

Server Operations Guide

BigWorld Technology 2.0. Released 2010.

Software designed and built in Australia by BigWorld.

**Level 2, Wentworth Park Grandstand, Wattle St
Glebe NSW 2037, Australia
www.bigworldtech.com**

Copyright © 1999-2010 BigWorld Pty Ltd. All rights reserved.

This document is proprietary commercial in confidence and access is restricted to authorised users. This document is protected by copyright laws of Australia, other countries and international treaties. Unauthorised use, reproduction or distribution of this document, or any portion of this document, may result in the imposition of civil and criminal penalties as provided by law.

Table of Contents

1. Introduction	7
2. Server Configuration with bw.xml	9
2.1. The entry parentFile	9
2.2. User dependent configuration	10
2.3. Command-Line Options	10
2.4. General Configuration Options	10
2.5. Billing System Configuration Options	17
2.6. Network Configuration Options	18
2.7. Network Address Translation Configuration Options	18
2.8. Load Balancing Configuration Options	19
2.8.1. Entities Cardinality Configuration Options	19
2.9. BaseApp Configuration Options	20
2.9.1. Secondary Database Configuration Options	24
2.9.2. Packet Log Configuration Options	25
2.9.3. ID Configuration Options	26
2.9.4. Client Upstream Limits	26
2.10. BaseAppMgr Configuration Options	27
2.11. Bots Configuration Options	28
2.12. CellApp Configuration Options	30
2.12.1. Noise Configuration Options	35
2.12.2. ID Configuration Options	36
2.12.3. CellApp Profiles Configuration Options	36
2.13. CellAppMgr Configuration Options	38
2.14. DBMgr Configuration Options	41
2.14.1. Data Consolidation Options	43
2.14.2. XML Database Options	44
2.15. LoginApp Configuration Options	44
2.16. Reviver Configuration Options	47
2.16.1. Reviver's BaseAppMgr Configuration Options	48
2.16.2. Reviver's CellAppMgr Configuration Options	48
2.16.3. Reviver's DBMgr Configuration Options	48
2.16.4. Reviver's LoginApp Configuration Options	48
3. Cluster Administration Tools	51
3.1. WebConsole	51
3.1.1. Modules	51
3.1.2. Installation and Configuration	52
3.1.3. User Administration	52
3.1.4. Production Mode vs Development Mode	54
3.1.5. Customising	55
3.2. Message Logger	55
3.2.1. Configuration	56
3.2.2. File Format	56
3.2.3. Command Line Utilities	57
3.2.4. Automatic Log Archiving	58
3.2.5. Production Scalability	58
3.3. StatLogger	59
3.3.1. Requirements	59
3.3.2. Output	60
3.3.3. Data Collection and Aggregation	60
3.3.4. Configuration	60
3.3.5. Database	65
3.4. Server Command-Line Utilities	67
3.4.1. Control Cluster	67
3.4.2. MessageLogger Related Utilities	69

3.5. Space Viewer	70
3.5.1. Selecting Spaces to View	71
3.5.2. Viewing Spaces	71
3.5.3. Customising Entity and Display Colours	72
3.5.4. Running Space Viewer Remotely	73
4. Fault Tolerance	75
4.1. CellApp Fault Tolerance	75
4.2. BaseApp Fault Tolerance	75
4.3. Fault Tolerance with Reviver	76
4.3.1. Specifying Components to Support	76
4.3.2. Recommended Reviver Layout	77
4.3.3. Command-Line Options	77
5. Backups and Disaster Recovery	79
5.1. Disaster Recovery	79
5.2. Database Snapshot Tool	79
5.2.1. Operational Behaviour	79
5.2.2. Usage	80
5.2.3. Requirements	80
5.2.4. Partitioning	82
5.2.5. Configuration	83
5.2.6. Restoring From a Snapshot	84
5.3. Data Consolidation Tool	85
5.3.1. Skipping Data Consolidation	86
5.3.2. Ignoring SQLite Errors	86
6. Controlled Startup and Shutdown	89
6.1. Server-wide Shutdown	89
6.2. Individual Application Retirement	89
6.2.1. BaseApp Retirement	89
6.2.2. CellApp Retirement	90
6.2.3. Retirement via WebConsole	90
6.2.4. Retirement via <code>control_cluster.py</code>	90
7. Stress Testing with Bots	91
7.1. The Login Process	91
7.2. Python Interface	92
7.2.1. Python Controller (<code>bot_op.py</code>)	92
7.2.2. Methods and Attributes	93
7.3. Controlling Movement	96
7.3.1. NodeProperties Section	98
7.4. Extending Bots	99
7.4.1. Creating New Movement Controllers	99
7.5. Miscellaneous Bots Issues	99
7.5.1. Running out of File Descriptors	99
8. Security	101
8.1. Security of the Server	101
8.2. Server Ports	102
8.3. Blocking Ports and Related Security Considerations	103
9. BigWorld Server Across Multiple Machines	105
9.1. How To Start	105
9.1.1. WebConsole	105
9.1.2. Auto Configuration Via <code>control_cluster.py</code>	105
9.1.3. Hard-Coded Scripts	105
9.1.4. Manual Start	105
9.2. How To Stop	105
9.3. How To Monitor	105
9.4. LoginApp and Scalability	106
10. Multiple BigWorld Servers in a Single LAN	107

10.1. Keeping Processes Separate	107
10.2. Centralised Cluster Monitoring	107
10.3. Auto-Detection of LoginApps	107
11. MySQL Support	109
11.1. Compiling DBMgr with MySQL Support	109
11.2. Update bw.xml To Use MySQL	109
11.3. Synchronise Database With Entity Definitions	109
11.4. Enabling Secondary Databases	110
11.5. Privileges	110
11.6. The ClearAutoLoad tool	111
12. RPM	113
12.1. Directory Structures and Files	113
12.2. How to Generate Binary RPM Packages	113
12.3. Customising RPM Packages	114
12.4. Setting up a Yum Repository	115
12.5. Install, Upgrade and Uninstall using Yum Command	116
12.5.1. Install and Upgrade using a RPM Package Directly	116
12.5.2. Install and Upgrade using Yum Repository	117
12.5.3. Remove an Installed Package	117
12.6. How to Obtain Version Number of an Installed Package	117
13. First Aid After a Crash	119
13.1. Change your coredump output directory, if necessary	119
13.2. Determine the first process(es) that crashed	119
13.2.1. BigWorld logs	120
13.2.2. Chronologically listing core files	120
13.3. Generate a stack trace of the process that crashed	120
13.3.1. Troubleshooting	122
13.4. Retrieve relevant log information	123
13.4.1. Generating a log summary from a core file	123
13.4.2. Archiving all logs	123
13.5. Back up the crash information	124
13.6. Notify BigWorld Support of the crash	124
13.6.1. Uploading large files	124
13.6.2. Providing BigWorld access to your server cluster	125
14. Common Log Messages	127
14.1. Warnings	127
14.2. Errors	128
15. Clock	131
15.1. BigWorld Timing Methods	131
15.2. Linux Clock Source	131
16. Machine Groups and Categories	133
16.1. Introduction	133
16.2. Configuration	133
16.3. User Defined Categories	134

Chapter 1. Introduction

BigWorld Technology is BigWorld's middleware for implementing Massively Multiplayer Online Games. This document is a guide to performing operations with the server software. It is not intended for game designers or game logic implementers, but rather for 'machine room' or 'cluster control' operators and administrators.

It is assumed that the server has been installed according to the instructions in the document *Server Installation Guide*. An understanding of the basic BigWorld processes is also assumed. For more details on these processes, see the document *Server Overview*'s chapters *Design Introduction* and *Server Components*.

Note

For details on BigWorld terminology, see the document *Glossary of Terms*.

Chapter 2. Server Configuration with bw.xml

The single most important configuration file on the server is `<res>/server/bw.xml`, where `<res>` is the resource tree used by the server (usually specified in `~/ .bwmachined.conf`).

All sever processes read this file. It contains many parameters, all of which are described in this chapter. The default values are appropriate for many different games, and care should be taken when changing them, since it might affect performance.

On the description of the parameters, please note the following:

- Boolean parameters should be specified as true or false.
- Where a tag is specified as `tag1/tag2`, the second tag is specified inside the scope of the first one. For example, the tag `dbMgr/allowEmptyDigest` is specified as:

```
<dbMgr>
  <allowEmptyDigest> true </allowEmptyDigest>
</dbMgr>
```

2.1. The entry parentFile

The entry `parentFile` in the configuration file specifies the next file in the chain of files where BigWorld should look for an entry for a configuration option.

To assign a default value to a configuration option, BigWorld follows the steps below:

- Searches for an entry for the configuration option in file `<res>/server/bw.xml`.
- If the file does not contain an entry for the configuration option, then the chain of configuration files is inspected, until the entry is found.
- If the entry is not specified in any of these files, a hard-coded default is used.

All default values for a production environment are stored in file `bigworld/res/server/production_defaults.xml`. The file `bigworld/res/server/development_defaults.xml` specialises this with default values for a development environment. Typically, the entry `parentFile` in file `<res>/server/bw.xml` is set to either `server/production_defaults.xml` or `server/development_defaults.xml`, and only non-default options are stored in your `server/bw.xml`.

The example below shows the configuration option `maxCPUOffload` in section `balance` having its default value overridden:



Overriding default values for configuration options

2.2. User dependent configuration

A file that is user dependent can be used instead of bw.xml. This is useful to allow multiple users to run from the same resource tree. In a production environment, for example, you may run the resources using a test user before using the production user.

If a file with the name server/bw_<username>.xml exists, this is used as the start of the server configuration chain instead of server/bw.xml.

Typically, the parentFile section in this file would refer to server/bw.xml and only options specific to the user, such as dbMgr/databaseName would be in this file.

2.3. Command-Line Options

The configuration options specified in file <res>/server/bw.xml can also be overridden via command-line arguments.

To override a default value, add arguments in the format +optionName value.

The example shows the baseApp section's configuration option pythonPort having its default value changed to 40001, and the option archivePeriod changed to 0:

```
baseapp +baseApp/pythonPort 40001 +baseApp/archivePeriod 0
```

Values changed via the command line are not sent to components started via BWMachined. This includes using WebConsole, control_cluster.py, and components started by a Reviver process.

2.4. General Configuration Options

The list below describes the general configuration options:

- **bitsPerSecondToClient (Integer)**

Desired default bandwidth from server to the client. To calculate the number of bytes to be sent in each packet, the formula below is used (where UDP_OVERHEAD is 28 bytes):

$$\text{packetSize} = (\text{bitsPerSecondToClient} / 8 / \text{gameUpdateHertz}) - \text{UDP_OVERHEAD}$$

- **debugConfigOptions (Integer)**

Level of logging information generated when processing configuration parameters in file `<res>/server/ bw.xml`.

The possible values are described in the list below:

- **0**

No log is generated.

- **1**

A log message is generated for each configuration option read.

- **2**

A verbose message is generated for each configuration option read.

- **desiredBaseApps (Integer)**

Number of BaseApps that need to be running for the server to start.

- **desiredCellApps (Integer)**

Number of CellApps that need to be running for the server to start.

- **externalInterface (String)**

Network adapter interface to use for external communication, if not explicitly set by the server component.

In a production environment, BaseApps are recommended to have two Ethernet adapters: one adapter connected to the Internet, and a separate one connected to the internal LAN.

During development, there is no problem with using the same interface.

Accepted formats are:

- **Adapter name** - Examples: `eth0`, `eth1`

- **IP[/netmask]** - Examples: `10.5.2.1`, `10.0.0.0/8`, `192.168.5.0/24`

- **Domain name** - Examples: `intern0.cluster/24`, `extern5.cluster/24`

This value can be overridden by a tag with same name in the following sections:

- `baseApp` - For more details, see “BaseApp Configuration Options” on page 20 .

- `loginApp` - For more details, see “LoginApp Configuration Options” on page 44 .

- **externalLatencyMax (Float)**

Maximum number of seconds by which packets sent from the server process will be artificially delayed. Each packet will be randomly delayed between this value and `externalLatencyMin`.

This value can be overridden by a tag with the same name in the following sections:

- `baseApp` - For more details, see “BaseApp Configuration Options” on page 20 .

- `loginApp` - For more details, see “LoginApp Configuration Options” on page 44 .

This feature is useful for testing during development.

See also `externalLossRatio`, `xref_externalLatencyMin`

- **externalLatencyMin (Float)**

Minimum number of seconds by which packets sent from the server process will be artificially delayed. Each packet will be randomly delayed between this value and `externalLatencyMax`.

This value can be overridden by a tag with the same name in the following sections:

- `baseApp` - For more details, see “BaseApp Configuration Options” on page 20 .
- `loginApp` - For more details, see “LoginApp Configuration Options” on page 44 .

This feature is useful for testing during development.

See also `xref_externalLossRatio`, `xref_externalLatencyMax`

- **externalLossRatio (Float)**

Proportion of outgoing packets that will be dropped by this processes external nub to simulate loss on the external network. This is a value between 0.0 and 1.0 indicating what proportion of packets will be dropped.

This value can be overridden by a tag with the same name in the following sections:

- `baseApp` - For more details, see “BaseApp Configuration Options” on page 20 .
- `loginApp` - For more details, see “LoginApp Configuration Options” on page 44 .

This feature is useful for testing during development.

See also `xref_externalLatencyMin`, `xref_externalLatencyMax`

- **gameUpdateHertz (Integer)**

Number of times per second that the server should send an update to the clients. This corresponds to the game tick frequency.

- **hasDevelopmentAssertions (Boolean)**

Flag indicating whether server should be aggressive in its use of assertions.

This option should be set to true during development, and to false when running a production server.

For example, if this option is set to true, then a corrupted packet sent from a client can cause an assertion on the server, while if this is set to false, then only an error message is generated and the server component continues to run.

- **internalInterface (String)**

This tag is deprecated, and its use is not recommended. For details, see the document Server Overview’s section Server Components → “BWMachined” → “BWMachined Interface Discovery”.

Network adapter interface to use for internal communication, if not explicitly set by the server component.

Accepted formats are:

- **Adapter name** - Examples: `eth0`, `eth1`
- **IP[/netmask]** - Examples: `10.5.2.1`, `10.0.0.0/8`, `192.168.5.0/24`

- **Domain name** - Examples: `intern0.clust/24`, `extern5.clust/24`

This value can be overridden by a tag with the same name in the following sections:

- `baseApp` - For details, see “BaseApp Configuration Options” on page 20 .
- `baseAppMgr` - For details, see “BaseAppMgr Configuration Options” on page 27 .
- `cellApp` - For details, see “CellApp Configuration Options” on page 30 .
- `cellAppMgr` - For details, see “CellAppMgr Configuration Options” on page 38 .
- `dbMgr` - For details, see “DBMgr Configuration Options” on page 41 .
- `loginApp` - For details, see “LoginApp Configuration Options” on page 44 .

- **`internalLatencyMax` (Float)**

Maximum number of seconds by which packets sent from the application's internal nub will be delayed.

For more details, see `internalLatencyMin`.

See also `internalLossRatio`.

- **`internalLatencyMin` (Float)**

Minimum number of seconds by which packets sent from the application's internal nub will be delayed.

Each packet will be randomly delayed between this value and `internalLatencyMax`.

This feature is useful for testing during development.

This value can be overridden by a tag with the same name in the following sections:

- `baseApp` - For details, see “BaseApp Configuration Options” on page 20 .
- `baseAppMgr` - For details, see “BaseAppMgr Configuration Options” on page 27 .
- `cellApp` - For details, see “CellApp Configuration Options” on page 30 .
- `cellAppMgr` - For details, see “CellApp Configuration Options” on page 30 .
- `dbMgr` - For details, see “DBMgr Configuration Options” on page 41 .
- `loginApp` - For details, see “LoginApp Configuration Options” on page 44 .
- `reviver` - For details, see “Reviver Configuration Options” on page 47 .

See also options `internalLatencyMax` and `internalLossRatio`.

- **`internalLossRatio` (Float)**

Proportion of packets on this application's internal nub that will be dropped to simulate loss on the internal network. This is a ratio between 0.0 and 1.0.

This feature is useful for testing during development.

This value can be overridden by a tag with the same name in the following sections:

- `baseApp` - For details, see “BaseApp Configuration Options” on page 20 .

- `baseAppMgr` - For details, see “BaseAppMgr Configuration Options” on page 27 .
- `cellApp` - For details, see “CellApp Configuration Options” on page 30 .
- `cellAppMgr` - For details, see “CellApp Configuration Options” on page 30 .
- `dbMgr` - For details, see “DBMgr Configuration Options” on page 41 .
- `loginApp` - For details, see “LoginApp Configuration Options” on page 44 .
- `reviver` - For details, see “Reviver Configuration Options” on page 47 .

See also options `internalLatencyMax` and `internalLatencyMin`.

- **loggerID (String)**

The ID used by the process when registering with MessageLoggers. If this ID does not match a MessageLogger's filter, the process will not log to that MessageLogger. (For details on MessageLogger, see “Message Logger” on page 55).

Multiple BigWorld servers can share the same logger process. If this behaviour is not desired, then you can use a unique loggerID per server instance - this will cause MessageLogger to filter out all messages that do not match the loggerID it has been told to monitor.

- **logSpamPatterns (List of Strings)**

A list of log message prefixes can be specified which will be suppressed on a per-second basis if the number sent to MessageLogger exceeds a certain threshold. Note that this is not intended a mechanism to sweep error messages “under the carpet”; it is designed to reduce the network load that can be generated by log traffic, which tests have indicated can be in excess of actual game traffic in some situations if suppression is disabled.

This option can be overridden by a tag with the same name in any app section. Note that the overriding does not merge the suppression lists, it simply replaces the global list with the one defined at the app level.

Additionally, the list of suppression patterns can be modified at runtime using the `logger/addSpamSuppressionPattern` and `logger/delSpamSuppressionPattern` watchers.

Please see `bigworld/res/server/production_defaults.xml` for an example of a suppression list.

- **logSpamThreshold (Integer)**

The number of a particular log message that can be sent to MessageLogger in a single second before suppression will take place. Note that only messages matching one of the `<logSpamPatterns>` (see above) will be suppressed.

- **monitoringInterface (String)**

Network adapter interface to use for non-game communications, such as logging output, telnet sessions, and watcher requests. For example: `eth0`.

See the BaseApp's configuration option `externalInterface` for accepted formats.

- **networkCompression/external (String)**

The type of compress to use on large some external messages sent over the network. The supported options are NONE and ZIP. For zip compression, a compression level can be set between 1 and 9 with 1 being the fastest and 9 being the best compression.

```
<networkCompression>
  <internal> ZIP <level> 3 </level> </internal>
  <external> NONE </external>
</networkCompression>
```

This setting can be overwritten entity type via NetworkCompression/external section of the entity type's .def file.

- **networkCompression/internal (String)**

The type of compress to use on some large internal messages sent over the network.

This setting can be overwritten entity type via NetworkCompression/internal section of the entity type's .def file.

See networkCompression/external for more information.

- **numStartupRetries (Integer)**

Number of times that CellApps and BaseApps will try to locate other components when starting up.

Each attempt is one second apart, so this value roughly indicates the number of seconds that these two components can be started before the other 'global' server components have started.

- **outputFilterThreshold (Integer)**

Value used to filter the messages that are printed and sent to the logger.

All messages are tagged with an integer value. If the message number is greater than or equal to the filter value, then the message is allowed (the bigger the value, the more messages are filtered out).

For example, a threshold of 2 allows only INFO messages and higher (TRACE and DEBUG messages are filtered out).

The possible values and their message thresholds are described in the list below:

- 0 - MESSAGE_PRIORITY_TRACE
- 1 - MESSAGE_PRIORITY_DEBUG
- 2 - MESSAGE_PRIORITY_INFO
- 3 - MESSAGE_PRIORITY_NOTICE
- 4 - MESSAGE_PRIORITY_WARNING
- 5 - MESSAGE_PRIORITY_ERROR
- 6 - MESSAGE_PRIORITY_CRITICAL
- 7 - MESSAGE_PRIORITY HACK
- 8 - MESSAGE_PRIORITY_SCRIPT

- **personality (String)**

Name of the personality module for the server.

This module should contain things such as methods to be called back from the server (for example, when the server is ready). The personality module is usually named after your game.

If not specified, the module named `BWPersonality` is used.

- **production (Boolean)**

If set to `true`, enables the server processes to run in a production mode which makes a best attempt at emitting `ERROR` messages when encountering configuration settings that are considered detrimental for a production environment. In rare cases this may also prevent server processes from starting if the configuration options are seen to be completely unrealistic for a production environment.

Currently this is a global configuration option and cannot be set per server application type.

- **shutdownServerOnBadState (Boolean)**

Flag indicating whether server should be shut down when in an unrecoverable state.

Currently, the following scenarios are handled:

- All CellApps are dead.
- An attempt to restore Base entities after a BaseApp crash fails. This could happen if:
 - All BaseApps are dead.
 - Two BaseApps die in quick succession. This can cause the loss of base entities that were on the first BaseApp to crash and being backed up on the second BaseApp to crash.

- **shutdownServerOnBaseAppDeath (Boolean)**

If set to `true`, the entire server will be shut down if a single BaseApp dies. Normally, the fault tolerance system would allow the server to continue running.

- **shutdownServerOnCellAppDeath (Boolean)**

If set to `true`, the entire server will be shut down if a single CellApp dies. Normally, the fault tolerance system would allow the server to continue running.

- **shouldUseChecksums (Boolean)**

If set to `true`, then all packets sent between server components will be checksummed to verify their correctness. This is in addition to the UDP checksum automatically provided by the Linux kernel and protects against packet corruption by buggy network drivers. If a corrupted packet is detected by Mercury (meaning that it has somehow passed the UDP checksum), you will see an error message like:

```
ERROR Packet::validateChecksum: Packet (flags 178, size 1459) failed checksum
(wanted 3dc56738, got 9fe7000a)
```

If after running servers for long enough and not seeing this error message you feel confident that the UDP checksum is reliable enough on your hardware, you can disable this option for a small performance improvement. The checksum is very simple and fast to calculate so this is likely to have only a small impact on performance.

