id), - Required, see section 2.1.1.
  args: [{
    id: (account id), - Required, see section 2.1.1.
  }]
}

```

Account information

To find out how to render each account’s messages a client will want to use the “name” attribute of the account. (Since accounts can be present in any number of channels at once clients will probably want some kind of to avoid unnecessary lookups, and otherwise they’d have to multiplex the information manually since updates will be sent on a per-account basis rather than a per-room basis.)

A look operation targeted at an account will reveal the information contained therein.¹

Talking and emoting

To talk in a channel you send a talk operation with the speech as an argument.

```

{
  parents: [‘talk’],
  to: (room id), - Required, see section 2.1.1.
  from: (account id), - Required, see section 2.1.1.
  args: [{
    say: (speech), - Required, see section 2.1.1.
  }]
}

```

The response is either an error operation (see section 2.5) or a sound with the sent talk operation as an argument (this is of course sent to all accounts present in the room). Note that the sound operation has its “to” set to the receiving accounts id and not the originating accounts id, use the contained talk operation to decode that.

```

{
  parents: [‘sound’],
  to: (account id), - Required, see section 2.1.1.
  from: (room id), - Required, see section 2.1.1.
  args: [{
    (see above)
  }]
}

```

¹My choice of words suggest me that I should get some sleep.

Emoting

To express yourself non-verbally (emoting) you send an imaginary operation with a description of the action to the channel.

```
{
  parents: ['imaginary'],
  to: (room id), - Required, see section 2.1.1.
  from: (account id), - Required, see section 2.1.1.
  args: [{
    description: (speech), - Required, see section 2.1.1.
  }]
}
```

The response is either an error operation (see section 2.5) or a sight operation with the sent imaginary operation as an argument. The same rules as for talk operations apply.

```
{
  parents: ['sight'],
  to: (account id), - Required, see section 2.1.1.
  from: (room id), - Required, see section 2.1.1.
  args: [{
    (see above)
  }]
}
```

Account to account communication

Any operation from an account to another account will be forwarded unmodified, there are no mandated rules concerning this but if you want other clients to understand you will want to simply send what you would expect to receive. Wrap a talk in a sound operation and an imaginary in a sight operation.

The only thing which the server guarantees is that the “to” and “from” attributes can be trusted and that the “refno” is set to the “serialno” of the originating operation.

You will also want to send yourself this information since the server won't send you a copy.

When decoding this you will spot that the “from” attribute isn't a room but an account. This is what you use to determine that it's a private message rather than public chatter.

1.3.3 Transition to in-game

Character creation

Character activation

1.4 In-Game

1.4.1 Entity information

1.5 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. Also the id attribute is always a read-only attribute.

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).

2.1.7 Read-Only

Read-Only attributes are sent by the server but can never be set (neither in a `create` op nor a `set` op) if someone tries to do that it will be ignored. One attribute which is always read-only is the `id` attribute.

2.2 Concurrent operations

2.2.1 Serialno/Refno

The `serialno/refno` scheme is a system which can help clients dealing with concurrent operations. The guarantee provided is that the `refno` of an op coming from the server will be the `serialno` of the op that was sent to the server. Since the ops can come from different clients there is no inherent guarantee that every `refno` will be unique and the safe way to make use of them is to check both that the structure and the `refno` of truly concurrent ops (ie when you get exactly one answer and you know the possible types of responses).

This also implies that servers must respect the `serialnos` (and set them as corresponding `refnos`) sent by clients, they are however not obliged to set `serialnos` of outgoing operations.

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

Unless otherwise specified all changes are made by me (Demitar). The (SOA *) fields tells whose input made me perform a specific change.

B.1 Major version 0, DRAFT

B.1.1 Version 0.3

- New section: States / Introduction (SOA zzorn)
- Fixed that account creation always returns an arg inheriting from account (SOA James)
- New section: Notes on atlas operations / Read-Only
- Removed reference to setting id/characters/character_types on account creation (SOA James)
- Added response specification when changing account information
- Added \readonly specification to atlas attributes
- Fixed serialno/refno information — it is dependable information (SOA James, alriddoch)
- Changed title from “Transactional Atlas\\as used by WorldForge games” to “Atlas as used by WorldForge games” (SOA sfb, bear, alriddoch)
- Added some explanation of the changes format to the beginning of chapter Changes
- Removed section “OOG” in favour of “out of game chat”
- Filled out section “out of game chat”
- New section: Server notices
- Updated section Connected to account for Server notices

B.1.2 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`, `\adhoc` 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.1.3 Version 0.1

- Initial Draft