- **shuttingDownDelay (Float)**

Number of seconds that the server should wait before a requested controlled shutdown is actually performed.

- **tickStatsPeriod (Float)**

Number of seconds between ticking statistics that keep a moving average. There can be a lot of statistics that need to be ticked. If too much time is spent maintaining these statistics, consider increasing this value. The time taken ticking statistics is measured by the `tickStats` profile. For example, see `profiles/summaries/tickStats` watcher value.

- **`timeSyncPeriod` (Float)**

Number of seconds between each synchronisation of game time between `CellAppMgr` and other applications.

- **`useDefaultSpace` (Boolean)**

Flag indicating whether to automatically create an initial space when the server starts.

This option is ignored if spaces are loaded from the database during start-up.

2.5. Billing System Configuration Options

The configuration options specified in this section relate to user authentication and billing systems.

- **`billingSystem/authenticateViaBaseEntity` (Boolean)**

This option is used to indicate that user authentication and billing system integration will be done via base entity script.

If true, the usual user authentication done by `DBMgr` is bypassed and a base entity of type `entityTypeForUnknownUsers` is always created. The `DBMgr` attempts to load the entity with the username. If this does not exist, a new entity is created.

This new entity is not initially stored in the database. It is up to the base entity script to call `Base.writeToDB()` if it wants the entity to persist.

See also option `entityTypeForUnknownUsers`.

- **`billingSystem/entityTypeForUnknownUsers` (String)**

This specifies the type of the entity to create when `shouldAcceptUnknownUsers` or `authenticateViaBaseEntity` are used.

See also options `shouldAcceptUnknownUsers` and `shouldRememberUnknownUsers`.

- **`isPasswordHashed` (Boolean)**

Flag indicating whether the password stored in the database's `bigworldLogOnMapping` table is hashed. Hashing is recommended for security reasons. With hashing, it is very difficult to determine the original password from the database. For more details, see "Default Authentication via MySQL".

- **`billingSystem/shouldAcceptUnknownUsers` (Boolean)**

If true, a user can log in with an unknown login name and a new entity of type specified in `entityTypeForUnknownUsers` will be created.

See also options `entityTypeForUnknownUsers` and `shouldRememberUnknownUsers`.

- **`billingSystem/shouldRememberUnknownUsers` (Boolean)**

If true, the entity created for an unknown user will be stored in the database and an account record will be stored for this entity. This option is only used when `shouldAcceptUnknownUsers` is true.

See also option `shouldAcceptUnknownUsers`.

2.6. Network Configuration Options

The configuration options specified in this section relate to network communication and the behaviour of various aspects of communication channels.

The options specified in the following list are specifically related to the behaviour of channels when packets start overflowing. This can occur when the send window fills up and buffering of packets is required in order to handle packet resends.

The maximum packet options defines a per channel type threshold to assist in preventing channels from using indefinite amounts of memory while buffering overflow packets.

- **maxChannelOverflow/isAssert (Boolean)**

Specifies whether the offending channel should raise a program assertion, effectively terminating the process, when the maximum number of overflow packets has been reached. This only applies to internal channels and does not apply for channels to client applications.

- **maxChannelOverflow/external (Integer)**

Number of packets to allow to overflow on an *external* channel before raising an ERROR message, or ASSERT'ing if `<isAssert>` has been set to true. A value of 0 disables any log messages and assertions from occurring.

- **maxChannelOverflow/internal (Integer)**

Number of packets to allow to overflow on an *internal* channel before raising an ERROR message, or ASSERT'ing if `<isAssert>` has been set to true. A value of 0 disables any log messages and assertions from occurring.

- **maxChannelOverflow/indexed (Integer)**

Number of packets to allow to overflow on an *indexed* channel before raising an ERROR message, or ASSERT'ing if `<isAssert>` has been set to true. A value of 0 disables any log messages and assertions from occurring.

An indexed channel is a channel that is used for communicating directly between the cell and base parts of an entity.

2.7. Network Address Translation Configuration Options

The configuration options specified in this section relate to running the server behind a NAT (Network Address Translator). This may be useful during development to allow a BigWorld server to run behind a NAT'ing firewall.

These configuration options are specified in file `<res>/server/bw.xml` under the section `<networkAddressTranslation>`, and are described below:

- **externalAddress (String)**

Address to be returned to client if he is outside the server cluster LAN.

If the client is outside the server cluster LAN (the option `localNetMask` is used to determine that), then LoginApp will return the IP address set in this option, instead of the address that the BaseApp thinks it is on.

This option is intended for use only during development, when the machines that the BaseApps are running on do not have real IP addresses (*i.e.*, they are behind a NAT'ing firewall), but you still want clients to log in from the Internet.

See also option `localNetMask`.

- **localNetMask (String)**

Mask to be used against the client's IP address in order to determine whether he is inside the server cluster LAN.

The net mask is an IP address followed by the number of bits to match. For example, `10.0.0.0/8` would match any IP starting with 10 (*i.e.*, `10.*.*.*`). The default is `0.0.0.0/0`, in which case no redirecting will be done.

If it is determined that the client is not on the server cluster LAN, the LoginApp will return the IP address set in option `externalAddress`.

See also option `externalAddress`.

2.8. Load Balancing Configuration Options

The load balancing configuration options are specified in file `<res>/server/bw.xml` under the section `<balance>`, and are described below:

- **maxCPUOffload (Float)**

Estimated amount of CPU processing that can be offloaded from a cell to another in one tick of load balancing.

A larger value should result in faster changes to the server's load balancing. This value is a fraction of 100% CPU usage, and its range is from 0.0 through 1.0, but is likely to always be less than 0.1.

- **minEntityOffload (Integer)**

Using the option `maxCPUOffload`, the load balancing algorithm may decide that less than one entity needs to be offloaded. Since this can get rounded down to zero, the load balancing can get restricted when small adjustments need to be made.

This option set a minimum value for this, so if it's set to 1 (the default), then the load balancing will have the capacity to make small adjustments.

- **minMovement (Float)**

Minimum number of metres that a partition line should move when load balancing.

A non-zero value can be useful to avoid the load balancing getting stuck.

- **slowApproachFactor (Float)**

Aggressiveness of the load balancing when cells are close to being balanced.

The value range is from 0 through 1.

A smaller value will result in a slower approach to being balanced. Too high a value may result in some instability when the space is close to being balanced.

2.8.1. Entities Cardinality Configuration Options

The configuration options for load balancing based on the number of entities are specified in file `<res>/server/bw.xml` under section `<balance>/<demo>`, and are described below:

- **enable (Boolean)**

Flag indicating whether the number of entities should be used to calculate load, rather than CPU load.

In normal situations, the server uses the CPU load on the CellApps to load balance. But sometimes it is desirable to use the number of entities per CellApp instead.

This may be useful, for example, when running multiple CellApps on a single machine for testing.

See also option `demo/numEntitiesPerCell`.

- **numEntitiesPerCell (Float)**

If option `demo/enable` is `true`, then this option is used for calculating a CellApp's load.

The load is calculated as `numEntities / numEntitiesPerCell`.

See also option `demo/enable`.

2.9. BaseApp Configuration Options

The BaseApp configuration options are specified in file `<res>/server/bw.xml` under the section `<baseApp>`, and are described below:

- **archivePeriod (Float)**

Period length in seconds where each entity is written to the database for the purpose of disaster recovery. Each Entity is guaranteed to have an archive less than $2 \times \text{archivePeriod}$ old. Setting it to zero switches off archiving.

A large value increases performance, but reduces the effectiveness of eventual disaster recovery. The opposite is true for a small value.

If secondary databases are disabled, this configuration option controls how often the entity is written to the primary database. In this case, this configuration option has a dramatic impact on the performance of the primary database when there are a large number of BaseApps. It is recommended to start with large values (a few minutes), and perform database testing and tuning before reducing it.

If secondary databases are enabled, this configuration option controls how often the entity is written to the secondary database. In this case, this configuration option can be set to a relatively small value (less than a minute) since it only impacts the BaseApp machine and the load is independent of number of BaseApps - unless the secondary database directory is on a shared network drive. For more details on secondary databases, see Server Programming Guide's chapter "Secondary Databases".

This option is also available for CellAppMgr.

- **backupPeriod (Float)**

Number of seconds between backups of each base entity to its backup BaseApp. This value is rounded to the nearest game tick.

As a first level of fault tolerance, base entities can be copied to a backup BaseApp (*i.e.*, backup to RAM), while cell entities are copied to their base entity. For more details on BaseApp and CellApp fault tolerance, see the document Server Programming Guide's chapter *Fault Tolerance*.

The value for this option is very dependant on the game. A small value means frequent backups, and consequently less lost data in case a BaseApp fails. But backups cost bandwidth and CPU on the BaseApp.

In general this period can be much smaller than the one specified in option `archivePeriod`.

Setting this to 0 disables backups.

See also option `backupPeriod` on “CellApp Configuration Options” on page 30 .

- **`backupTimeout` (Float)**

Number of seconds for a BaseApp to respond to the watchdog messages from its backup BaseApp before it is considered dead and the backup BaseApp takes its place.

- **`backUpUndefinedProperties` (Boolean)**

Flag indicating whether undefined properties should be backed up.

Properties of an entity are defined in the entity's definition file. However, it is possible to define additional properties for this entity in the base script of this entity. For example, an additional property can be defined by initialising it in the constructor of this entity class. These additional properties are referred to as undefined properties.

If this option is set to true, undefined properties will be backed up and an error will be emitted for each of the properties that cannot be pickled. If this option is set to false, undefined properties will not be backed up. Default value is true.

- **`clientOverflowLimit` (Integer)**

If the send window for the channel to the client grows larger than this many packets, the client is disconnected.

Generally, it is better to rely on `inactivityTimeout` to detect an unresponsive client and so this option should be set to greater than `inactivityTimeout * gameUpdateHertz`.

See also option `inactivityTimeout` on “BaseApp Configuration Options” on page 20

- **`createBaseElsewhereThreshold` (Float)**

Threshold of local BaseApp load below which calls to `BigWorld.createBaseAnywhere` cause the new base entity to be created locally.

- **`externalInterface`**

For details, see “General Configuration Options” on page 10 .

- **`externalLatencyMax`**

For details, see “General Configuration Options” on page 10 .

- **`externalLatencyMin`**

For details, see “General Configuration Options” on page 10 .

- **`externalLossRatio`**

For details, see “General Configuration Options” on page 10 .

- **`externalPorts/port` (Integer)**

Port that may be used for the BaseApp's external socket - the BaseApp can have more than one `externalPorts/port` definition.

This option is useful when BaseApp is running behind a firewall and NAT port mappings need to be set up.

If the BaseApps are run behind a firewall, then each BaseApp expected to run on a single machine should have an `externalPort` definition.

If this option is not specified, or all specified externalPorts are taken then the BaseApp will bind to any available port on the external interface.

```
<baseApp>
  <externalPorts>
    <port> 40013 </port>
    <port> 40014 </port>
    <port> 40015 </port>
  </externalPorts>
</baseApp>
```

- **inactivityTimeout (Float)**

Number of seconds that a proxy will proceed without communication from the client before it considers the connection to be dead.

See also option `clientOverflowLimit` on “BaseApp Configuration Options” on page 20

- **internalInterface**

For details, see “General Configuration Options” on page 10 .

This tag is deprecated, and its use is not recommended. For details, see the document Server Overview’s section Server Components → “BWMachined” → “BWMachined Interface Discovery”.

- **internalLatencyMax**

For details, see “General Configuration Options” on page 10 .

- **internalLatencyMin**

For details, see “General Configuration Options” on page 10 .

- **internalLossRatio**

For details, see “General Configuration Options” on page 10 .

- **loadSmoothingBias (Float)**

Value to smooth the load changes on a component by when calculating new load.

The BigWorld server uses the load on a component to perform its load balancing. Unfiltered, the load can change too quickly to be useful. The option `loadSmoothingBias` is used to smooth out this value.

The filtered value is calculated at each game tick as follows:

```
newSmoothedLoad = (1 - loadSmoothingBias) * oldSmoothedLoad + loadSmoothingBias
* load
```

This option is also available for CellApps and CellAppMgr.

- **minClientInactivityResendDelay (Float)**

Minimum number of seconds that the BaseApp waits before resending unacknowledged packets to the client. BigWorld also uses other metrics like the round-trip-time to the client to adjust the resend timeout period. However, BigWorld will not set the resend timeout period to less than the value specified in this configuration option.

The BaseApp will resend packets to the client due various reasons like lost packets, network jitter or client delays in sending packet acknowledgements. Increasing the minimum resend delay will generally reduce the number of packet resends. However, increasing this value will also increase the impact of a lost packet on the end user experience. The client will only process packets in order. Therefore, a lost packet will cause the client to buffer subsequent packets from the server and wait for the lost packet to arrive. If the resend timeout is long, the user will notice a temporary halting of updates from the server.

- **pythonPort (Integer)**

Port that the Python telnet session listener should listen on.

If set to zero, then a random port is chosen.

If the option is missing, then the port number will be set according to the formula:

`40,000 + BaseApp ID`

If the desired port is not available in any case, then a random one is used.

This option is also available for CellApps.

- **reservedTickFraction (Float)**

Percentage of tick time that should be remaining on current tick so the next one is considered to be pending.

This value is expressed as fraction. For example, setting it to 0.2 means that the next tick will be considered pending when there is still 20% of the current tick's time remaining.

Increasing this parameter will make the server more conservative in its CPU usage.

This affects how aggressive the method `BigWorld.fetchFromChunks` will be about yielding processing to the next tick.

Note: This should rarely be changed from the default value.

- **sendAuthToClient (Boolean)**

Flag indicating whether BaseApps must send authentication messages to clients (clients always send authentication to the server).

Use this feature to avoid hacking of clients, and prevent users from spoofing server messages to other clients.

Without this authentication, someone can send fake messages to clients, pretending to be the server (they will need the IP address of the client, and the port that the server is using, which can only be determined from the target client's data stream).

This option's value defaults to false, in order to save bandwidth.

- **sendWindowCallbackThreshold (Float)**

The fraction of an entity channel's send window that needs to be used before the `onWindowOverflow` callback is called on the associated Base entity.

- **shouldResolveMailBoxes (Boolean)**

Flag indicating whether a mailbox should be resolved to a Base entity, when possible. If a mailbox refers to a Base entity on the local BaseApp, the entity is used instead of the mailbox.

Although it is more efficient to have this option set, it is generally better to have it disabled. Having this enabled can lead to hard to find errors as behaviour changes depending on whether an entity happens to be local or not.

- **verboseExternalInterface (Boolean)**

Flag indicating whether to generate verbose log output related to external network traffic.

- **warnOnNoDef (Boolean)**

Flag indicating whether to generate a warning when Base entity properties are set that do not have a description in the entity's def file.

- **watcherValues (String)**

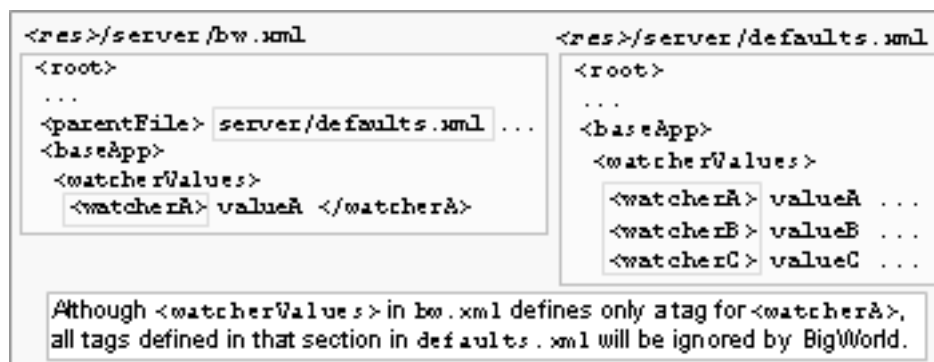
This is not an actual configuration option, but instead a sub-section inside the section `<baseApp>`, used to set arbitrary watcher values for the BaseApp at initialisation time.

This might be useful when there is a watcher value with no corresponding entry on file `<res>/server/bw.xml`.

For example, to set value of `logger/cppThresholds/baseapp.cpp` to 2:

```
<baseApp>
  <watcherValues>
    <debug>
      <baseapp.cpp> 2 </baseapp.cpp>
    ...
```

Do not use this feature if there is a parameter that can be set directly. Like all configuration options, this one is only evaluated once. It means that if there is an entry for `watcherValues` in `<res>/server/production_defaults.xml`, then all tags defined in it will be ignored if `<res>/server/bw.xml` also has an entry for `watcherValues` (even if different tags are specified in each one). This option is also available for CellApps.



File hierarchy

2.9.1. Secondary Database Configuration Options

The BaseApp secondary database configuration options are specified in file `<res>/server/bw.xml` under section `<baseApp>/<secondaryDB>`, and are described below:

- **enable (Boolean)**

Flag indicating whether to use secondary databases.

For more details about secondary databases, see Server Programming Guide's section "Secondary Databases".

- **maxCommitPeriod (Float)**

Maximum number of seconds between each commit. Higher values will result in better performance. Lower values will reduce the amount of data loss in case of a total system crash.

Explicit `BigWorld.writeToDB` calls will always result in a commit. This option only affects the automatic archiving of entities.

If set to zero or this option is empty, the value defaults to 5.

- **directory (String)**

Directory where the secondary database files are stored. Secondary databases are SQLite files.

The secondary database files are cleaned up when the system shuts down. However, these files should not be treated as normal temporary files since they are crucial to the data recovery process in case of a complete system crash.

If the first character of the path is a `/` character, the path is treated as an absolute path. Otherwise, the path is treated as relative to the first `<res>` path that contains the directory.

Note

It is not recommended to set the SQLite secondary database directory to reside on a network file-system. SQLite has specific requirements with regards to using file locking services from the file-system which may not be implemented properly for network file-systems.

It is recommended that each BaseApp machine has a dedicated space on a local file-system for storing secondary database files.

2.9.2. Packet Log Configuration Options

The BaseApp packet logging configuration options are specified in file `<res>/server/ bw.xml` under section `<baseApp>/<packetLog>`, and are described below:

- **addr (String)**

Client for which the packet's content should be logged.

The address is specified as dotted decimal format (e.g., 10.40.1.1).

If this option is empty, then all packets will be logged.

- **enable (Boolean)**

Flag indicating whether to write all packets to a local log file `proxy.log`.

This can be useful for debugging.

- **flushMode (Boolean)**

Flag indicating whether the log file should be flushed after each write.

This option is useful to ensure all log writes are captured if the BaseApp is crashing.

- **hexMode (Boolean)**

Flag indicating whether the logged packets' contents should be written in hexadecimal.

2.9.3. ID Configuration Options

The BaseApp ID configuration options are specified in file `<res>/server/bw.xml` under section `<baseApp>/<ids>`, and are described below:

- **criticallyLowSize (Integer)**

Minimum number of IDs in the BaseApp's available ID pool before the other limits are automatically adjusted.

The adjustment aims to help avoid this from occurring again.

- **desiredSize (Integer)**

Target number of IDs in the BaseApp's available ID pool when requesting IDs to the parent broker ID (in case it fell below `lowSize`), or returning IDs to it (in case it rose above `highLevel`) - for CellApps and BaseApps, the parent ID broker is the CellAppMgr, while for CellAppMgr it is DBMgr.

- **highSize (Integer)**

Maximum number of IDs in the BaseApp's available ID pool before IDs are returned to the parent ID broker - for CellApps and BaseApps, the parent ID broker is the CellAppMgr, while for CellAppMgr it is DBMgr.

ID recycling is currently disabled, so this value is actually never used.

- **lowSize (Integer)**

Minimum number of IDs that should be available in the BaseApp's available ID pool before a request is sent to the parent ID broker to restore it to the value specified in configuration option `desiredSize` - for CellApps and BaseApps, the parent ID broker is the CellAppMgr, while for CellAppMgr it is DBMgr.

2.9.4. Client Upstream Limits

The client upstream bandwidth can be limited so as to prevent denial-of-service attacks from malicious clients, or errant script code that cause messages from the client to be sent in high volume. This can lead to Mercury channels within the server to become heavily loaded.

To prevent this, limits can be specified on the count and size of incoming messages from clients to BaseApps. Once hard limits are reached on the count/size of incoming messages, messages are buffered and played back over time. Once the hard limits on buffering are reached, clients are disconnected.

The configuration parameters are specified in the file `<res>/server/bw.xml` in the `<baseApp>/<clientUpstreamLimits>` section. The available sub-elements are described in the following table.

- **warnMessagesPerSecond (Integer)**

This is a warning limit for the number of received messages from a client that are dispatched, measured in number of messages per second. When messages are received above this limit, a warning is emitted. No further warnings for this client of this type are emitted until the incoming message frequency drops below this limit.

- **maxMessagesPerSecond (Integer)**

This is a hard limit for the number of received messages from a client that are dispatched, measured in number of messages per second. When messages are received above this limit, those and subsequent messages are buffered for later playback until the buffer is empty. The message queue for all clients are checked every tick to replay any eligible messages.

- **warnBytesPerSecond (Integer)**

This is a warning limit for the throughput (in bytes) of received messages that are dispatched, measured in bytes per second. When this limit is exceeded, a warning is emitted. No further warnings for this client of this type are emitted until the incoming message data throughput drops below this limit.

- **maxBytesPerSecond (Integer)**

This is a hard limit for the size of data (in bytes) in received messages that are dispatched per second. When messages are received above this limit, those and subsequent messages are buffered for later playback until the buffer is empty. The message queue for all clients are checked every tick to replay any eligible messages.

- **warnMessagesBuffered (Integer)**

This is a warning limit for the number of received messages that may be buffered from the client. When this limit is exceeded, a warning is emitted. No further warnings for this client of this type are emitted until the number of queued messages drops below this threshold as a result of queue playback.

- **maxMessagesBuffered (Integer)**

This is a maximum limit for the number of received messages that may be buffered from the client. When this limit is exceeded, the client is disconnected and any messages that are buffered are discarded and not dispatched.

- **warnBytesBuffered (Integer)**

This is a warning limit for the total number of bytes that may be buffered for a client. When this limit is exceeded, a warning is emitted. No further warnings for this client of this type are emitted until the total size of buffered messages drops below this threshold as a result of queue playback.

- **maxBytesBuffered (Integer)**

This is a maximum limit for the total number of bytes that may be buffered for a client. When this limit is exceeded, the client is disconnected and any messages that are buffered are discarded and not dispatched.

2.10. BaseAppMgr Configuration Options

The BaseAppMgr configuration options are specified in file `<res>/server/bw.xml` under section `baseAppMgr`, and are described below:

- **baseAppOverloadLevel (Float)**

Minimum load level that all BaseApps should reach for the system to be considered in overload, and thus reject new logins.

Similar overload levels are specified for for any CellApp (by option `<cellAppOverloadLevel>` - for more details, see “CellAppMgr Configuration Options” on page 38), and for DBMgr (by option `<overloadLevel>` - for more details, see “DBMgr Configuration Options” on page 41).

- **baseAppTimeout (Float)**

Number of seconds for a BaseApp to respond before it is considered dead.

- **createBaseRatio (Float)**

The number of BaseApps that a destination BaseApp will have pointing to it. For example, if `createBaseRatio` is 4, the least loaded quarter of the machines will each have 4 BaseApps choosing them as the destination to create Base entities.

See also the option `updateCreateBaseInfoPeriod`.

- **hardKillDeadBaseApps (Boolean)**

Determines if a non-responsive BaseApp will be terminated with a SIGQUIT signal.

Non-responsive BaseApps must be terminated in order for its backup to take over its IP address and ID. BaseApp non-responsiveness is determined by its backup, so a BaseApp running without a backup will never be reported as being non-responsive (see also configuration option `baseApp/backupTimeout` in “BaseApp Configuration Options” on page 20).

When this option is set to false, no signal is sent to the non-responsive BaseApp.

Only use this option for debugging, *e.g.*, to attach a debugger to the hung process.

- **internalInterface**

For details, see “General Configuration Options” on page 10 .

This tag is deprecated, and its use is not recommended. For details, see the document Server Overview’s section Server Components → “BWMachined” → “BWMachined Interface Discovery”.

- **internalLatencyMax**

For details, see “General Configuration Options” on page 10 .

- **internalLatencyMin**

For details, see “General Configuration Options” on page 10 .

- **internalLossRatio**

For details, see “General Configuration Options” on page 10 .

- **overloadLogins (Integer)**

Maximum amount of logins that will be accepted during the overload tolerance period (see the `overloadTolerancePeriod` option) before rejecting any further logins.

- **overloadTolerancePeriod (Float)**

Number of seconds that logins will be accepted during a situation where the BaseApps are overloaded (see the `baseAppOverloadLevel` option). After this period of time, any further logins will be rejected.

- **updateCreateBaseInfoPeriod (Float)**

Time (in seconds) between updating all BaseApps with information to assist them in creating entities on other BaseApps when using `BigWorld.createBaseAnywhere()`.

See also the option `createBaseRatio`.

2.11. Bots Configuration Options

The Bots configuration options are specified in file `<res>/server/bw.xml` under the section `<bots>`, and are described below:

- **controllerData (String)**

Default data that the bot's controller will be created with (when bots are created, they get a controller associated with them to control their movement).

This may have different meanings for different controller types. For example, some controller types may interpret this as a filename to load data from.

- **controllerType (String)**

Type of the controller to be created with bot.

- **interface (String)**

Network adapter interface to use for game communication. For example: `eth0`. If not specified, the process will bind to all interfaces and report one of them. This should be appropriate in most situations.

See the BaseApp's configuration option `externalInterface` for accepted formats.

- **password (String)**

Password that the bots should use when logging in to the server.

- **port (Integer)**

Port on the server machine in which bots process will log to (only used if the option `<serverName>` is specified).

Ignored if bots automatically locates the LoginApp.

- **pythonPort (Integer)**

Port that the Python telnet session listener should listen to.

- **randomName (Boolean)**

Flag indicating whether a random suffix should be added to each bots name, in order to make them unique.

You should only set this option to false if you plan to use only a single bot. If you plan to use more than one bot, then you will need them to have different names - otherwise only the first one will be able to log in.

See also option `username`.

- **scripts (Boolean)**

Flag indicating whether the bots process should use Python scripting for received entities.

If set to false, then received entities are effectively ignored. Turning this option on has a significant performance penalty.

- **serverName (String)**

Name of the server machine that the bots process should log in to (*i.e.*, the machine running LoginApp).

If this option is empty, then the bots process will attempt to find an appropriate LoginApp on the local network.

- **shouldLog (Boolean)**

Flag indicating whether messages generated by the bots process should be sent to the central logger.

- **standinEntity (String)**

Default entity script to be used when a specific Entity type does not have its corresponding game script.

- **loginMD5Digest (String)**

MD5 digest string (in hex readable form) for server login.

- **userName (String)**

Username that bots should use when logging in to the server.

When randomName is true, this is the prefix before the randomly generated part of the name.

See also option randomName.

2.12. CellApp Configuration Options

The CellApp configuration options are specified in file `<res>/server/bw.xml` under the section `<cellApp>`, and are described below:

▪ aoiUpdateSchemes (Section)

This section contains a list of the different schemes that can be used when calculating the update rate of entities in an Area of Interest. The update rate of entities depends on their distance from the viewing client. The greater their update delta, the less frequent an entity's position is sent to the viewing client.

Each scheme contains the following values:

- name - The name of the scheme.
- minDelta - The update delta when the distance to the entity is 0 metres.
- maxDelta - The update delta when the distance to the entity is 500 metres.

The delta value for distances between this is calculated by a linear interpolation between these two points.

Setting minDelta to 1 and maxDelta to 101 means the delta for a given distance would be:

```
delta = 1 + 0.2 * distance
```

This means that entities at 20 metres (with a delta of 5) would be updated roughly 4.2 times more frequently than entities at metres (with a delta of 21).

The scheme can be set for an entity with the `Entity.aoiUpdateScheme` property or for an entity pair with the `Entity.setAoIUpdateScheme()` method.

```
<cellApp>
...
<aoiUpdateSchemes>

  <scheme>
    <name> default </name>
    <minDelta> 1 </minDelta>
    <maxDelta> 101 </maxDelta>
  </scheme>

  <!-- Update rate is not dependant on distance. This could be used on
  entities that are in sniper scope, for example. -->
  <scheme>
    <name> sniper </name>
    <minDelta> 10 </minDelta>
    <maxDelta> 10 </maxDelta>
  </scheme>

  <!-- Update twice as frequently as other entities at the same distance.
  Useful for large entities like dragons. -->
  <scheme>
```

```

        <name> largeEntity </name>
        <minDelta> 0.5 </minDelta>
        <maxDelta> 50.5 </maxDelta>
    </scheme>

</aoiUpdateSchemes>
</cellApp>

```

- **backupPeriod (Float)**

Number of seconds between backups of each cell entity to its base entity. This value is rounded to the nearest game tick.

As a first level of fault tolerance, cell entities are copied to their base entities, while base entities can be copied to their backup BaseApps. For more details on BaseApp and CellApp fault tolerance, see the document *Server Programming Guide's chapter Fault Tolerance*.

The value for this option is very dependant on the game. A small value means frequent backups, and consequently less lost data in case a CellApp fails. But backups cost bandwidth and CPU on the CellApp.

Setting this to 0 disables backups.

See also options `archivePeriod` and `backupPeriod` on “BaseApp Configuration Options” on page 20 .

- **checkOffloadsPeriod (Float)**

Number of seconds between offload checks.

This is a periodic check if entities need to be offloaded, or new ghosts created.

Other options affected by this setting are: `ghostingMaxPerCheck` and `offloadMaxPerCheck`.

- **chunkLoadingPeriod (Float)**

Number of seconds between checks on the progress of loading and unloading chunks.

Chunk loading occurs in a separate thread, but this check in the main thread queues up more chunks for the loading thread to load (or unload).

- **defaultAoIRadius (Float)**

The default AoI radius of new cell entities for proxy entities when they are created. See also the CellApp Python API's entry **Main → Cell → BigWorld → Classes → Entity → `setAoIRadius`**.

Note: If this is larger than the option `ghostDistance`, then the AoI radius of player entities is clamped to `ghostDistance` across cell boundaries.

- **enforceGhostDecorators (Boolean)**

Specifies whether to enforce the requirement of adding a decorator to methods that can safely be called on ghost entities.

When enabled, methods that have not been labelled as safe and are called on an entity that could be a ghost will generate a Python exception. To be considered safe, the method must either be described in the `.def` file or be decorated with `@bwdecorators.callableOnGhost`.

```

import bwdecorators
class Table( BigWorld.Entity ):

```

```
@bwdecorators.callableOnGhost
def getArea( self ):
    return self.width * self.height
```

- **entitySpamSize (Integer)**

Number of bytes that an entity in a player's AoI can add to an update packet to that player before a warning message is displayed.

This can be useful to identify entities that are causing a lot of downstream network traffic.

- **fastShutdown (Boolean)**

Specifies whether to avoid normal chunk unloading when the system is being shut down. This considerably speeds up the shutdown process.

- **ghostDistance (Float)**

The distance in metres outside the active cell boundaries where entities will be ghosted.

- **ghostingMaxPerCheck (Integer)**

Maximum number of ghost creation messages that can be sent per offload check.

The frequency of the offload check is determined by option `checkOffloadsPeriod`.

- **ghostUpdateHertz (Integer)**

Number of times per second channels to neighbouring CellApps are flushed.

Channels are created between neighbouring CellApps. Messages (such as ghost data) sent over these channels are not sent immediately, but are instead flushed periodically. This is done to avoid the high overhead of sending a packet.

If the value of this option is decreased, then there will be more lag for cross-cell communications.

Note: Bases always flush messages immediately.

- **internalInterface**

For details, see “General Configuration Options” on page 10 .

This tag is deprecated, and its use is not recommended. For details, see the document Server Overview’s section Server Components, “BWMachined”, “BWMachined Interface Discovery”.

- **internalLatencyMax**

For details, see “General Configuration Options” on page 10 .

- **internalLatencyMin**

For details, see “General Configuration Options” on page 10 .

- **internalLossRatio**

For details, see “General Configuration Options” on page 10 .

- **loadDominantTextureMaps (Boolean)**

Specifies whether to load the terrain's dominant texture maps. By default, this flag is set to `false`.

Loading the dominant texture maps enables using features such as material kinds returned by `BigWorld.collide`.

- **loadSmoothingBias (Float)**

Value to smooth the load changes on a component by when calculating new load.

The BigWorld server uses the load on a component to perform its load balancing. Unfiltered, the load can change too quickly to be useful. The option `loadSmoothingBias` is used to smooth out this value.

The filtered value is calculated at each game tick as follows:

$$\text{newSmoothedLoad} = (1 - \text{loadSmoothingBias}) * \text{oldSmoothedLoad} + \text{loadSmoothingBias} * \text{load}$$

This option is also available for `CellAppMgr` and `BaseApps`.

- **maxControllersAbsolute (Integer)**

Number of controllers that an entity must have before an exception is raised on attempts to create a new controller on it.

- **maxControllersExpected (Integer)**

Minimum number of controllers an entity must have before a warning is generated on attempts to create a new controller on it.

- **maxGhostsToDelete (Integer)**

Maximum number of ghosts to be deleted from other cells on every offload check (the frequency of this check is set via `checkOffloadsPeriod`).

This option is useful for adding antihysteresis and for smoothing the load caused by ghost deletion.

- **minGhostLifespan (Float)**

Minimum number of seconds for which a real entity will keep a ghost one.

This is useful for adding antihysteresis to the ghost creation and to the deletion process.

- **maxPhysicsNetworkJitter (Float)**

Maximum number of seconds to allow for when network jitter when considering movement cheating.

The movement of a player may vary slightly due to variations in network latency. This value sets the level of tolerance for this jitter.

- **navigatorUseGirthGrids (Boolean)**

Specifies if the waypoint search optimisation scheme should be used.

Girth grid is an optimisation scheme for waypoint search in a chunk. When this scheme is used, a chunk is divided up into a set of 12x12 grids according to the girth provided (you can have a list of 12x12 grid set for different girth sizes). Every grid square contains a subset of waypoints that overlap the covered area in a chunk.

During a (waypoint) search, only targeted grid squares (*i.e.*, subset of waypoints) are searched for the waypoint, instead of searching through the full set of waypoints. This scheme will generally improve the waypoint search performance.

- **obstacleTreeDepth (Integer)**

Depth of the obstacle tree to create.

Higher numbers increase the speed of collision detection but use more memory.

- **offloadHysteresis (Float)**

Number of metres that an entity has to be over a cell boundary before it is actually offloaded.

This helps avoid entities being offloaded back and forth between cells when standing very close to the border.

- **offloadMaxPerCheck (Integer)**

Maximum number of entities that can be offloaded from a cell during an offload check.

The frequency of the offload check is determined by option `checkOffloadsPeriod`.

- **pythonPort (Integer)**

Port that the Python telnet session listener should listen on.

If set to zero, then a random port is chosen.

If the option is missing, then the port number will be set according to the formula:

$$50,000 + \text{CellApp ID}$$

If the desired port is not available in any case, then a random one is used.

This option is also available for CellApps.

- **reservedTickFraction (Float)**

Percentage of tick time that should be remaining on current tick so the next one is considered to be pending.

This value is expressed as fraction. For example, setting it to 0.2 means that the next tick will be considered pending when there is still 20% of the current tick's time remaining.

Increasing this parameter will make the server more conservative in its CPU usage.

This affects how aggressive the method `BigWorld.fetchFromChunks` will be about yielding processing to the next tick.

Note: This should rarely be changed from the default value.

- **sendWindowCallbackThreshold (Float)**

The fraction of an entity channel's send window that needs to be used before the `onWindowOverflow` callback is called on the associated entity.

- **shouldResolveMailBoxes (Boolean)**

Flag indicating whether a mailbox should be resolved to a Cell entity, when possible. If a mailbox refers to a Cell entity on the local CellApp, the entity is used instead of the mailbox.

Although it is more efficient to have this option set, it is generally better to have it disabled. Having this enabled can lead to hard to find errors as behaviour changes depending on whether an entity happens to be local or not.

- **treatAllOtherEntitiesAsGhosts (Boolean)**

Puts the CellApp in a debugging mode in which a script running on it will see only its own entity as real - all other entities will be treated as ghosts.

Method calls, property access, and other functions will operate as if the other entities really are ghosts. This means that:

- Method calls will go via the network.
- Property access will be read-only, and limited to CELL_PUBLIC (or more public) properties¹.
- Many internal functions will not work, *e.g.*, adding a new Controller.

This is very useful for testing how your scripts work when dealing with ghost entities, especially if two interacting entities are nearby and would consequently rarely be ghosts.

- **watcherValues (String)**

This is not an actual configuration option, but instead a sub-section inside the section `cellApp`, used to set arbitrary watcher values for the CellApp at initialisation time.

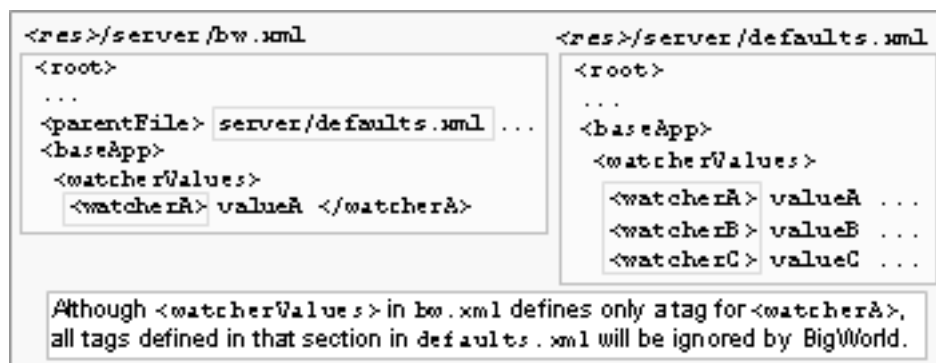
This might be useful when there is a watcher value with no corresponding entry on `<res>/server/bw.xml`.

For example, to set the watcher value `logger/cppThresholds/cellapp.cpp` to 2:

```
<cellApp>
  <watcherValues>
    <debug>
      <cellapp.cpp> 2 </cellapp.cpp>
    ...
  </watcherValues>
</cellApp>
```

Do not use this feature if there is a parameter that can be set directly.

Like all configuration options, this one is only evaluated once. It means that if there is an entry for `watcherValues` in `<res>/server/production_defaults.xml`, then all tags defined in it will be ignored if `<res>/server/bw.xml` also has an entry for `watcherValues` (even if different tags are specified in each one). This option is also available for BaseApps.



File hierarchy

2.12.1. Noise Configuration Options

The CellApp noise configuration options are specified in file `<res>/server/bw.xml` under section `<cellApp>/<noise>`, and are described below:

- **horizontalSpeed (Float)**

If an entity's horizontal speed exceeds this value (in metres per second), the entity makes a noise.

See `Entity.makeNoise` script method for more information. The event and info are 0 for noises generated this way.

- **standardRange (Float)**

Distance in metres through which a noise is propagated.

This value is multiplied by the level of a noise. For details, see the CellApp Python API's entry **Main → Cell → BigWorld → Classes → Entity → makeNoise**.

- **verticalSpeed (Float)**

If an entity's falling speed exceeds this value (in metres per second), the entity makes a noise.

This is done via script method `Entity.makeNoise` - for noises generated this way, the parameter event and info are set to 0. For details, see the CellApp Python API's entry **Main → Cell → BigWorld → Classes → Entity → makeNoise**.

2.12.2. ID Configuration Options

The CellApp ID configuration options are specified in file `<res>/server/bw.xml` under section `<cellApp>/<ids>`, and are described below:

- **criticallyLowSize (Integer)**

Minimum number of IDs in the CellApp's available ID pool before the other limits are automatically adjusted.

The adjustment aims to help avoid this from occurring again.

- **desiredSize (Integer)**

Target number of IDs in the CellApp's available ID pool when requesting IDs to the parent broker ID (in case it fell below `lowSize`), or returning IDs to it (in case it rose above `highSize`) - for CellApps and BaseApps, the parent ID broker is the CellAppMgr, and for CellAppMgr it is DBMgr.

- **highSize (Integer)**

Maximum number of IDs in the CellApp's available ID pool before IDs are returned to the parent ID broker - for CellApps and BaseApps, the parent ID broker is the CellAppMgr, and for CellAppMgr it is DBMgr.

ID recycling is currently disabled, so this value is actually never used.

- **lowSize (Integer)**

Minimum number of IDs that should be available in the CellApp's available ID pool before a request is sent to the parent ID broker to restore it to the value specified in configuration option `desiredSize` - for CellApps and BaseApps, the parent ID broker is the CellAppMgr, and for CellAppMgr it is DBMgr.

2.12.3. CellApp Profiles Configuration Options

The CellApp profiles configuration options are specified in file `<res>/server/bw.xml` under section `<cellApp>/<profiles>`. It contains sub-sections for enabling profiling of specific CellApp functionality. All the profiling options specified below can be modified after server startup on a per CellApp basis by using Watchers. The values are exposed in the CellApp watcher tree under `profilesConfigs/<option>`.

The list below details the available profiling sub-sections and options:

- **initGhost**

The **initGhost** sub-options define profiling information relating to the initialisation of ghost entities. Specifically the maximum time taken and network stream size required to initialise ghost entities from their reals after which a WARNING message will be generated. The initialisation primarily consists of the streaming of ghosted entity properties.

- **sizeWarningLevel (Integer)**

The size (in bytes) that an `initGhost()` method receives before a WARNING message is displayed.

- **timeWarningLevel (Float)**

The amount of time (in seconds) that an `initGhost()` method can take before a WARNING message is displayed. An example of the type of message generated is as follows.

```
WARNING      Profile initGhost/timeWarningLevel exceeded (Creature 23 of size
12338 bytes took 0.00477 seconds)
```

- **initReal**

Similar to the **initGhost** option, the **initReal** sub-options define profiling information relating to the initialisation of real entities. Specifically the maximum time taken and network stream size required to initialise real entities from their reals after which a WARNING message will be generated. The initialisation primarily consists of the streaming of entity properties.

- **sizeWarningLevel (Integer)**

The size (in bytes) that an `initReal()` method receives before a WARNING message is displayed. An example of the type of message generated is as follows.

```
WARNING      Profile initReal/sizeWarningLevel exceeded (Creature 13 of size
68765 bytes took 0.3726 seconds)
```

- **timeWarningLevel (Float)**

The amount of time (in seconds) that an `initReal()` method can take before a WARNING message is displayed.

- **onLoad**

The `onLoad()` operation is invoked when creating a real entity that had been offloaded from another CellApp.

- **sizeWarningLevel (Integer)**

The size (in bytes) that an `onLoad()` method receives before a WARNING message is displayed. An example of the type of message generated is as follows. The size is considered as the total size of the real entity and which is comprised of non-ghosted (i.e. CELL_PRIVATE) properties and other state information such as entities in the AoI, controller state etc.

- **timeWarningLevel (Float)**

The amount of time (in seconds) that an `onLoad()` method can take before a WARNING message is displayed.

- **backup/sizeWarningLevel (Integer)**

backup is the operation of performing a fixed point in time copy of the cell entity to the database (via the base). The size (in bytes) is the maximum persistent size of the entity (*i.e.* only persistent properties) after which a WARNING message is displayed.

2.13. CellAppMgr Configuration Options

The CellAppMgr configuration options are specified in file `<res>/server/bw.xml` under the section `<CellAppMgr>`, and are described below:

- **archivePeriod (Float)**

Number of seconds between database writes of space configuration and space data, for the purpose of disaster recovery.

Setting it to zero switches off database writes.

A large value increases performance, but reduces the effectiveness of eventual disaster recovery. The opposite is true for a small value.

Please note that unlike the BaseApp configuration option of the same name, this configuration option is not affected by the use of secondary databases. The data is always written to the primary database. It is recommended to start with large values (a few minutes), and perform database testing and tuning before reducing it.

This option is also available for BaseApps.

- **cellAppLoad/lowerBound (Float)**

Minimum average load that a CellApp must achieve before being retired.

This value is a fraction, and its range is 0.0 through 1.0.

Together with `cellAppLoadUpperBound`, this option should be tuned as to conserve CPU, while still allowing for temporary overloads (which depend on game code characteristics, such as AI usage).

This value must be less than the `cellAppLoadUpperBound`.

See also option `cellAppLoadUpperBound`.

- **cellAppLoad/safetyBound (Float)**

Limit that the load balancing can increase a cell's load to.

This value has to be greater than `cellAppLoadUpperBound`.

In some situations, such as adding a new cell, some cells may have their load temporarily increased.

- **cellAppLoad/safetyRatio (Float)**

Cell's average load safety ratio.

When the average load of the cells in a space is high, the limit that the load balancing can safely increase a cell's load to is calculated by multiplying this value by the average load. The real boundary is calculated as:

$$\max(\text{cellAppLoadSafetyBound}, \text{avgCellAppLoad} * \text{cellAppLoadSafetyRatio})$$

- **cellAppLoad/upperBound (Float)**

Minimum average load that a CellApp must achieve before a new cell is considered to be required.

This value is a fraction, and its range is 0.0 through 1.0.

Too low an upper bound will waste CPU, but too high an upper bound could potentially allow a CPU spike to kill a CellApp.

This value must be greater than the `cellAppLoadLowerBound`.

See also option `cellAppLoadLowerBound`.

- **cellAppLoad/warningLevel (Float)**

Minimum value that average load of CellApps must achieve (when there are no other CellApps available to share the load) before warning messages are sent to the log.

- **cellAppLoad/overloadLevel (Float)**

Minimum load level that any CellApp should reach for the system to be considered in overload, and thus reject new logins.

Similar overload levels are specified for all BaseApps (by option `<baseAppOverloadLevel>` - for details, see “BaseAppMgr Configuration Options” on page 27), and for DBMgr (by option `<overloadLevel>` - for details, see “DBMgr Configuration Options” on page 41).

- **cellAppTimeout (Float)**

Number of seconds for a CellApp to respond before it is considered dead.

- **hardKillDeadCellApps (Boolean)**

Determines if a non-responsive CellApp will be terminated with a SIGQUIT signal.

CellApp non-responsiveness is determined by `cellAppTimeout` option.

It is important to terminate the non-responsive CellApp, in order to prevent duplicate entities, since some of the entities on the non-responsive CellApp will be recreated on neighbouring CellApps².

When this option is set to false, no signal will be sent to the non-responsive CellApp. This option should only be used for debugging purposes, *e.g.*, to attach a debugger to the hung process.

- **internalInterface**

For details, see “General Configuration Options” on page 10 .

This tag is deprecated, and its use is not recommended. For details, see the document Server Overview’s section Server Components, “BWMachined”, “BWMachined Interface Discovery”.

- **internalLatencyMax**

For details, see “General Configuration Options” on page 10 .

- **internalLatencyMin**

For details, see “General Configuration Options” on page 10 .

- **internalLossRatio**

For details, see “General Configuration Options” on page 10 .

- **loadBalancePeriod (Float)**

Number of seconds between adjustments of the cell boundaries to improve the load balance on CellApps.

- **loadSmoothingBias (Float)**

Value to smooth the load changes on a component by when calculating new load.

The BigWorld server uses the load on a component to perform its load balancing. Unfiltered, the load can change too quickly to be useful. The option `loadSmoothingBias` is used to smooth out this value.

The filtered value is calculated at each game tick as follows:

$$\text{newSmoothedLoad} = (1 - \text{loadSmoothingBias}) * \text{oldSmoothedLoad} + \text{loadSmoothingBias} * \text{load}$$

The `CellAppMgr` further smooths the `CellApp` loads when it is informed of them.

This option is also available for `BaseApps` and `CellApps`.

- **maxLoadingCells (Integer)**

Maximum number of cells that the meta-load balancing will try to add to a space to help it initially load its geometry.

To disable this feature, set this value to 0.

See also `minLoadingArea`.

- **metaLoadBalanceScheme (Integer)**

The type of scheme that should be used to determine which Spaces should be a priority to meta load balanced. Current valid values are:

- **0 - Largest Space First**

The largest space contained in the most loaded group of connected `CellApps` will be chosen as the candidate for adding a new `Cell` into.

- **1 - Smallest Space First**

The smallest space contained in the most loaded group of connected `CellApps` will be chosen as the candidate for adding a new `Cell` into. This will enable the space to migrate off onto another `CellApp` thus reducing the overall load of the heavily loaded group.

- **metaLoadBalancePeriod (Float)**

Number of seconds between checking whether any cells should be added to or removed from any spaces to improve the load balance on `CellApps`.

- **minLoadingArea (Integer)**

Minimum average area (in square metres) that the cells in a space must have for it to be added by the meta-load balancing when attempting to initially load the geometry of a space.

If the average area is less than this value, then no more cells will be added to the meta-load balancing - this is meant to prevent too many cells being allocated to small spaces.

The number of cells that will be used for loading a space can be thought of as:

$$\text{min}(\text{maxLoadingCells}, \text{spaceGeometrySize} / \text{minLoadingArea})$$

See also `maxLoadingCells`.

- **overloadTolerancePeriod (Float)**

Number of seconds that logins will be accepted during a situation where the CellApps are overloaded (see the `cellAppOverloadLevel` option). After this period of time, any further logins will be rejected.

- **shouldLimitBalanceToChunks (Boolean)**

Determines if the loaded chunks of cells should be considered when load balancing.

This is enabled by default, so that load balancing will not cause a cell to cover an area that has not yet been loaded.

2.14. DBMgr Configuration Options

The DBMgr configuration options are specified in file `<res>/server/bw.xml` under the section `<dbMgr>`, and are described below:

- **allowEmptyDigest (Boolean)**

Flag indicating whether DBMgr should allow clients to log in with an empty MD5 digest.

When a client logs in, an MD5 digest of the entity definitions is sent to the server. This is to ensure that the client is using resources consistent with the server's.

This should be true if you are using either an example client, as found in `bigworld/src/examples`, or bots to log in, since those programs do not read the entity definition files (named `<res>/scripts/entity_defs/<entity>.def`).

- **databaseName (String)**

Only valid when the MySQL database layer is used.

Name of the underlying database to use.

- **dumpEntityDescription (Integer)**

Level of logging information generated when loading entity definition files (named `<res>/scripts/entity_defs/<entity>.def`).

The possible values are described in the list below:

- 0 - No log is generated.
- 1 - Brief details about the entity are generated.
- 2 - Full listing of entity is generated.

- **host (String)**

Only valid when the MySQL database layer is used.

Machine where the database is running.

The localhost identified can be used to refer to the current machine.

- **internalInterface (String)**

For details, see "General Configuration Options" on page 10.

This tag is deprecated, and its use is not recommended. For details, see the document Server Overview's section Server Components, "BWMachined", "BWMachined Interface Discovery".

- **internalLatencyMax (Float)**

For details, see “General Configuration Options” on page 10 .

- **internalLatencyMin (Float)**

For details, see “General Configuration Options” on page 10 .

- **internalLossRatio (Float)**

For details, see “General Configuration Options” on page 10 .

- **numConnections (Integer)**

Only valid when the MySQL database layer is used.

Number of connections to make to the underlying database.

This must be greater than or equal to 1. The default is 5.

- **overloadLevel (Float)**

Minimum load level that DBMgr should reach for the system to be considered in overload, and thus reject new logins.

Similar overload levels are specified for all BaseApp's (using `baseAppOverloadLevel` - for details, see “BaseAppMgr Configuration Options” on page 27), and for any CellApp (using `cellAppOverloadLevel` - for details, see “CellAppMgr Configuration Options” on page 38 .

- **overloadTolerancePeriod (Float)**

Number of seconds that logins will be accepted during a situation where the CellApps are overloaded (see the `overloadLevel` option). After this period of time, any further logins will be rejected.

- **password (String)**

Only valid when the MySQL database layer is used.

Password to be used when the DBMgr connects to the underlying database.

- **port (String)**

Only valid when the MySQL database layer is used.

Port to be used when the DBMgr connects to the underlying database. Set it to 0 to use the MySQL default port.

- **syncTablesToDefs (Boolean)**

Flag indicating whether changes to entity definition should be replicated to the MYSQL database.

The possible values are described below:

- **false**

No changes will be made to MySQL database structure.

- **true**

Make all necessary changes to the database structure to match the entity definitions. This includes deleting redundant columns and tables.

DBMgr will fail to start if the database structure does not exactly match the entity definitions.

This option is convenient during development as entity definitions change frequently. During production it is recommended to set this option to a safe mode (*i.e.*, false), then manually update the MySQL database structure by running the `sync_db` tool.

- **type (String)**

Type of database to use. Current options are: `xml` and `mysql`.

- **unicodeString/characterSet (String)**

This is the character set that the database should use to store `UNICODE_STRING` properties in the database. This is set to `utf8` by default. It should be rare that this should ever be changed. For more details on the effects of setting this option please refer to the Server Programming Guide, chapter *Character Sets and Encodings*, section “`UNICODE_STRING` storage”.

See the option `dbMgr/unicodeString/collation` for information on setting the collation type.

- **unicodeString/collation (String)**

This is the default collation that the database associates with `UNICODE_STRING` properties. This is set to `utf8_bin` by default. This can be set to `utf8_general_ci`, for example, if case insensitive lookups on identifier fields are required. With this setting, identifiers need to be unique using a case-insensitive comparison. For more details on the effects of setting this option please refer to the Server Programming Guide, chapter *Character Sets and Encodings*, section “Sorting search results”.

See `dbMgr/unicodeString/characterSet` for setting the character set.

- **username (String)**

Only valid when the MySQL database layer is used.

Username used when the DBMgr connects to the underlying database.

2.14.1. Data Consolidation Options

The options for data consolidation tool are located in the `<dbMgr>/<consolidation>` section of the server configuration file `<res>/server/bw.xml`

For more detail about the data consolidation process, see “Data Consolidation Tool” on page 85).

- **directory (String)**

Directory where the data consolidation tool puts temporary secondary database files copied from BaseApp machines. These files are automatically deleted when the data consolidation process completes.

If the first character of the path is a `/` character, the path is treated as an absolute path. Otherwise, the path is treated as relative to the first `<res>` path that contains the directory.

Note

It is not recommended to set the SQLite secondary database directory to reside on a network file-system. SQLite has specific requirements with regards to using file locking services from the file-system which may not be implemented properly for network file-systems.

It is recommended that each BaseApp machine has a dedicated space on a local file-system for storing secondary database files.

2.14.2. XML Database Options

The options specific to using XML as the database type are located in the `<dbMgr>/<xml>` section of the server configuration file `<res>/server/bw.xml`

Note

The XML database is written in the main thread. If saving takes a long time, this can block any processing by DBMgr. To avoid this, disable archiving and automatic saving.

- **archivePeriod (Float)**

Number of seconds between making an archive of the XML database. The newest archive is `scripts/db.xml.1` then `scripts/db.xml.2` etc.

To disable archiving, set this value to `0.0`.

- **numArchives (Integer)**

Number of archives to keep of the XML database.

- **savePeriod (Float)**

Number of seconds between saving the XML database to disk. The previous file is backed up as `scripts/db.xml.bak`.

To disable backing up, set this value to `0.0`.

2.15. LoginApp Configuration Options

The LoginApp configuration options are specified in file `<res>/server/bw.xml` under the section `<loginApp>`, and are described below:

- **allowLogin (Boolean)**

Flag indicating whether login attempts should be accepted.

This is useful when the server needs to be started without accepting logins for a while (*e.g.*, to have the server load chunks before allowing logins).

Logins can be enabled by setting `allowLogin` watcher on each LoginApp to `true` - for a simpler way to achieve this for all running LoginApps, see `control_cluster.py`'s `set` command³.

- **allowProbe (Boolean)**

Flag indicating whether login probes should be accepted.

It is recommended to have this flag set to `false` for servers running on the public Internet.

This setting can be changed at runtime by settings the `allowProbe` watcher.

See also option `logProbes`.

- **allowUnencryptedLogins (Boolean)**

Flag indicating whether a client sending a plaintext login request and/or a null session key should be allowed to connect.

This flag should only be used for testing purposes. Setting this flag to true on a production system is not recommended.

- **externalInterface**

For details, see “General Configuration Options” on page 10 .

- **externalLatencyMax**

For details, see “General Configuration Options” on page 10 .

- **externalLatencyMin**

For details, see “General Configuration Options” on page 10 .

- **externalLossRatio**

For details, see “General Configuration Options” on page 10 .

- **internalInterface**

For details, see “General Configuration Options” on page 10 .

This tag is deprecated, and its use is not recommended. For details, see the document Server Overview’s section Server Components, “BWMachined”, “BWMachined Interface Discovery”.

- **internalLatencyMax**

For details, see “General Configuration Options” on page 10 .

- **internalLatencyMin**

For details, see “General Configuration Options” on page 10 .

- **internalLossRatio**

For details, see “General Configuration Options” on page 10 .

- **loginRateLimit (Integer)**

Specifies the number of allowed logins when rate-limiting. As many logins as this are allowed during each rate-limiting period as specified in option `rateLimitDuration`.

For example, if option `loginRateLimit` is set to 100, and option `rateLimitDuration` is set to 10, then 100 logins will be allowed every 10 seconds. If the login rate limit quota is exceeded during this period, no further logins are allowed until the start of the next 10 second period.

See also option `rateLimitDuration`.

- **logProbes (Boolean)**

Flag indicating whether login probe attempts should generate a log message.

This might be useful for servers running on the public Internet, to verify users who are attempting to probe and/or hack the server.

See also option `allowProbe`.

- **externalPorts/port (Integer)**

Port that the LoginApp should listen to for login requests.

If set to zero, then a random port is chosen, which is useful when running multiple BigWorld server instances in a LAN. The client would then need to auto-detect the available servers. For more details, see “Auto-Detection of LoginApps” on page 107.

Multiple port entries may be specified if necessary in order to provide a range of ports to use on a machine. This is useful if multiple LoginApps are intended to be used on the same machine, and would generally be used in conjunction with the option `shouldShutDownIfPortUsed`. For example:

```
<loginApp>
  <externalPorts>
    <port> 20013 </port>
    <port> 20014 </port>
    <port> 20015 </port>
  </externalPorts>
  <shouldShutDownIfPortUsed> true </shouldShutDownIfPortUsed>
</loginApp>
```

- **maxLoginDelay (Float)**

If the response from the LoginApp to the client is lost, the successful login needs to be resent to the client. This setting is the number of seconds that a successful login attempt is kept.

- **privateKey (String)**

This is the filename to read the LoginApp's RSA private key from. This is the key that is used to decrypt the client's login credentials.

Please see *Encrypting Client-Server Traffic* for more information about the encryption support in BigWorld.

- **rateLimitDuration (Integer)**

Specifies the rate-limiting time period in seconds. As many logins as is specified in option `loginRateLimit` are allowed during every one of these periods.

For example, if option `loginRateLimit` is set to 100, and option `rateLimitDuration` is set to 10, then 100 logins will be allowed every 10 seconds. If the login rate limit quota is exceeded during this period, no further logins are allowed until the start of the next 10 second period.

Setting this value to 0 disables login rate limiting.

See also option `loginRateLimit`.

- **registerExternalInterface (Boolean)**

If set to true, the LoginApp's external interface will be registered with `bwmachined`. This may be useful during development to allow clients to discover running server. This should be set to false in a production environment.

- **shouldShutDownIfPortUsed (Boolean)**

Flag indicating whether the LoginApp should shut down if the specified port to listen to is not available.

If set to true, the LoginApp will shut down if the specified ports to listen to is not available. If set to false, the LoginApp will find another port to listen to.

See also option `port`.

- **shutDownSystemOnExit (Boolean)**

Flag indicating whether the server should be also shut down when LoginApp is shut down.

If set to `true`, then a controlled shutdown is performed on the server.

- **verboseExternalInterface (Boolean)**

Flag indicating whether to generate verbose log output related to external network traffic.

What is verbose about it? what should they expect to see and why would they want to use it

2.16. Reviver Configuration Options

The Reviver configuration options are specified in file `<res>/server/bw.xml` under the reviver section, and are described below:

- **internalLatencyMax**

For details, see “General Configuration Options” on page 10 .

- **internalLatencyMin**

For details, see “General Configuration Options” on page 10 .

- **internalLossRatio**

For details, see “General Configuration Options” on page 10 .

- **pingPeriod (Float)**

Number of seconds between pings sent to monitored processes for execution check purposes.

Reviver's monitor processes periodically by pinging them to check that they are still running and functioning normally.

See also option `subjectTimeout`.

- **reattachPeriod (Float)**

Number of seconds between each time that Reviver checks processes to determine if it can monitor them.

If another Reviver stops monitoring a process, this option allows this Reviver to start monitoring that process.

See also option `subjectTimeout`.

- **shutDownOnRevive (Boolean)**

Flag indicating whether the Reviver itself should be shut down after reviving a monitored process.

This is usually set to `true`, since once a Reviver has started a process, that machine should probably be considered busy.

See also option `subjectTimeout`.

- **subjectTimeout (Float)**

Number of seconds that a monitored process will wait for a response from its current Reviver before accepting to be monitored by another one.

If a monitored process does not receive a response from its current Reviver, then it is assumed that the Reviver has been stopped after reviving another one of its monitored processes (this behaviour is set by option `shutdownOnRevive`).

See also options `pingPeriod`, `reattachPeriod` and `shutdownOnRevive`.

- **timeoutInPings (Integer)**

Number of pings that can be missed by a monitored process before Reviver assumes it is dead.

2.16.1. Reviver's BaseAppMgr Configuration Options

The BaseAppMgr configuration options are specified in file `<res>/server/bw.xml` under section `<reviver>/<baseAppMgr>`. These are the same as general Reviver options, but specific to BaseAppMgr. If the setting is not specified, then the general one is used. For details on these options, see “Reviver Configuration Options” on page 47 .

The options are listed below:

- **pingPeriod (Float)**
- **subjectTimeout (Float)**
- **timeoutInPings (Integer)**

2.16.2. Reviver's CellAppMgr Configuration Options

The CellAppMgr configuration options are specified in file `<res>/server/bw.xml` under section `<reviver>/<cellAppMgr>`. These are the same as general Reviver options, but specific to CellAppMgr. If the setting is not specified, then the general one is used. For details on these options, see “Reviver Configuration Options” on page 47 .

The options are listed below:

- **pingPeriod (Float)**
- **subjectTimeout (Float)**
- **timeoutInPings (Integer)**

2.16.3. Reviver's DBMgr Configuration Options

The DBMgr configuration options are specified in file `<res>/server/bw.xml` under section `<reviver>/<dbMgr>`. These are the same as general Reviver options, but specific to DBMgr. If the setting is not specified, then the general one is used. For details on these options, see “Reviver Configuration Options” on page 47 .

The options are listed below:

- **pingPeriod (Float)**
- **subjectTimeout (Float)**
- **timeoutInPings (Integer)**

2.16.4. Reviver's LoginApp Configuration Options

The LoginApp configuration options are specified in file `<res>/server/bw.xml` under section `<reviver>/<loginApp>`. These are the same as general Reviver options, but specific to LoginApp. If the

setting is not specified, then the general one is used. For details on these options, see “Reviver Configuration Options” on page 47 .

The options are listed below:

- **pingPeriod (Float)**
- **subjectTimeout (Float)**
- **timeoutInPings (Integer)**

Chapter 3. Cluster Administration Tools

A variety of tools are provided to assist in managing a BigWorld server. Broadly, these fall into four categories:

- **WebConsole**
- **Logging Processes**
 - **Message Logger** (See “Message Logger” on page 55 .)
 - **Stat Logger** (See “StatLogger” on page 59 .)
- **Server command-line utilities (e.g., `control_cluster.py`)** (See “Server Command-Line Utilities” on page 67 .)
- **Space Viewer** (See “Space Viewer” on page 70 .)

The server tools are implemented almost entirely in Python, and can be easily extended, if additional functionality is desired.

Most of the server tools' functionality is available from WebConsole, an easy to use web interface which simplifies server administration for remote users and non-programmers (e.g., artists, game designers, etc.) that may need to run and administer servers.

Almost all server tool functionality is also available via command-line utilities, which is useful when only system console access is available (e.g., when remotely administering a server cluster via ssh).

Where possible, documentation for the server tools is maintained as online help. For WebConsole, this means the Help link displayed in the navigation menu for each module. For command-line utilities, help is available via the `--help` switch.

This chapter provides an overview of the suite of tools available. For detailed documentation, however, please refer to the online help.

3.1. WebConsole

Located under `bigworld/tools/server/web_console`, WebConsole provides a simple web interface to manage and monitor a BigWorld server. For details on how to install and run WebConsole, see the document Server Tools Installation Guide.

3.1.1. Modules

WebConsole is functionally divided into units referred to as *Modules*. Each module in WebConsole has a separate top level menu item in the main navigation menu on the left hand side of the page.

Complete documentation for each WebConsole module is provided as online help, accessible as a link on the left hand side of the WebConsole page.

WebConsole is composed of the modules described below.

3.1.1.1. ClusterControl

- Allows users to start, stop, and restart the game server.
- Browse the active machines and users on the local network.
- View and modify saved cluster configurations.

3.1.1.2. Watchers

- Allows users to view and modify Watcher values of active server processes.

- Generate pre-configured and user generated tabular reports of watcher values across sets of server processes in a cluster.
- Define custom sets of watcher values for live monitoring.

3.1.1.3. LogViewer

- Allows users to view, filter, and search server message logs.
- Provides a real time / live view of server output.
- Summaries of per-user log usage.

3.1.1.4. StatGrapher

- Provides live and historic graphic views of server statistics.

3.1.1.5. Python Console

- Allows users to connect to the Python server of any Python enabled process (*i.e.*, CellApp, BaseApp, Bots).

3.1.1.6. Commands

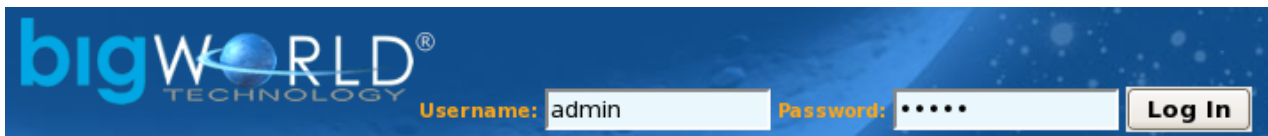
- Provides access to a set of pre-defined functionality on server processes that can be run from WebConsole as required.

3.1.2. Installation and Configuration

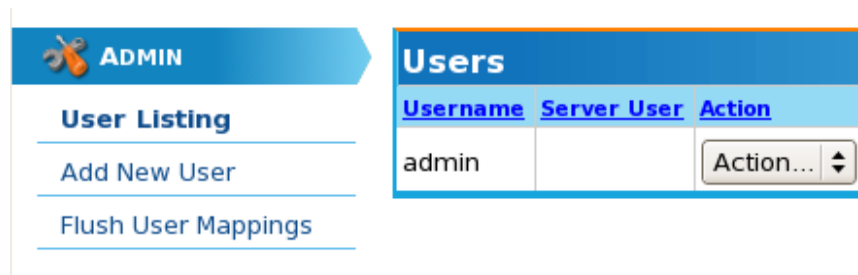
For detailed instructions on installing and configuring WebConsole please refer to the Server Tools Installation Guide.

3.1.3. User Administration

The first time WebConsole is started you will need to add at least one user in order to operate a BigWorld server. To achieve this, from the front page of WebConsole, login with a username / password of admin. If you have never loaded the WebConsole homepage before, it should be visible at the url `http://hostname:8080`, where *hostname* is the network name of the machine you install WebConsole on.



Once logged in you should be presented with a webpage similar to the following.



3.1.3.1. Adding A New User

To add a new user, click on the "Add New User" menu item on the left hand side of the page. You will then be presented with a form to input as follows:

Add A New User

Username	<input type="text"/>
Password	<input type="password"/>
Confirm Password	<input type="password"/>
Server User	<input type="text"/>
<input type="button" value="Add User"/>	

The table below summarises the input fields providing sample input:


Field	Description
Username	The username that for the new user.
Password	The password that the new user will login with.
Confirm Password	The password from the previous field, retyped to ensure no mistake was made on entry.
Server User	The Linux user this account will be associated with. This will be the Linux user that the BigWorld server processes will be run as. If you are uncertain of this field, please talk to your system administrator to find out what your Linux user account is.

The following image shows a new user account AliceB being created and associated with the Linux user account alice.

Add A New User

Username	AliceB
Password
Confirm Password
Server User	alice
<input type="button" value="Add User"/>	

Once all the user information has been input, simply click the Add User button and you will be returned to the main user listing with the new user in the list.

 **ADMIN**

User Listing
[Add New User](#)
[Flush User Mappings](#)

Users

Username	Server User	Action
AliceB	alice	Action... ▾
admin		Action... ▾

3.1.3.2. Flush User Mappings

The Flush User Mappings menu option provides the ability for WebConsole to force all BWMachine instances in the network to forget their current cached user list. This feature is useful and sometimes necessary when adding users to a network information system such as LDAP which are sometimes used as account management systems.

If you are having problems detecting a newly added user in your network, or are seeing an old user appear in your cluster display, flushing the user mapping will quite often help resolve the issue.

3.1.4. Production Mode vs Development Mode

WebConsole can be started using two methods, as a system wide service¹, or from the command line.

Generally, only developers working on modifying the web page templates or underlying code will run WebConsole from the command line. In order to achieve this, make sure that the current directory is `bigworld/tools/server/web_console`, then issue the command:

```
$ ./start-web_console.py
```

There are some minor operational differences when running WebConsole in command line mode vs as a system service:

- When run as a system daemon, WebConsole uses the configuration file `prod.cfg`, which defines a production environment mode.
- When run from the command line, WebConsole uses a development environment configuration file called `dev.cfg`.

Running in development mode leaves the web server in a state where an automatic restart is triggered if there are any changes to the template files or Python code it is using.

The production configuration file does not exist by default in `bigworld/tools/server/web_console`, as it is partially generated during the installation process while using the `install_tools.py` script. The original production configuration file that the installation script uses is `bigworld/tools/server/install/web_console.cfg`.

Both configuration files need to specify the database to use. Before running WebConsole, make sure that the appropriate configuration file has a line that looks like this:

```
sqlobject.dburi="notrans_mysql://username:password@localhost:3306/  
bw_web_console"
```

Specifying the database

To modify the `bigworld/tools/server/install/web_console.cfg` file to a working configuration file, replace the line:

```
###BW_SQLURI###
```

with an `sqlobject.dburi` line as indicated above.

For information regarding TurboGears configuration files and content, we recommend the TurboGears documentation website <http://docs.turbogears.org/1.0/Configuration>.

¹For details see the Server Tools Installation Guide's section *Post-installation*.

If you have never run WebConsole before and choose to run it from the command line (as opposed to installing the system service), it is necessary to create a database for WebConsole within MySQL. To do this, connect to your MySQL database using the username and password that you have defined in `dev.cfg`, then issue the following command:

```
CREATE DATABASE bw_web_console;
```

3.1.5. Customising

As WebConsole is a convenient interface to your cluster environment, it may be useful to add new content to WebConsole that is unique to your companies requirements. To achieve this please refer to the Server Programming Guide chapter *Web Console*.

3.2. Message Logger

Message Logger is a process used for logging output from BigWorld processes within a network. This is primarily intended as a mechanism for logging output from server processes, however in a development environment it can be useful for game clients and other tools to send log messages as well. The source code for Message Logger can be located in `bigworld/src/server/tools/message_logger`, while its executable is found in `bigworld/tools/server/bin`. Message Logger is a stand alone Linux process that can be run on any machine in a BigWorld cluster that is running BWMachined.

A suite of Python tools and libraries is also provided for accessing and manipulating logs that have been generated by Message Logger. These tools can be found in `bigworld/tools/server/message_logger`.

Detailed documentation for the supported command line arguments is available as online help using the `--help` option.

When a server process, such as a CellApp, is started it performs a search of the network cluster it is operating within for all active Message Logger processes. Any output that is generated as a result of script `print` statements or C++ calls to the message handlers (`INFO_MSG`, `ERROR_MSG`, etc....) are sent to all known Message Logger instances. Message Logger processes that are started after a server process will notify all running server components of their existence and commence logging of all subsequent messages.

The BigWorld Client can also be configured to send messages to Message Logger. By default this is enabled in the Hybrid build and disabled in the Release build. The Consumer build logging options are controlled by a set of `#defines` in `src/lib/cstdmf/config.hpp`.

The following values should be defined in the Consumer build to enable the Client to send log messages to Message Logger instances:

```
#define FORCE_ENABLE_MSG_LOGGING    1
#define FORCE_ENABLE_DPRINTF        1
#define FORCE_ENABLE_WATCHERS       1
```

Messages from the client are by default sent as the `root` user (i.e., with a linux UID of 0). In order to associate logs generated from a Client with a specific user within a network, a Windows environment variable `UID` can be set. This variable should have a value corresponding with the user's Linux account user id. For example a developer Alice with a Linux account name of `aliceb` could discover her user id on Linux by logging in to her Linux account and typing the following command:

```
$ id
uid=3592(aliceb) gid=501(UmbrellaCorp)
```

Alice can then set a Windows environment variable of `UID` with the value 3592. Any logs generated by Alice running the BigWorld Client will now be associated with her.

3.2.1. Configuration

Except where overridden by the corresponding command line option, Message Logger reads its configuration from the file `bigworld/tools/server/message_logger/message_logger.conf`.

The configuration file is in standard Windows INI file format, and supports the following options:

Option Name	Description
<code>logdir</code>	The location of the top level directory to which Message Logger will write its logs. This option can be either a relative or an absolute path. If a relative path is specified, then it is calculated relative to the location of the configuration file.
<code>segment_size</code>	Size (in bytes) at which the logger will automatically roll the current log segment for a particular user.
<code>default_archive</code>	File used by <code>mltar.py</code> when the <code>--default_archive</code> option is used. This file is also inserted into Message Logger's logrotate script during installation.
<code>groups</code>	A comma separated list of machine group names. When specified MessageLogger will only accept log messages originating from machines that have a matching group name listed in the <code>[Groups]</code> section of <code>/etc/bwmachined.conf</code> . For more details see "Production Scalability" on page 58 .

3.2.2. File Format

Message Logger generates log files in a binary format in order to make the best use of disk space as possible. In order to access the binary logs, two Python modules are provided:

Python Module	Description
<code>bwlog.so</code>	Location: <code>bigworld/tools/server/bin/Hybrid64</code> A binary Python extension compiled from Message Logger source code.
<code>message_log.py</code>	Location: <code>bigworld/tools/server/message_logger</code> Python classes that simplify and extend the functionality exposed from <code>bwlog.so</code> .

For examples on how to use these modules, browse the source code of the command-line message logger utilities described in "Command Line Utilities" on page 57 .

Message Logger generates logs in a two level directory structure. The top level directory contains files that are common to all users, and contains one sub directory per user. The files in the top level directory are:

Filename	Description
<code>component_names</code>	List of all distinct component names (<i>i.e.</i> , <code>CellApp</code> , <code>BaseApp</code> , etc...) that have sent messages to this logger. This is used to resolve numeric component type IDs to names, when displaying log output.
<code>hostnames</code>	Mapping from IP addresses to hostnames. This is used for resolving hostnames when displaying log output.
<code>strings</code>	List of all unique format strings that have been sent by logging components, along with parser data for interpreting arguments to each. This is used to reconstruct the log messages from format string IDs and binary argument blobs.
<code>version</code>	Log format version. This is used to prevent accidentally mixing two log formats.
<code>pid</code>	Process id of the active Message Logger process writing logs to this directory. This is used by both the tools and Message Logger to identify the process (if any) currently generating logs.

Each user's sub directory has a file containing its UID, as well as the following files:

Filename	Description
components	A record of each individual process instance registered with this logger.
entries.<timestamp>, args.<timestamp>	<p>A collection of log messages.</p> <p>The fixed length portion of each log message (time, component ID, etc...) is stored in the entries.<timestamp> file, while the variable-length portion of the entry (<i>i.e.</i>, arguments to printf) is stored in the corresponding args.<timestamp> file.</p> <p>The variable length section data corresponds to an entry in the format string file strings.</p>

3.2.3. Command Line Utilities

The `bigworld/tools/server/message_logger` directory contains a variety of command line utilities providing the functionality of standard UNIX shell utilities for BigWorld message logs. Using these utilities you can operate on the binary logs using standard UNIX shell utilities and pipelines in the same way you can with ordinary text files.

The following tools are provided:

Utility	Description
mlcat.py	Provides both cat and tail -f style of text log outputting.
mls.py	Displays information about log segments, such as start and end times, entry counts, and sizes; for individual users or all of them.
mltar.py	Provides an easy way to select log segments from a user's log and archive them, as well as all the required shared files for viewing on another machine.
mlrm.py	Provides an easy way to clean up unwanted log segments.

The detailed documentation for each utility is maintained as online help, which can be accessed via the `--help` option.

The following list provides some common examples of tasks you might wish to achieve using the Message Logger tools:

- **mlcat.py -f**
Watches live server logs.
- **mlcat.py --around="Mon 22 Jan 2007 19:00:00" -u devuser**
Views output surrounding a log entry of interest for user devuser.
- **mltar.py -zcf bwsupport.tar.gz --active-segment**
Collects logs to email to support.
- **mltar.py -u <uid> -zcf bwsupport.tar.gz --active-segment**
Collects logs for the Unix user whose uid is uid to email to support.
- **mls.py -u gameuser**
Displays all log segments for user gameuser, to determine the segment with entries of interest.
- **mlrm.py --days=30**

Removes logs over a month old.

- `mltar.py -xf lastweek.tar.gz -o samplelogs`

Extracts the archive `lastweek.tar.gz` to the directory `samplelogs`.

Note

Since the archives are compressed tar files, you can use `tar` to achieve the same results if you find that easier, as illustrated below:

```
$ mkdir samplelogs
$ tar -zxvf lastweek.tar.gz -C samplelogs
```

- `mltar.py -xd`

Extracts the latest archive back to the default logdir.

3.2.4. Automatic Log Archiving

The Linux `logrotate` scripts are used to rotate (stop writing an old log file and start writing to a new file), archive (backup the old log file), and delete (log files deemed to be past an expiry time) log files on a daily basis, which can assist in conserving disk space. A `logrotate` configuration file, `/etc/logrotate.d/bw_message_logger`, is set up as part of the Server Tools installation. For more information about `logrotate`, please see the `logrotate` manpage:

```
$ man 8 logrotate
```

There are two issues to consider when customising the `logrotate`:

- Log rotation can put a load on the logging machine.
- If rotation is configured to occur more frequently, for example the rotation is changed to occur on hourly basis, then in this case the `rotate` option should be updated to 168 (i.e. 7 x 24) to ensure that the log files cover the same period of time.

3.2.5. Production Scalability

In order to assist Message Logger coping with heavy network and disk load in production environments, Message Logger supports logging subsets of machines within a server cluster. This is achieved by assigning a Message Logger process a list of machine groups that it should accept logs from, as well as assigning every machine in the cluster to a machine group. Both Message Logger and cluster machines may be assigned to multiple machine groups. In order to enable this feature both MessageLogger and BWMachined will need to be configured correctly throughout your cluster environment.

To limit a Message Logger process to only accept logs from a set of machine groups requires adding a comma separated list of group names to a `groups` entry in the file `bigworld/tools/server/message_logger/message_logger.conf`. For example, to assign a MessageLogger process to only accept logs from the machine groups `QA_Cluster1` and `QA_Cluster2` the following entry would be placed in `message_logger.conf`:

```
[message_logger]
```

```
...
groups = QA_Cluster1, QA_Cluster2
```

MessageLogger needs to be restarted in order to detect these changes, however once restarted an entry should appear in the MessageLogger log file `/var/log/bigworld/message_logger.log` similar to the following:

```
2008-11-12 11:37:20: INFO: Logger::init: MessageLogger machine groups
specified. Only accepting logs from: QA_Cluster1, QA_Cluster2
```

For details on configuring BWMachined see the Server Installation Guide section ????

Note

Currently in order to view the logs generated when using the machine group log separation will require either separate installations of WebConsole, or manual searching of the files with the `mlcat.py` tool on the host where MessageLogger is running. This limitation will be addressed in a future release of BigWorld.

3.3. StatLogger

Written in Python, StatLogger is a daemon process that runs in the background, polling all servers on the network at regular intervals (by default, every 2 seconds). Any computer running BWMachined will be automatically discovered, along with any BigWorld components running on that machine. StatLogger collects and logs information for every server component discovered, regardless of which user is running them.

Statistics for machines and processes are collected in 2 ways:

- Communication with BWMachined daemons running on each computer.
- Requests made directly to the processes via the Watcher mechanism.

Once collected, StatLogger logs this data to a MySQL database. For details on the structure of the database, see “Database Structure” on page 66 .

The main objective of StatLogger is to collect and store data in a format that can be used by StatGrapher², which is the visualisation counterpart to StatLogger, and presents the data in a graphical format.

3.3.1. Requirements

The list below describes the requirement for running StatLogger:

- **CPU**

On an Athlon 2700+, monitoring 230 processes (which includes 200 CellApps) with StatLogger consumes roughly a constant 15% of available CPU time. The MySQL daemon also experienced a constant load increase of 5.5% due to the large amount of database queries being generated.

- **Disk**

Due to the amount of data being collected, StatLogger can potentially consume a lot of disk space. The rate at which disk space is consumed depends on the amount of machines and server process for which

²For details, see “StatGrapher” on page 52 .

statistics are collected. 4GB is recommended for a large amount of processes (around 250) and machines over a month. The disk space consumption rate of a single log gets smaller the longer a log is run, since StatLogger stores older data in lower detail than newer data.

- **Network**

StatLogger's network requirements depend on the number of server machines and components present. It can potentially require a large amount of network throughput. For example, 230 processes and 9 server machines require 100Kb/s of downstream traffic, and 20kb/s of upstream traffic, while 6 processes (*i.e.*, for a minimal server) and 9 server machines require 4kb/s of downstream and 2kb/s of upstream traffic.

- **RAM**

Memory requirements are low, as statistics are immediately logged to the database, rather than being kept in memory. Tests indicate an average usage of memory between 7MB to 10MB, regardless of the amount of processes running.

- **Software**

MySQL 4.1+ and Python 2.4 with MySQLdb.

StatLogger must be run on the same local network as the BigWorld server processes that it is intended to monitor due to the manner in which machine and server component discovery occurs.

StatLogger requires a valid MySQL user with access to create databases. Normally the creation and configuration of this MySQL user is handled during the tools installation process³, but an existing database user can be used instead, by manually editing the configuration file. For details, see "Configuration" on page 60 .

3.3.2. Output

StatLogger outputs various status messages describing which machines and processes it discovers or loses (*i.e.*, from the process or machine shutting down) from the network, each prefixed with a timestamp.

When running StatLogger manually from the command line the default is to display this information to the terminal on which it is run unless the `-o` option is specified.

When installed as a daemon, these messages output to the log file `/var/log/bigworld/stat_logger.log` by default. This log file is specified by the `LOG_FILE` variable in `/etc/init.d/bw_stat_logger`.

3.3.3. Data Collection and Aggregation

As Stat Logger is responsible for collecting a constant stream of data from an arbitrarily large BigWorld server cluster for historical tracking it is necessary to prevent the database size from growing out of proportion. In order to achieve this Stat Logger implements a mechanism of aggregating and merging data from high resolution time periods into lower resolution time scales the older the data becomes. This approach allows high resolution data to be maintained for recent events when diagnostics are most likely to occur but allows a broad view of server performance over a long scale to be maintained. More information on customising how StatLogger aggregates data can be found below in "Aggregation Window Configuration" on page 61 .

3.3.4. Configuration

Located in StatLogger's directory, the preference file `preferences.xml` is used to configure the script. The preference file can be used to configure:

³For details, see the Server Tools Installation Guide.

- Database connection details.
- Watcher values to collect from processes.
- Aggregation preferences.

Apart from the database user configuration, in most cases the provided standard settings should be sufficient for development and production environments.

The preferences.xml file configures database setup options, as well as what data StatLogger should collect from the server components and machines.

3.3.4.1. Database Connection Details

The <options> section of the preferences file contains the following tags:

Option Name	Description
dbHost	Hostname or IP address of the MySQL database server that will contain the log databases.
dbUser	Database user with which StatLogger (and StatGrapher) will access the database. For details on StatGrapher, see "StatGrapher" on page 52 .
dbPass	Database user's password.
dbPort	Port on which the MySQL server is listening for connections.
dbPrefix	Prefix of database names which StatLogger can access.
sampleTickInterval	Interval in seconds at which StatLogger will poll the components and store statistics (decimals are supported). Generally this value does not need to be changed, however if it does, then it should not be any smaller than 2 seconds (the default recommended value).

3.3.4.2. Aggregation Window Configuration

The <collect/aggregation/window> section configures the aggregation windows that will be stored in the database.

As outlined in "Data Collection and Aggregation" on page 60, StatLogger has been designed to store multiple versions of data at varying levels of detail with the idea that long term data does not need to be stored at the same level of detail as the more recent data.

StatLogger requires at least one aggregation window setting in this section with a samplePeriodTicks value of 1.

Multiple aggregation window setting takes the form of:

```
<preferences>
...
<collect>
  <aggregation>
    *<window>
      <samples>          <num_samples_in_this_win>    </samples>
      <samplePeriodTicks> <num_of_ticks_per_sec>      </samplePeriodTicks>
    </window>
  ...

```

Multiple aggregation window settings (Grammar)

There are some constraints that must be adhered to when creating this list of aggregation window settings:

- There must always be an aggregation window with samplePeriodTicks value of 1.
- Aggregation window settings must be ordered in ascending order by their samplePeriodTicks value, with the smallest values first.
- Each successive aggregation window should cover a larger range of ticks than the previous one. The tick range is calculated by multiplying samples value by the samplePeriodTicks value (*i.e.*, number of samples x ticks consolidated into one sample).
- Each successive samplePeriodTicks value must be a multiple of the samplePeriodTicks value from the previous window.

These aggregation windows are used directly by StatGrapher, so it is advised not to have large discrepancies between the samplePeriodTick values of successive aggregation windows. Furthermore, the final aggregation window setting should not have a large samples value, as this may place a heavy load on both StatLogger and StatGrapher.

An example aggregation section is shown below, with a samples value of 365:

```
<preferences>
...
<collect>
  <aggregation>

    <!-- Every sample (2secs) in most recent 24hrs. 43200 samples -->
    <window>
      <samples>          43200 </samples>
      <samplePeriodTicks> 1    </samplePeriodTicks>
    </window>

    <!-- Every 10th sample (20secs) in most recent 48hrs. 8760samples -->
    <window>
      <samples>          8760 </samples>
      <samplePeriodTicks> 10   </samplePeriodTicks>
    </window>

    <!-- Every 150th sample (5mins) in most recent 30 days. 8760 samples -->
    <window>
      <samples>          8760 </samples>
      <samplePeriodTicks> 150  </samplePeriodTicks>
    </window>

    <!-- Every 1800th sample (60mins) in most recent 365 days. 8760 samples
-->
    <window>
      <samples>          8760 </samples>
      <samplePeriodTicks> 1800 </samplePeriodTicks>
    </window>

    <!-- Every 43200th sample (1day) in most recent 365 days. 365 samples
-->
    <window>
      <samples>          365   </samples>
      <samplePeriodTicks> 43200 </samplePeriodTicks>
    </window>
  </aggregation>
...
```

Multiple aggregation window settings (Example)

3.3.4.3. Machine Statistic Configuration

The `<collect/machineStatisticList>` section is similar to the `statisticList`⁴ and `allProcessStatisticList`⁵ sections, except in that watcher values are not supported since machines do not have a Watcher interface. The `<valueAt>` settings only support strings representing members of the Machine class defined in `bigworld/tools/server/pycommon/machine.py`.

3.3.4.4. Generic Process Statistic Configuration

The `<collect/allProcessStatisticList>` section is similar to the `statisticList`⁶ section, except that its statistics are regarded as common to all processes being monitored *i.e.*, those specified in the `<processList>`⁷ section.

It is recommended that any watcher values used in this section are supported by all processes being monitored, although StatLogger will store empty values for processes that do not support a watcher value.

3.3.4.5. Process Configuration

The `<collect/processList/process>` section configures the statistics that will be collected for each process. There must be one process section for each server component to be monitored.

The process section contains the following tags:

Tag Name	Description
name	Display name for the process type.
matchtext	Name of the component type's executable. Value must be in lowercase, like the executable names.
statisticList	List of statistics to be collected for this process by StatLogger ^a .

^aFor details see "Process Statistic Configuration" on page 64 .

An example process section is shown below:

```
<preferences>
...
<collect>
...
  <processList>
    <process>
      <name>      CellApp </name>
      <matchtext> cellapp </name>
      <statisticList>
        ...
      </statisticList>
    </process>
  ...
</processList>
```

Example process configuration

⁴For details see "Process Statistic Configuration" on page 64 .

⁵For details see "Generic Process Statistic Configuration" on page 63 .

⁶For details see "Process Statistic Configuration" on page 64 .

⁷For details see "Process Configuration" on page 63 .

3.3.4.6. Process Statistic Configuration

The `<collect/processList/process/statisticList/statistic>` section specifies the statistics that must be collected for each process.

The statistic section contains the following tags:

Tag Name	Description
name	Display name for the statistic type.
valueAt	Where to retrieve the values from. There are two distinct sources of information, depending on the first character of this tag's value, see the section "valueAt Properties" on page 65 for details.
maxAt	This value is not explicitly used in StatLogger, however it is used by StatGrapher to determine the scale of the graph.
type	Current unused.
consolidate	Consolidation function to use when moving data up an aggregation window. Possible values are: MAX, MIN, and AVG. Example: We consolidate four data samples at 4 seconds per sample into the next aggregation window, which stores data at 16 seconds per sample. The data represents CPU load consumed by a process. The <code>consolidate</code> value is specified as <code>AVG</code> , so StatLogger averages the four data samples, then store this value as a single value in the higher aggregation window.
display	Contains properties that influence the StatGrapher display of this statistics.

An example statisticList section is shown below:

```
<preferences>
...
<collect>
...
  <processList>
    <process>
      <name>          CellApp </name>
      <matchtext>    cellapp </name>
      <statisticList>
        <statistic>
          <name>          Cell Load          </name>
          <valueAt>        /load              </valueAt>
          <maxAt>          1.0                </maxAt>
          <logicalMax>     None               </logicalMax>
          <type>           FLOAT              </type>
          <consolidate>    AVG                </consolidate>
          <display>
            <colour>      #FF6600            </colour>
            <show>        true               </show>
            <description>
              The load of this CellApp.
            </description>
          </display>
        </statistic>
        <statistic>
          <name>          Num Entities Ever    </name>
          <valueAt>        /stats/totalEntitiesEver </valueAt>
          <maxAt>          10000.0            </maxAt>
          <type>           FLOAT              </type>
          <consolidate>    MAX                </consolidate>
          <display>
            <colour>      #663366            </colour>
```

```

        <show>         false                                </show>
        <description>
            The number of entities created on the CellApp since server
startup.
        </description>
        </display>
    </statistic>
    ...

```

Process statistic configuration — Example

3.3.4.6.1. valueAt Properties

The behaviour of the `valueAt` tag changes based on the first character of the string. If first character is a slash '/' `valueAt` will be interpreted as a watcher path. The best way to list the watchers that can be graphed is via the WebConsole's ClusterControl module⁸.

Any single scalar value present in ClusterControl can be added to StatGrapher, including any customer watcher values added within your own game code.

If the first character is not a slash, the `valueAt` string is treated as referring to a member a `Process` class instance as found in `bigworld/tools/server/pycommon/process.py`. Strictly speaking, `valueAt` is eval'd against the `Process` object which enables slightly more flexibility than just being able to reference class members.

Examples:

valueAt	Interpretation
/stats/numInAoI	Watcher path of <code><process type>:/stats/numInAoI</code>
mem	Evaluated to <code>Process.mem</code> , which retrieves the value of memory usage of a process.

Note

Caution is required with this type of value as it is possible to use a function call that changes the state of the process (*e.g.*, causing a process to shutdown). It is highly unlikely that this will occur by accident, unless the exact function call is entered in the `valueAt` setting.

3.3.4.7. Further Notes on Configuration

The configuration file directly affects the database structure used for storing the statistics. If the collect section of the configuration file is changed, then StatLogger will detect the change when it is next run, and will subsequently create a new statistics database from scratch, in order to accommodate the new structure.

3.3.5. Database

StatLogger uses two active databases during its operation, however over an installation's life multiple databases may be created as StatLogger options are modified.

The two active databases required by StatLogger are:

⁸For details see "ClusterControl" on page 51 .

1. Statistics database associated with a unique preference configuration.
2. Meta database for tracking all statistic databases known by StatLogger.

By default, the meta database is named `bw_stat_log_info` and contains a single table called `bw_stat_log_databases`. The `bw_stat_log_databases` table stores the names of all the known statistics databases. The meta database is always created if it does not exist in the MySQL server.

The statistics databases by default are named `bw_stat_log_data<n>`, where `<n>` is the incremental database number. Users can choose to create statistics databases with specific names using the `-n` option when running StatLogger.

New statistics databases are created in the following situations:

- There are no statistics databases in the MySQL server.
- The preferences file was structurally changed, which causes a new statistic database to be created. For details on the `preferences.xml` file, see “Configuration” on page 60 .
- StatLogger was started with the `-n <database_name>` command-line option, and there was no statistic database with that name in the MySQL server.

There is typically no need to create new statistics databases unless isolated sets of statistics are needed. This may occur when performing tests with the BigWorld server. Once StatLogger has been configured to suit your environment only one statistics database should be needed.

3.3.5.1. Database Structure

Below is an overview of the tables in a statistics database. This is intended as a quick reference for locating data within the database. For a comprehensive understanding of how the data is used the source code is considered to be the best reference.

Table Name	Description
<code>pref_processes</code>	Process preferences as specified in the configuration file's <code><collect/processList></code> section. For details, see “Process Configuration” on page 63 .
<code>pref_statistics</code>	Statistic preferences specified in configuration file.
<code>seen_machines</code>	Machines observed while StatLogger was running.
<code>seen_processes</code>	Processes observed while StatLogger was running.
<code>seen_users</code>	Users observed while StatLogger was running.
<code>stat_<proc>_lvl_<lvl></code>	Statistics collected for processes of type <code><proc></code> , with <code><lvl></code> index of aggregation.
<code>stat_machines_level_<lvl></code>	Statistics collected for machines with <code><lvl></code> index of aggregation. The higher the aggregation level, the lower the resolution of the data being stored in that table.
<code>std_aggregation_windows</code>	Aggregation level details specified in configuration file's <code>collect/aggregation</code> section.
<code>std_info</code>	<code>sampleTickInterval</code> specified in the configuration file's <code><options></code> section. For details, see “Database Connection Details” on page 61 . Database structure version (Used internally by StatLogger).
<code>std_session_times</code>	Start and end times of when StatLogger was run.
<code>std_tick_times</code>	Timestamps of the start of each interval.
<code>std_touch_time</code>	The last time the database was written to, in database local time. This value is only used by StatGrapher regardless of whether StatLogger is currently logging to this database or not.

3.4. Server Command-Line Utilities

3.4.1. Control Cluster

Location: `bigworld/tools/server/control_cluster.py`

Control Cluster is a command line tool for managing a BigWorld cluster from a text only interface.

For detailed information on all the available commands, please refer to the Control Cluster help by using the `--help` option.

```
$ ./control_cluster.py --help
```

For detailed help information on a specific command simply provide the command name after the `--help` option. For example:

```
$ ./control_cluster.py --help getmany
getmany <processes> [watcher-path]

Queries the specified watcher path on all processes of the specified type
currently active in the cluster. The path can contain '*' wildcards.
```

Control Cluster provides functionality through a number of commands that can be used to interact within a BigWorld server environment. The commands can be discussed in terms of four functional areas.

Server Commands Commands that operate on an entire server configuration to either modify or report its current state. This includes starting and stopping a server, saving the current server layout for reuse and displaying the current server configuration on the command line.

As server commands require interacting with a specific server instance, there is an implied user for performing these commands. Unless otherwise specified the user is taken as the username of the person running the Control Cluster command. The user to perform a command as can be changed if required by using the option `--user`. For example:

```
$ ./control_cluster.py --user bob start
```

Process Commands These commands operation on specific server process instances. They are used for starting and stopping server processes when there is already an active server, as well as interacting with an active server process, for example retrieving watcher values.

As with the server commands, the process commands have an implied user as they operate on server processes that have an associated user ID at runtime. The user you run a command as can be modified at any stage by using the `--user` option.

Profiling Commands Profiling commands are used for diagnosing problems on server processes. While technically operating in the same way that process commands do, they are separated into a unique category to highlight their availability and encourage their usage.

Profiling commands can be used for extracting all manner of performance information from server processes including C++ performance, Python game

script performance, client traffic performance and intra-process network statistics.

Cluster Commands The cluster commands are different to the server and process commands in that there is no implied user for them. These commands operate on the entire cluster as represented by every machine in the network that has a BWMachined instance running and is reachable through broadcasting. Operations such as querying all machines in the cluster for their current load and checking the validity of the BWMachined ring are included in these commands.

3.4.1.1. Machine Selection

Many commands require a machine or machines to be provided to perform a command on. As commands quite often need to be performed across large sets of machines at the same time, Control Cluster provide a number of different shorthand notations which can be used anywhere a <machine> or <machines> is requested.

For example the start command requires a machine set to operate on as seen by the help output.

```
$ ./control_cluster.py help start
start <machines>

Start the server on the specified set of machines. To start the server on
all machines, you must explicitly pass 'all' for machine selection.
```

The following list outlines the different machine selection approaches that can be used.

- **All machines**

All machines in the network can be referred to using the syntax `all`.

- **Hostname**

An exact hostname such as `bw01`, `dmz-qa-10`.

- **IP Address**

An exact IPv4 address such as `192.168.1.34`.

- **Hostname Wildcard**

A hostname containing a wildcard such as `dmz-qa-*`. Be aware that depending on the unix shell you use (e.g., bash or zsh) you may have to quote the wildcard `*`.

- **BWMachined Version**

BWMachined has a protocol version associated with each instance. If you are running multiple versions of BWMachined within your network (a practice discouraged by BigWorld), you can specify all hosts using a particular BWMachined version using the following syntax: `version:42` where 42 in this case refers to protocol version 42.

- **BWMachined Groups**

Using the `[Groups]` tag in `/etc/bwmachined.conf` to specify different groups of machines within your server cluster, it is then possible to refer to these groups using the syntax `group:groupname`.

This information is also available from the command line help by using the help command `machine-selection`.


```
$ ./control_cluster.py help machine-selection
```

3.4.1.2. Process Selection

Similar to machine selection, many commands can operate on numerous processes at the same time. For example the `getmany` command can select watcher values from many processes to provide a tabular report. In most cases whenever `<processes>` is seen in the help output of a command, you can use one of the following approaches for supplying sets of processes.

The following list outlines the different process naming schemes that can be used with commands.

- **All processes**

All processes in a server cluster can be referred to using the statement `all`. This is especially useful in relation to watcher commands.

- **Process name**

An exact process name such as `cellapp01` or `cellappmgr`.

- **All processes of a type**

A set of all process types such as `cellapps` or `baseapps`.

- **Single process of a type**

A process type name such as `cellapp` or `baseapp`. This can be useful when a command can be run on any process of the provided type but there may be many processes running within the server cluster.

When using this method of process selection, if there are multiple processes of the specified type running, a random process of the provided type will be selected.

- **Non-zero numbering**

An exact process name that would otherwise have zero's padding the process id, such as `cellapp1` or `baseapp9`.

The advantage of this approach is that it allows the user to take advantage of shell expansion approaches that are already available in different shells such as `bash` and `zsh`. For example in both `bash` and `zsh` using the syntax `cellapp{1..3}` would expand to `cellapp1 cellapp2 cellapp3`, while `baseapp{1,7}` would expand to `baseapp1 baseapp7`.

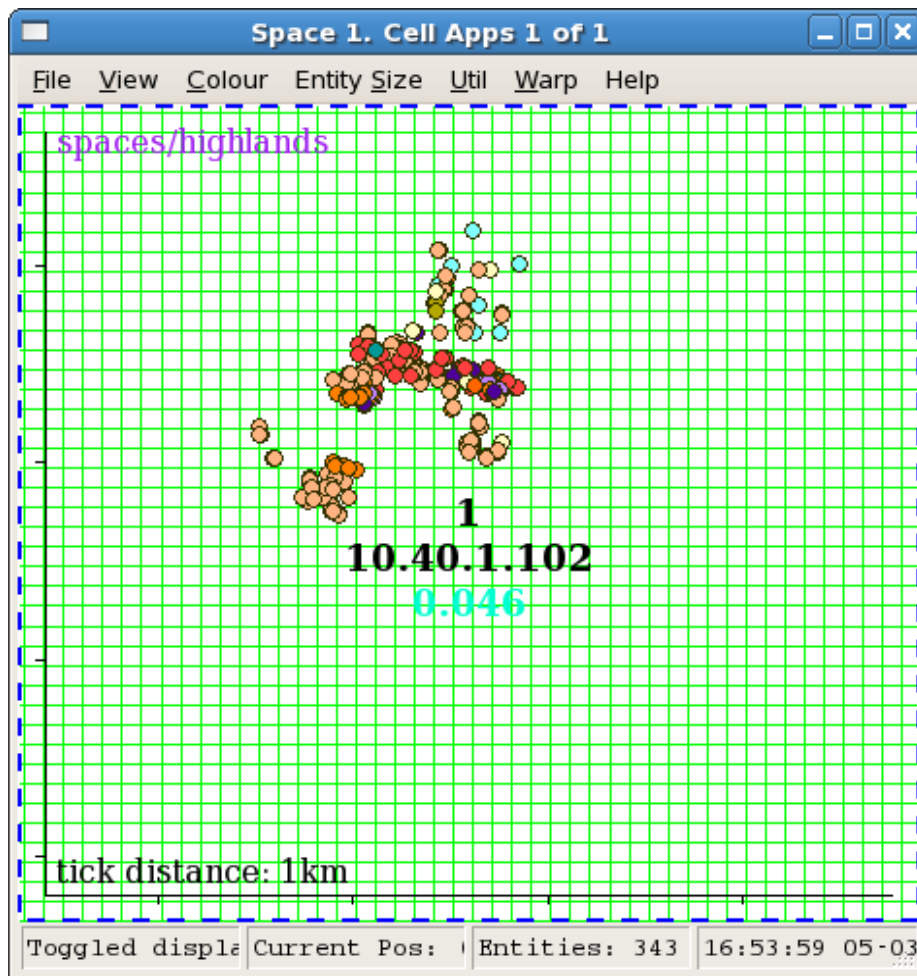
This information is also available from the command line help by using the `help` command `process-selection`.

```
$ ./control_cluster.py help process-selection
```

3.4.2. MessageLogger Related Utilities

For details on the tools available for interacting with server message logs, see “Command Line Utilities” on page 57.

3.5. Space Viewer



Location: `bigworld/tools/server/space_viewer`

Space Viewer is a program that is used for displaying a dynamic graphic representation of spaces. This includes the distribution of cells in a space, as well as the entities within each of the cells.

Space Viewer uses a client / server architecture, and is therefore composed of two distinct parts which may be used simultaneously or separately. The server component (SVLogger) either communicates with an active BigWorld server to collect and log information about a space, or is used to read a previously collected log. The client component (SpaceViewer) connects to the server process and uses a GUI to display the space information provided to it.

For comprehensive documentation on Space Viewer's options and contextual help, please refer to the `--help` output along with the application help available from the Help menu.

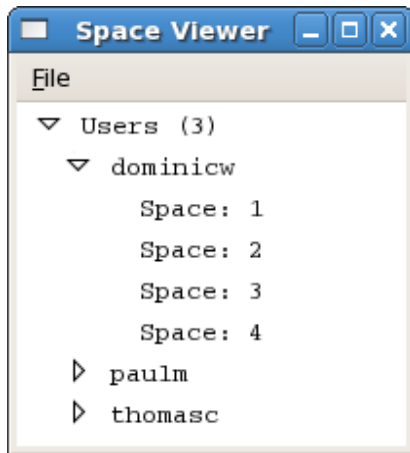
Since the two halves of Space Viewer communicate via TCP, they can be run on different machines, although the client is most responsive when both components are run on the same machine.

The most common use of Space Viewer is to monitor the state of a live BigWorld server, and as such, both the client and server components are run in tandem. This can be achieved by simply running the SpaceViewer script (`space_viewer.py`) when logged in to a GUI session on either Windows or Linux.

Other scenarios such as during scalability testing require running the server component as a separate process. This allows collecting space data over long periods of time without requiring an attached GUI session. In

order to achieve this simply run the SVLogger script (`svlogger.py`) from a Linux command line console. The SpaceViewer GUI can be attached to the server process at any stage by using the `--connect` option of SpaceViewer. For more information on remote monitoring using Space Viewer, see “Running Space Viewer Remotely” on page 73.

The following image illustrates a number of users with active servers within the same network cluster. The dominicw user has been expanded to show all spaces that are active within that server.



Space Viewer window

3.5.1. Selecting Spaces to View

When running SpaceViewer if you do not see your username listed in the main Space Viewer window, select the **File** → **Refresh** menu item. This is sometimes required as the list is not automatically updated.

Clicking the arrow icon to the left of user name expands the list to display all spaces available for that server instance.

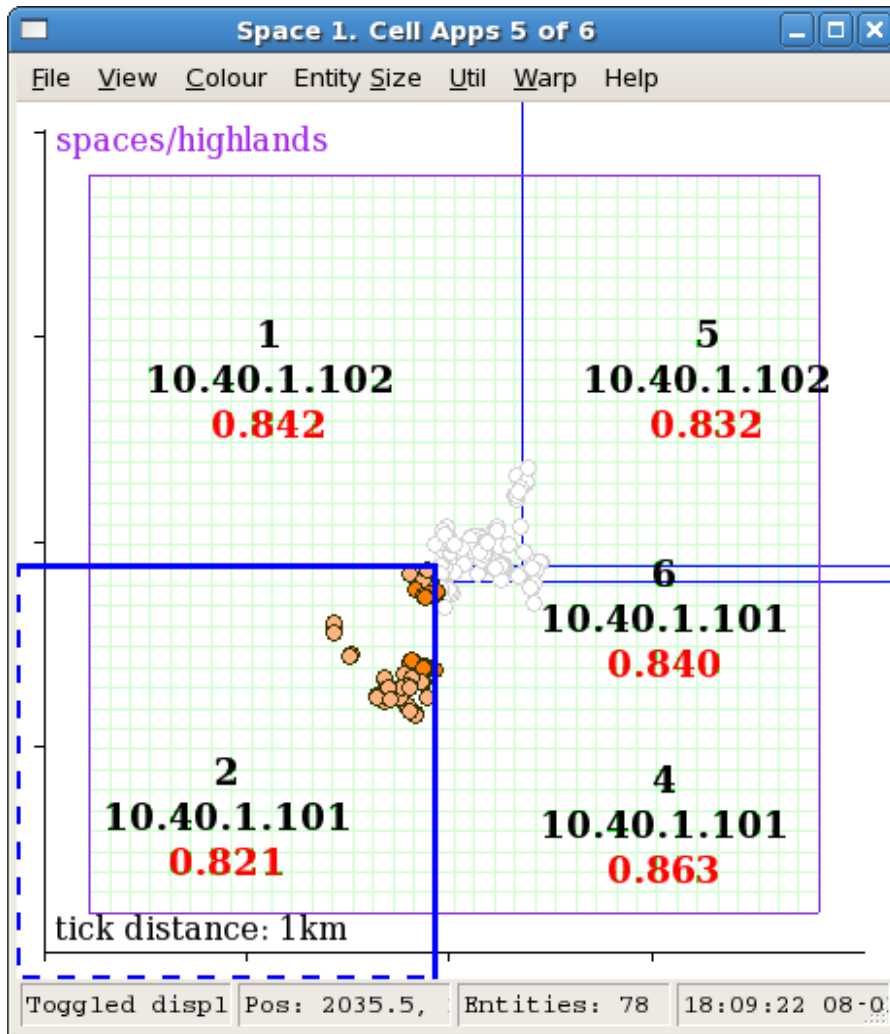
In the example above, the dominicw server has 4 spaces. To view the top down map of a space, double click its entry on the list. This will start an SVLogger process which will log data from the selected space as well as a new client window that attaches itself to the logger. Closing the client window will cause the logger to terminate.

3.5.2. Viewing Spaces

Multiple spaces can be viewed simultaneously, each in their own window. To view a space, double-click its entry on the space list displayed in Space Viewer.

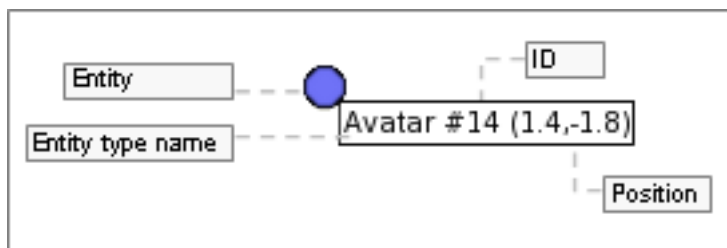
The image below shows a space being managed by five cells. Cell 2 is currently selected as can be seen by the thick blue border and coloured circles representing all the entities within the cell. Entities that are being ghosted on cell 2 are displayed as greyed out circles.

Upon first loading the space window no cell is selected for viewing. The cell boundaries and the cell information (cell number, IP address, load) will always be shown, however to see the entities within a particular cell, as with cell 2 in the screenshot below, it must be selected with a single click.



Space window

In order to facilitate identification of entities, tooltip information is available for all non-ghosted entities. This information will appear if the cursor hovers over an entity marker for a sufficiently long time.



Entity tooltip

The entity type name is retrieved from file `<res>/scripts/entities.xml`.

3.5.3. Customising Entity and Display Colours

Space Viewer allows you to customise the colours to be used when drawing the elements in the Space window. Any customisation is not saved and thus will not be in effect the next time the application is started. If you desire to permanently change the colours, this can be achieved by customising the Space Viewer source code.

Entities in the current cell are displayed with a coloured circle. The colour can be customised in the file `bigworld/tools/server/space_viewer/style.py`. The simplest way to modify the colours is to update the dictionary colours which contain RGB colour definitions for each entity type.

The code fragment below illustrates the definition of permanent entity colours:

```
# specify colour for specific entity types.
ENTITY_COLOURS = {
    "Avatar"    : ((111, 114, 247), 2.0),
    "Guard"     : ((244, 111, 247), 1.0),
    "Creature"  : ((124, 247, 111), 1.0),
    "Merchant"  : ((114, 241, 244), 1.0)
}
```

Example local file `style.py`

An alternate approach would be to replace the entire function `getColourForEntity(entityID, entityTypeID)`, using your own algorithm to colour the entities.

All other colours are defined by the dictionary `colourOptions`. This dictionary can be modified to any of the colours defined by the `colours` array.

3.5.4. Running Space Viewer Remotely

As described at the start of this section, Space Viewer's two components can be run separately if required. This would usually take the form of running SVLogger over an extended period of time, connecting clients to it to monitor live state as necessary, then eventually shutting down the logger. Later, SVLogger can be connected to its previously recorded log (instead of a live server) and client windows can be attached to it to replay the log data.

3.5.4.1. SVLogger

Location: `bigworld/tools/server/space_viewer/svlogger.py`

The SVLogger script provides log writing and reading functionality for Space Viewer. In writing mode it is responsible for gathering the space data from a running BigWorld server which is achieved by polling the CellAppMgr approximately every second. In reading mode a previously recorded log is opened for replay using any client connections.

When run without any options SVLogger will default to placing logs into `/tmp/svlog-username`. This location can be changed for keeping a more permanent archive of logs by using the `-o` option. For example:

```
$ ./svlogger -o /home/alice/space_logs
```

For a comprehensive list of options available for SVLogger please refer to the command line help available from the `--help` option.

3.5.4.2. Connecting a Space Viewer Window to SVLogger

Once you have started SVLogger manually, you will need to manually connect Space Viewer client windows to the logger if you wish to view its space data. This is easily done by running Space Viewer with the `--connect <ip:port>` option. For example:

```
$ ./space_viewer --connect 192.168.1.24:20110
```

The address passed to this option should be the address that SVLogger is listening for connections on, which should be displayed in the initial output from after SVLogger starts.

Chapter 4. Fault Tolerance

The Fault Tolerance system in BigWorld Technology is designed to transparently cope with the loss of a machine due to physical, electrical, or logical fault.

The game systems in general should not be affected, and the game experience will continue normally for most clients, with possibly a brief interruption for clients closest to the faulty machine.

The loss of any single machine (running a single process) is always handled. The loss of multiple machines in a short time frame may not always be adequately handled. In such extreme cases — for example, a complete power outage affecting all machines — the Disaster Recovery system will be invoked. For more details, see “Disaster Recovery” on page 79 .

Note that, in spite of handling failures, the Fault Tolerance system should not be relied upon to cover up a software bug that causes a component to crash. All such bugs should be found and fixed in the source code. If the bug is believed to be in BigWorld Technology code (and not in a customer's extension), then contact BigWorld Support with details of the problem. For more details, see *First Aid After a Crash* on page 119 .

4.1. CellApp Fault Tolerance

CellApp fault tolerance works by backing up the cell entities to their base entities.

As long as a backup period is specified for the cell entities, the fault tolerance for the CellApp processes is automatic. An operator should ensure that there is enough spare capacity in available CellApps to take up the load of a lost process.

For details on how to specify the CellApp backup period, see “CellApp Configuration Options” on page 30 .

Note

Cell entities without a base entity are not backed up, and therefore will not be restored if their process is lost.

Note

Since cell entities back up to base entities, running CellApp and BaseApp processes on the same machine should be avoided. If the entire machine is lost, this level of fault tolerance will not work.

For more details on the implementation of CellApp fault tolerance on code level, see the document *Server Programming Guide's* chapter *Fault Tolerance*.

4.2. BaseApp Fault Tolerance

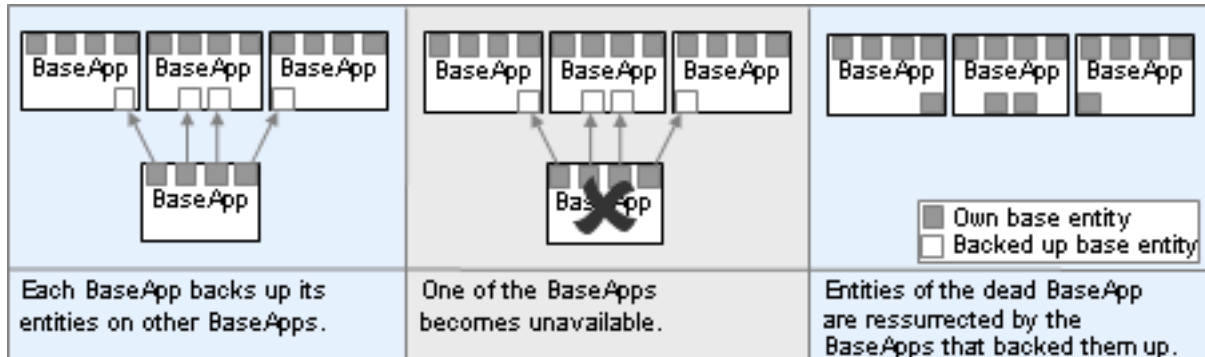
The BaseApp supports a BaseApp level fault-tolerance by backing each base entity to another BaseApp via a hashing scheme.

For more details, see the document *Server Overview's* section *Server Components* → “BaseApp” → “BaseApp”. For details on the implementation of BaseApp fault tolerance on code level, see the document *Server Programming Guide's* chapter *Fault Tolerance*.

Each BaseApp is assigned a set of other BaseApps and a hash function from an entity's ID to one of these backups. Over a period of time, all entities are backed up. This is then repeated. If a BaseApp dies, then the entities are restored on to the appropriate backup.

There is a possibility of base entities that were previously on the same BaseApp ending up on different BaseApps (which places limits on scripting).

In addition, any attached clients will be disconnected on an unexpected BaseApp failure, requiring a re-login. However, if BaseApps are retired, attached clients will be migrated to another running BaseApp.



Distributed BaseApp Backup — Dead BaseApp's entities can be restored to Backup BaseApp

4.3. Fault Tolerance with Reviver

Fault tolerance for BaseAppMgr, CellAppMgr, DBMgr, and LoginApp is provided by the Reviver, by starting a new instance of the process to replace the unavailable one.

Although it is possible for one Reviver to watch all processes, it is recommended to run a multiple Reviver instances on different machines as Revivers normally stop after reviving a process.

For more details on Reviver, see the Server Overview's chapters *Design Introduction* and *Server Components*.

4.3.1. Specifying Components to Support

When the Reviver starts, it queries the local BWMachined process, and will only support the components that have an entry in a special category called `Components` in the machine's configuration file `/etc/bwmachined.conf`.

An example `/etc/bwmachined.conf` specifying that the Reviver should support all singleton server components would contain a section as below:

```
[Components]
baseApp
baseAppMgr
cellApp
cellAppMgr
dbMgr
loginApp
```

Note

If the `[Components]` category does not contain any entries, then Reviver will support all singleton server components.

BaseApp and CellApp will not be restarted by Reviver — the `[Components]` entries are used by WebConsole and `control_cluster.py` to determine which processes should be started by BWMachined on that host.

Be aware that the configuration file is only read when BWMachineD starts. As such if you make any changes to the configuration file, you will need to restart BWMachineD before the changes are recognised. To restart BWMachineD run the following command as the root user:

```
# /etc/init.t/bwmachined2 restart
```

4.3.2. Recommended Reviver Layout

As a Reviver process is responsible for restarting a dead server process as a once off task before shutting down, a mis-configured Reviver layout may result in a situation where your cluster can suffer a process outage that is not handled by the available Revivers.

As a worst case scenario consider a server cluster that hosts the BaseAppMgr and CellAppMgr processes on a machine with the hostname `gameworld` and a single Reviver process on a backup machine with the hostname `gamereviver`. If the machine `gamereviver` has been configured to revive both the BaseAppMgr and CellAppMgr processes and `gameworld` suffers a power failure, only one process will be revived on `gamereviver`.

For this reason BigWorld recommends that you run enough Reviver instances so that each process is responsible for one singleton process. Using this approach should result in fault tolerance layout that avoids gaps in your cluster revival plan.

4.3.3. Command-Line Options

The supported components can also be specified via command-line (even though we recommend that you use `/etc/bwmachined.conf` for that), as below:

```
reviver [--add|--del {baseAppMgr|cellAppMgr|dbMgr|loginApp} ]
```

If `reviver` is invoked with no options, then it will try to monitor all singleton processes specified in category Components of `/etc/bwmachined.conf`.

The options for invoking `reviver` are described in the list below:

- **--add { baseAppMgr | cellAppMgr | dbMgr | loginApp }**

Starts the Reviver, trying to monitor only the components specified in this option. That means that the components list in `/etc/bwmachined.conf` will be ignored.

- **--del { baseAppMgr | cellAppMgr | dbMgr | loginApp }**

Starts the Reviver, trying to monitor all components specified in the list in `/etc/bwmachined.conf`, except the ones specified in this option.

- **To start Reviver trying to monitor all processes specified in Components in `/etc/bwmachined.conf`:**

```
reviver
```

- **To start Reviver trying to monitor only DBMgr and LoginApp:**

```
reviver --add dbMgr --add loginApp
```

- **To start Reviver trying to monitor all processes specified in Components in `/etc/bwmachined.conf`, except DBMgr and LoginApp:**

```
reviver -del dbMgr -del loginApp
```

Chapter 5. Backups and Disaster Recovery

BigWorld Technology's fault tolerance ensures that the server continues to operate if a single process is lost. The server also provides a second level of fault tolerance known as disaster recovery that comes into play when multiple server components fail at once and the server needs to be shut down.

For additional protection against exploits or bugs the database should be backed up regularly. The snapshot tool facilitates making backups of the database.

5.1. Disaster Recovery

The server's state can be written periodically to the database. In case the entire server fails, then it can be restarted using this information. To enable the periodic archiving of entities, an archive period needs to be set via the configuration option `<archivePeriod>` for BaseApp and CellAppMgr processes. For more details, see "BaseApp Configuration Options" on page 20, and "CellAppMgr Configuration Options" on page 38.

Disaster recovery only works when the underlying database being used is MySQL, the XML database should not be used in mission critical environments. See *MySQL Support* on page 109 for more information on how to enable MySQL database support in BigWorld.

Starting the server using recovery information from the database is the same as starting it from a controlled shutdown. For more details on controlled shutdowns see *Controlled Startup and Shutdown* on page 89.

For details on the scripting API related to disaster recovery, see the document *Server Programming Guide's* chapter *Disaster Recovery*.

5.2. Database Snapshot Tool

When using "Secondary Databases", it is necessary to snapshot all data sources at the same time to attempt to provide a cohesive data set that can be restored from. For background information on the need for the snapshot tool see *Server Programming Guide's* chapter "Database Snapshot".

The snapshot tool is a Python script that facilitates making a backup copy of the database that combines information from the primary database and secondary databases. The script itself is located in `bigworld/tools/server/snapshot/`.

When running the snapshot tool a separate machine should be used to invoke the script to avoid interfering with the normal operation of the server. The snapshot tool can use a significant amount of CPU and disk resources and does so erratically. Henceforth the machine running the snapshot tool will be referred to as the snapshot machine.

5.2.1. Operational Behaviour

The snapshot tool performs the following sequence of operations:

1. TransferDB (`bigworld/bin/Hybrid64/commands/transfer_db`) is executed on each BaseApp.
TransferDB performs a snapshot of the secondary database and sends the newly created snapshot back to the snapshot machine using Rsync.
2. TransferDB is executed on the primary database's MySQL server.
TransferDB performs an LVM snapshot of the primary database and sends the newly created snapshot back to the snapshot machine using Rsync.
3. A MySQL server is started on the snapshot machine and specifies its data directory to be the copy of the primary database.
4. Data consolidation is performed on the snapshot copies of the primary and secondary databases.

5. An archive is created of the consolidated snapshot.

The snapshot tool logs all messages to MessageLogger.

Note

A snapshot should be considered invalid if an error occurs during any phase of the snapshot procedure.

5.2.2. Usage

The snapshot tool requires a single command-line argument specifying the directory the snapshot will be written to, and may also be provided options if required. Options available to the snapshot tool are described in the `--help` as listed below:

```
Usage: snapshot.py [[options]] SNAPSHOT_DIR

Options:
  -h, --help            show this help message and exit
  -b BWLIMIT_KBPS, --bwlimit-kbps=BLIMIT_KBPS
                        file transfer bandwidth limit in kbps, default is
                        unlimited
  -n, --no-consolidate  skip consolidation, default is false
```

The list below provides some common examples of using the snapshot tool:

- **snapshot.py /home/bwtools/snapshots**

Takes a snapshot. The snapshot is archived to /home/bwtools/snapshots.

- **snapshot.py /home/bwtools/snapshots --bwlimit-kbps=5000**

Takes a snapshot. The snapshot is archived to /home/bwtools/snapshots. Each secondary and primary database is transferred with a 5000 kbps bandwidth limit.

- **snapshot.py /home/bwtools/snapshots --no-consolidate**

Takes a snapshot. The unconsolidated snapshot is archived to /home/bwtools/snapshots.

- **snapshot.py /home/bwtools/snapshots -n -b5000**

Takes a snapshot. The unconsolidated snapshot is archived to /home/bwtools/snapshots. Each secondary and primary database is transferred with a 5000 kbps bandwidth limit.

5.2.3. Requirements

For the snapshot tool to work, the following conditions must be met:

- Secondary databases must be enabled.

To enable secondary databases set the `baseApp/secondaryDB/enable` property to `true`. See “Secondary Database Configuration Options” on page 24 for details.

- The primary database is stored on an LVM volume and there is sufficient unpartitioned space on the primary database machine to perform an LVM snapshot. Note that, for the purposes of database snapshotting, no other cluster machine is required to have spare unpartitioned space on a LVM partition. See also “Partitioning” on page 82.

- BWMachined must be running on the primary database machine.

For background information see Server Overview's chapter *Server Components* → “BWMachined”.

- All the entity scripts and entity definition files must be present and locatable on the snapshot machine.

In order for the resource tree to be found the BWMachined configuration file (`.bwmachined.conf`) must be configured correctly for the user running the snapshot tool.

- The snapshot section in `/etc/bigworld.conf` must be configured correctly on the primary database machine.

See “Configuration” on page 83 .

- The directory the LVM volume is to be mounted to exists and has no other devices mounted to it.
- The user performing the snapshot must be able to connect to the snapshot machine using **ssh** from the primary database machine and BaseApp machines without a password.

For a user whose `$HOME` directory is shared between machines (e.g, using NFS) this can be done using the following commands:

```
$ ssh-keygen -t dsa
$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```

- The LVM userland program suite must be installed and accessible by the snapshot user on the primary database machine.

The LVM userland applications can be installed under CentOS using the command:

```
# yum install lvm2
```

By default the tools are installed into `/usr/sbin`, so it might be necessary to modify the snapshot user's environment settings to at this directory to the `$PATH`.

You can do this by adding to the `PATH` environment variable in the `$(HOME)/.bash_profile` file, by adding a line such as this:

```
export PATH=$PATH:/usr/sbin
```

You will need to logout and log back in for this change to take effect.

- The following standard system applications must also be accessible by the snapshot user. **mount**, **umount** and **chmod**.
- MySQL server must be installed on the snapshot machine. Specifically **mysqld_multi** must be accessible.

To install the MySQL server binaries, run the following command as root:

```
# yum install mysql-server
```

- Rsync must be installed and be on the path of all machines being used, i.e, the snapshot machine, the primary database machine as well as all the BaseApp machines.

To install Rsync run the following command as root:

```
# yum install rsync
```

- The SnapshotHelper utility must be built.

To build the SnapshotHelper binary, MySQL development files must be installed. See “Compiling DBMgr with MySQL Support” on page 109 for details on how to install MySQL development files.

After installing MySQL development files, issue the **make** command from within the `bigworld/src/server/tools/snapshot_helper/` directory.

```
$ cd bigworld/src/server/tools/snapshot_helper
$ make
```

- The SnapshotHelper binary must have its `setuid` attribute set so user ID is root upon execution on the primary database machine.

To make the SnapshotHelper binary `setuid` root run the following commands as the root user:

```
# chown root:root
$MF_ROOT/bigworld/tools/server/bin/Hybrid[64]/snapshot_helper
# chmod 4511 $MF_ROOT/bigworld/tools/server/bin/Hybrid[64]/snapshot_helper
```

Note

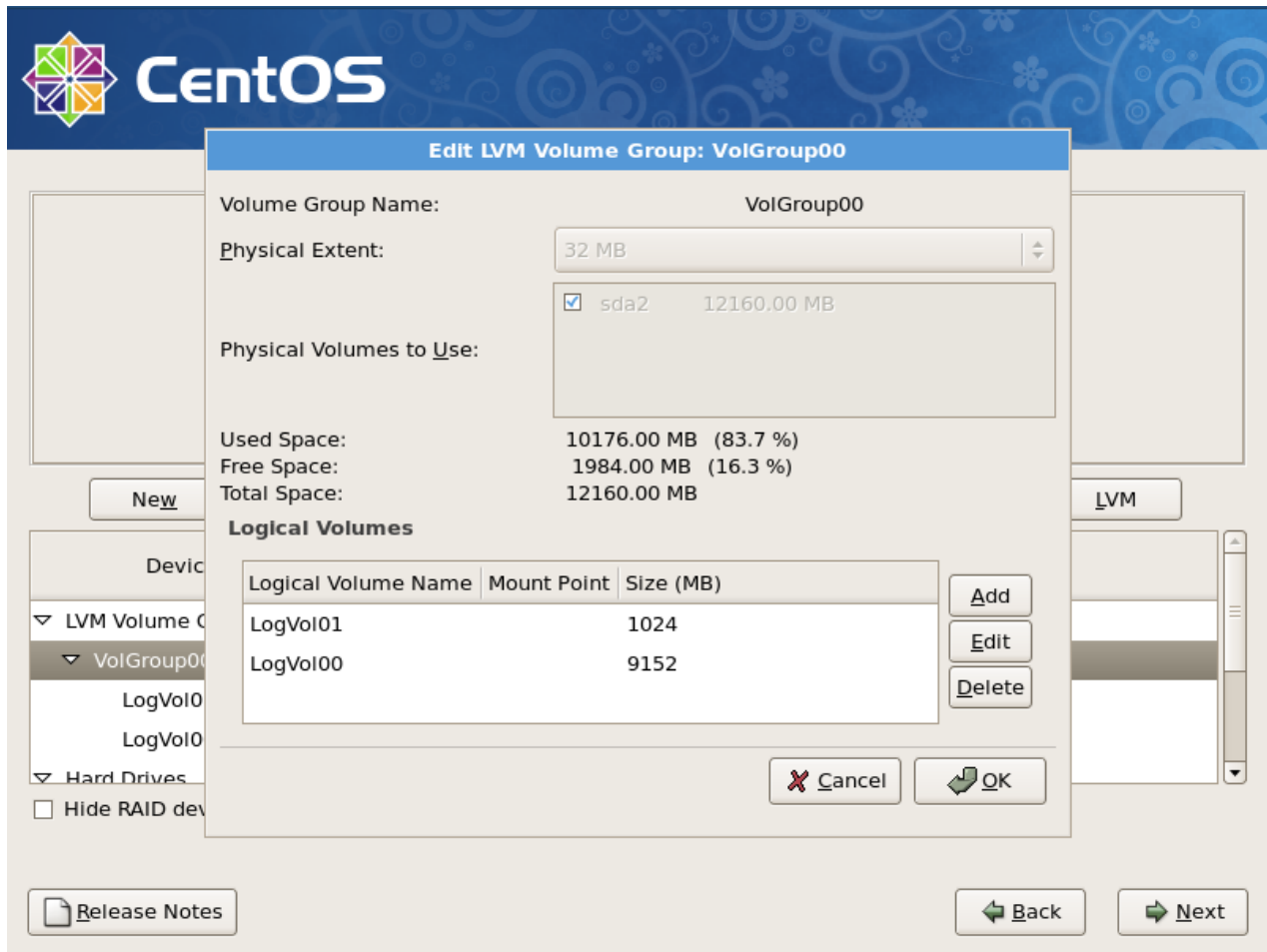
For security purposes all privileged commands (**lvcreate**, **lvremove**, **mount**, **unmount** and **chmod**) are invoked via **snapshot_helper** which reads command arguments from the trusted file `/etc/bigworld.conf`.

5.2.4. Partitioning

The snapshot tool requires that you have unallocated space on the LVM logical disk that your MySQL database resides on. As a rule-of-thumb, we recommend that the size of the unallocated space be 15-20% of the overall volume's size. Having too little space will cause the snapshot tool to fail.

Unallocated space can be added when partitioning the disks of a machine hosting MySQL. The steps below illustrate how to add a region of unallocated space for snapshotting, in addition to a swap partition and a file system partition.

- In the disk partitioning step, choose `Create custom layout`.
- Select the LVM group `VolGroup00` that is created by default, and click on the `Edit` button.
- Select each of the `LogVol100` and `LogVol101`, and delete them. You may wish to take note of the size of the swap partition (`LogVol101`), and reuse this size when recreating the swap partition.
- Add a new logical volume by clicking on the `Add` button. The mount point should be `/`, and we recommend you leave the default filesystem as `ext3`. The size should be 80-85% of the overall size of the volume group, less the space allocated for swap. In the example illustrated in the image below, we have allocated split a 12160MB drive into 1984MB to be unallocated, 1024MB for swap, and 9152MB for the `/` mount which will hold the file system.



LVM partitioning setup

5.2.5. Configuration

The snapshot tool behaviour can be configured using the file `/etc/bigworld.conf`.

Options for the snapshot tool must be placed within a `[snapshot]` section. Below is a list of valid keywords that can be used in the configuration file:

- **datadir**

The primary database's data directory. By default for CentOS distributions this is `/var/lib/mysql`.

This should be the same value as found in the MySQL configuration file `/etc/my.cnf`.

```
$ cat /etc/my.cnf
[mysqld]
datadir=/var/lib/mysql
...
```

- **lvgroup**

The name of the LVM volume group in which the primary database belongs to.

Note

The LVM snapshot that is created will belong to this LVM group.

- **lvorigin**

The name of the origin LVM volume snapshot. The MySQL data directory should be on this volume.

- **lvsnapshot**

The name of the LVM snapshot volume that will be created and deleted by the snapshot tool.

For example, if *lvgroup* is *VolGroup00* and *lvsnapshot* is *ExampleSnapshotVolume*, a new volume device will be created called */dev/VolGroup00/ExampleSnapshotVolume*.

The snapshot name is also used to mount the snapshot volume device onto the active filesystem. Using the previous example the following mount command will be issued:

```
mount /dev/VolGroup00/ExampleSnapshotVolume /mnt/ExampleSnapshotVolume
```

Note

The mount point */mnt/lvsnapshot* is assumed to already exist on the system prior to snapshots being created.

- **lvsizegb**

The size of the LVM snapshot volume in gigabytes. The snapshot does not need the same amount of storage the origin has as only the modified files are copied, in a typical scenario, 15-20% might be enough. If the LVM snapshot volume becomes full it will be dropped, so it is important to allocate enough space.

Below is an example */etc/bigworld.conf* configuration file:

```
[snapshot]
datadir = /var/lib/mysql
lvgroup = VolGroup00
lvorigin = LogVol100
lvsnapshot = bwsnapshot
lvsizegb = 2
```

Note: The snapshot tool also reads *<res>/server/bw.xml* for the *dbMgr/host*, *dbMgr/username*, *dbMgr/password* and *dbMgr/databaseName*.

5.2.6. Restoring From a Snapshot

The snapshot tool will take a copy of a MySQL data directory consolidated with a copy of the secondary databases. To restore from a snapshot MySQL data directory, perform the following steps on the machine hosting MySQL:

1. Stop the BigWorld server if it is running.
2. Stop the MySQL server if it is running, you can do this by running as root:


```
# /etc/init.d/mysqld stop
```

3. Move the existing `/var/lib/mysql` to a temporary location. You may choose to remove it after the restoration process is complete. For example:

```
# mv /var/lib/mysql /tmp/mysql-backup
```

4. Copy the snapshot directory's `mysql` directory to `/var/lib/mysql` (you will generally need to be root to do this) Note that snapshots are identified by date and time of the snapshot, and that the colons used in the time need to be escaped by a backslash. For example:

```
# cp -r snapshots/20100924_152642/mysql /var/lib
```

5. Change the ownership and permissions:

```
# chown -R mysql:mysql /var/lib/mysql
# find /var/lib/mysql -type d | xargs chmod 700
# find /var/lib/mysql -type f | xargs chmod 660
```

6. Restore the SELinux contexts for these files:

```
# /sbin/restorecon -r /var/lib/mysql
```

7. Start the MySQL server, verify it is running.

```
# /etc/init.d/mysqld start
```

8. Start the BigWorld server.

5.3. Data Consolidation Tool

The data consolidation tool is used to incorporate data from secondary databases into the primary database. The application binary is located in `bigworld/bin/Hybrid[64]/commands/consolidate_dbs`. This tool is only provided in source code form and must be built before use. See “Enabling Secondary Databases” on page 110 for details on how to build the data consolidation tool.

Once this tool has been compiled, providing DBMgr is using MySQL the data consolidation tool will be automatically run during system shutdown or during system startup if the server was not shutdown correctly. There should be no need to manually run this tool except for consolidating backups created by the snapshot tool. For details on the snapshot tool see “Database Snapshot Tool” on page 79. The data consolidation tool sends log messages to MessageLogger which can be viewed using the LogViewer tool in WebConsole.

This tool accepts two command line arguments:

- **Primary Database**

The parameters required to connect to the primary database.

This argument should be specified in the form `host;username;password;database_name`.

Note

When specifying this argument on the command line, it is necessary to quote the entire argument in order to prevent the shell from treating it as multiple commands.

For example:

```
./consolidate_dbs
"qa_host_03;bigworld_user;bigworld_password;fantasydemo"
```

▪ Secondary Databases

A file containing a newline separated list of fully qualified paths to secondary database files.

When run without command line arguments, it will use DBMgr's configuration options located in `<res>/server/bw.xml` to connect to the primary database. It will use the data in the `bigworldSecondaryDatabase` table to retrieve the secondary databases.

The data consolidation tool uses the **transfer_db** utility located in `bigworld/bin/Hybrid[64]/` command to transfer the secondary database files from the BaseApp machines to the machine where the data consolidation tool is running. The **transfer_db** utility will be launched (via **bwmachined**) on each machine that contains a secondary database file. The **transfer_db** utility then opens a TCP connection to the data consolidation tool through which it will send the secondary database file.

The data consolidation tool will store the secondary databases locally in the directory specified by the `<dbMgr>/<consolidation>/<directory>` configuration option. It will incorporate the data from the secondary databases into the primary database once all the secondary database files have been transferred to its local machine. If the data consolidation process is successful, both the local and remote copies of the secondary databases will be deleted, and the `bigworldSecondaryDatabase` table will be cleared. If the data consolidation process is unsuccessful, only the local copy of secondary database files are deleted.

The data consolidation tool logs to Message Logger under the `ConsolidatedDBs` process. Under most circumstances, the data consolidation tool will log errors from **transfer_db** as well. However, under some circumstances, **transfer_db** errors will appear under the `Tools` process. The **bwmachined** logs, in `/var/log/messages` on each machine where secondary databases are located, can help to diagnose problems, especially those related to the failure to run **transfer_db**.

5.3.1. Skipping Data Consolidation

If, for some reason, there is a problem with the data consolidation tool and the server refuses to start, the `bigworldSecondaryDatabase` table can be manually cleared to skip the data consolidation process. Alternatively, **consolidate_dbs** can be run with the command-line parameter `--clear` to achieve the same result.

While skipping the data consolidation process may allow the server to start again, the data in the secondary databases that were not consolidated is essentially lost. Though the secondary database files still exist, it is not recommended to try to consolidate the data after the server has been restarted since the data in the primary database may be more recent than the data in the left over secondary databases.

5.3.2. Ignoring SQLite Errors

When **consolidate_dbs** encounters an error in reading a secondary database file, it will abort the entire data consolidation process. Such errors should be investigated and corrected. If a secondary database file is genuinely corrupted and cannot be repaired, then it may be preferable for **consolidate_dbs** to read as

much data as possible from the corrupted secondary database and then proceed to consolidate the rest of the secondary databases. When the `--ignore-sqlite-errors` command-line option is specified, **consolidate_dbs** will proceed to consolidating the next secondary database when it encounters an error instead of aborting the consolidation process.

Chapter 6. Controlled Startup and Shutdown

For maintenance and upgrade reasons, the server or some part of it may be required to shutdown. There are two typical scenarios:

1. Server-wide shutdown, where the entire server needs to be restarted.
2. Individual application retirement that does not cause the server to stop running.

6.1. Server-wide Shutdown

There are times when the entire server might need to be shut down, and restarted later in a similar state.

To tell the server to shut down in a controlled way, a message must be sent to all LoginApps. This may be in the form of a Watcher message or a USR1 signal. The easiest way to do this is to use the script `control_cluster.py` with the option `stop` or via WebConsole. For more details, see “Control Cluster” on page 67.

Controlled startup and shutdown only work when the underlying database is MySQL — it does not work when the XML version is used.

If the `bw.xml`'s `dbMgr/clearRecoveryData` configuration option is set to `false`, then the server is automatically started using information written during the controlled shutdown. If the server had failed unexpectedly, then it is started using the disaster recovery information. Setting that configuration option to `true` will cause the server to be started in its initial state, *i.e.*, an empty space with no entities. For more details, see “DBMgr Configuration Options” on page 41.

The main information that is restored from the database is:

- Spaces and their data
- Game time
- Which entities should be in each space.

When using MySQL as the underlying database, this information is stored in the following tables:

- Spaces and their data — `bigworldSpaces`, `bigworldSpaceData`
- Game time — `bigworldGameTime`
- Which entities should be in each space — `bigworldLogOns`

For details on these tables, see the document *Server Programming Guide*'s section *MySQL Database Schema* → “Non-Entity Tables”

For details on related scripting, see the document *Server Programming Guide*'s chapter *Controlled Startup and Shutdown*.

6.2. Individual Application Retirement

Individual BaseApps and CellApps can be retired. This can be useful if maintenance is required on a single machine where only BaseApps and CellApps are running.

6.2.1. BaseApp Retirement

When BaseApps retire, base entities and proxy entities are offloaded to other BaseApps, and connected clients will be reconnected with another BaseApp transparently.

It may take some time for the base entities and proxy entities to be offloaded from the retiring BaseApp to other BaseApps. The retiring BaseApp will shutdown once these entities have been successfully offloaded and a new backup cycle has completed, ensuring no data loss.

If another BaseApp terminates unexpectedly while the retirement is in progress, any entities that were offloaded to the dead BaseApp will be re-offloaded on to another BaseApp.

For redundancy reasons, it is recommended that retirement only be done if there are at least two other BaseApps running. This ensures that in the event of a unexpected BaseApp termination while retirement occurs, every base entity in the system is adequately backed up.

6.2.2. CellApp Retirement

CellApps can also be retired. Each cell administered by the retiring CellApp will shrink in size gradually, each cell disappearing once the area reaches zero. Once there are no more cells, the CellApp will shut itself down.

6.2.3. Retirement via WebConsole

A BaseApp or a CellApp can be individually retired by using the WebConsole in the ClusterControl module, by selecting *Retire App* from the action menu for the particular CellApp or BaseApp to be retired. Refer to the section on “WebConsole” on page 51 .

6.2.4. Retirement via `control_cluster.py`

A BaseApp or a CellApp can be individually retired by using the ControlCluster command line tool (see “Server Command-Line Utilities” on page 67), using the **retireproc** command. For example:

```
$ ./control_cluster.py retireproc cellapp01
$ ./control_cluster.py retireproc baseapp03
```

Chapter 7. Stress Testing with Bots

The Bots application is a process that can command arbitrary numbers of simulated clients to log into the server and perform activities. It is written in C++, but controlled through Python, like the BigWorld Client.

7.1. The Login Process

The bots are designed to log in a server just like a real client. By default, the Bots process locates a LoginApp by broadcasting a message requesting all LoginApps to respond, and then using the first one to reply. This is the simplest method to log in if there is only one instance of a BigWorld server running.

The script is located in folder `bigworld/bin/$MF_CONFIG`, and has the following syntax:

```
bots [-serverName <srv> [-port <port>]]  
      [-username <user> [-password <pwd>] [-randomName]] [-scripts]
```

The options for invoking bots are described in the list below:

- **-serverName <srv>**

Server that the bot clients will login to (overrides the auto-locate method).

- **-port <port>**

Login port. Ignored if Bots is using auto-locate method.

- **-username <user>**

Username that the bot clients will log in as.

- **-password <pwd>**

Password for bot clients' login.

- **-randomName**

Randomises the login name based on the specified <username>.

- **-scripts**

Enables Bots to run scripts. This has a considerable performance cost.

The auto-locate procedure can be overridden by specifying the option `-serverName <serverName>` on the command line. If both methods fail, then the Bots process requests the server name on stdin (if attached to a terminal). If all these methods fail, then the process starts, but cannot create new bots until the server is set (via the Python method `setDefaultServer`).

If the option `-scripts` is specified then Bots looks for the entity definition files under folder `<res>/scripts/entity_defs`, and for scripts under folder `<res>/scripts/bot`.

Specific scripts can be created for Bots, but if a client script does not reference the client's C++ modules, then it can be reused. Note that the bots process does not support the BigWorld Client APIs, since it is intended to be lightweight.

In many operation scenarios the entity definition file for Bots could be either missing or different from the definition files for the server. The Bots would use an incorrect message digest for login; and hence the login

will fail because LoginApp relies on the digest for verifying entity definition consistency. Previously, only way around this problem is turn the digest checking on the server off in `<res>/server/bw.xml`'s dbMgr options section¹. This option can also be changed on the fly via a watcher, using WebConsole's ClusterControl module; see “ClusterControl” on page 51). However, this is not a viable option, if you want to use Bots on production servers. You can now set custom MD5 digest in the `<res>/server/bw.xml`'s bots section under option loginMD5Digest. This option can also be modified on the fly through WebConsole.

Bots can now realistically simulate general Internet networking environment by introduce artificial packet loss, network delays, sudden disconnections and frequent relogins. All this simulation can be programmed with Bots' personality script.

7.2. Python Interface

The Bots process was designed to be controlled programmatically via Python, but since Python can be used interactively, it is also a quick way to get some bots into the system and control them.

You can telnet to port 6075 to talk to the bots process, as illustrated below:

```
$ telnet 10.40.7.12 6075
Welcome to the Bot process
>>> BigWorld.getDefaultServer()
10.40.7.1
>>> BigWorld.setDefaultServer('server2')
>>> BigWorld.setDefaultTag('red')
>>> BigWorld.addBotsSlowly(500, 0.1)
>>> BigWorld.setDefaultTag('blue')
>>> BigWorld.addBotsSlowly(500, 0.1)
>>> BigWorld.addBotsWithName( [('Bot_01', '01'), ('Bot_02', '02')] )
```

Example of use of Python interface to Bots

7.2.1. Python Controller (bot_op.py)

BigWorld provides a Python program to simplify the management of large numbers of bots.

It automatically starts bot processes as needed, load balancing (based on CPU load) on all available machines not running any other BigWorld components.

The script is located in folder `bigworld/tools/server` and has the following syntax:

```
python bot_op.py [add [<number_of_bots>]]
                  [del [<number_of_bots>]]
                  [movement <controller_type> <controller_data> [<bot_tag>]]
                  [set (<watcher_name> <value>)+]
                  [run [<command>]]
                  [addprocs [<num_of_procs>]]
                  -u [username or uid]
```

The options for invoking `bot_op.py` are described in the list below:

- **add [<number_of_bots>]**

Adds the specified number of bots to the system. If `<number_of_bots>` is not specified, then one bots is added.

¹For details check `allowEmptyDigest` option in “DBMgr Configuration Options” on page 41 .

Bots are added 16 at a time, up to a maximum of 256 per bots process.

The script waits 1 second after each add, to check if a bot was effectively added. When all bots processes are full, it creates a new process on an available machine that is using less than 80% CPU (preferably one that is already running some bots processes).

- **addprocs** [**<num_of_procs>**]

Spawns the given number of bots processes on each machine that is considered eligible for running them, or 1 on each if no number is given.

- **del** [**<number_of_bots>**]

Deletes the specified number of bots from the system. If **<number_of_bots>** is not specified, then it defaults to 1. The number of bots will be ceiling clamped to the actual number of bots on the server.

- **movement** **<controller_type>** **<controller_data>** [**<bot_tag>**]

Changes the bot's movement controller. If option **<bot_tag>** is specified, then only bots matching it will be changed. Otherwise, all bots will be changed.

- **set** (**<watcher_name>** **<value>**)+

Sets the watcher to specified value on all bot applications.

- **run** [**<command>**]

Runs the specified Python command on all bots. If **<command>** is not specified, then it is read from stdin.

- **-u** [**username|uid**]

Manually specify the UID of the server that the script will take action on.

7.2.2. Methods and Attributes

The list below describes the bots' attributes and methods on ClientApp and BigWorld:

- **ClientApp Attributes**

- **id (int)** — Bot ID (read-only).
- **spaceID(int)** — ID of the space the bot is currently in. 0 means not in any space (read-only).
- **loginName(string)** — Bot's login name.
- **loginPassword(string)** — Bot's login password.
- **tag (string)** — Bot's user-specified tag. Allows the control of partial sets of bots.
- **speed (float)** — Bot's speed.
- **position (Vector3)** — Bot's position.
- **yaw (float)** — Bot's yaw.
- **pitch (float)** — Bot's pitch.
- **roll (float)** — Bot's roll.
- **isOnline(bool)** — Indicates whether the bot is connected to a server (read-only).

- **isDestroyed(bool)** — Indicates whether the bot has been destroyed (read-only).
- **entities(PyObject)**

Dictionary-like attribute containing the list of entities that are in the simulated client's current AOI.

- **ClientApp Methods**

- **logOn()**

Initiate log on process for the simulated client to connect to a BW server.

- **logOff()**

Make a simulate client disconnect from server gracefully. If the simulated client is not online, it will do nothing.

- **dropConnection()**

Make a simulate client drop the connection with the server. If the simulated client is not online, it will do nothing.

- **setConnectionLossRatio(float lossRatio)**

Set up network packet loss ratio for simulating unstable network environment. Range between 0.0 and 1.0. 0.0 means no packet loss. 1.0 mean 100 percent packet loss.

- **setConnectionLatency(float minLatency, float maxLatency)**

Set up simulated network data latency (in milliseconds).

- **moveTo(Math.Vector3 position)**

Set next destination position for the player avatar of the simulated client.

- **snapTo(Math.Vector3 position)**

Set the player avatar position for the simulated client.

- **stop()**

Stop the player avatar of the simulated client moving.

- **faceTowards(Math.Vector3 direction)**

Set the direction of avatar of the simulated client.

- **addTimer()**

Add a timer for the bot client. This function returns an integer timer id; otherwise it returns -1.

- **delTimer(int timerID)**

deletes an active timer corresponding the timerID.

- **onTick()**

This optional function is invoked every game tick, should it be defined in the player Avatar script.

- **BigWorld Attributes**

- **bots**

Dictionary-like attribute containing the list of simulated clients. The key is the player's ID, and value is an instance of ClientApp.

- **bots [<player_id>]. entities**

Dictionary-like attribute containing the list of entities created under the simulated client specified by <player_id>. The key is the entity's ID, and the value is the reference to the entity.

In non-script bots, only the player entity is created and stored in this attribute.

- **bots [<player_id>].entities[<ent_id>].clientApp**

Attribute referring to the ClientApp that owns the entity. An entity can access its owner ClientApp via this attribute. `BigWorld.bots[<player_id>].entities[<ent_id>].clientApp` equals to `BigWorld.bots[<player_id>]`.

- **BigWorld Methods**

- **setMovementController(string type, string data) — Context: ClientApp**

Sets a new movement controller for the bot. On failure, the controller is left unchanged. Returns `true` on success, `false` on failure.

- **setLoginMD5Digest(string MD5Digest) — Context: BigWorld**

Set the 32 character long MD5Digest (in hex readable form) for server login. If the input string length is not exactly 32 character, the MD5 digest will be reset to empty.

- **addBots(int numBots) — Context: BigWorld**

Immediately adds the specified number of simulated clients to the application. If too many bots are added at once, then some of them may time out, depending on how long it takes to log in all of them.

- **addBotsSlowly(int numBots,float delayBetweenAdd,int groupSize=1) — Context: BigWorld**

Adds bots to the system in groups of groupSize, and add a delay between each group. This method returns immediately, adding the bots in the background, and is the best way to add bots to a system.

- **addBotsWithName(PyObject loginInfo)**

Immediately add a set of bots with login name and login password specified from loginInfo. The structure of the loginInfo is a list of tuples which consist of two strings. The first string is the login name and the second one is login password. In the event of incorrect loginInfo, a Python exception will be thrown.

- **delBots(int numBots) — Context: BigWorld**

Deletes the specified number of simulated clients from the application. The method `onClientAppDestroy()` of the personality module is called.

- **getDefaultServer() — Context: BigWorld**

Returns the server name that the bots try to log in.

- **setDefaultServer(string serverName) — Context: BigWorld**

Sets the server name that the bots try to log in.

- **onTick()** — Context: BWPersonality

This optional function is invoked every game tick, should it be defined in the personality module in `<res>/server/bw.xml`.

- **onLoseConnection(int playerID)** — Context: BWPersonality

If this callback function is defined in the personality module specified in file `<res>/server/bw.xml`, it will be invoked when a simulated client loses its connection with a server abnormally. The personality script can decide whether the client should be destroyed or remain dormant for later reactivation by return True or False respectively. **NOTE:** if this callback is not defined, bots that lose connection will be automatically destroyed.

- **onClientAppDestroy(int playerID)** — Context: BWPersonality

If defined in the personality module specified in file `<res>/server/bw.xml`, this callback method is called for each destroyed bot client. The argument is the ID of the deleted player entity. A bot client may be destroyed due to either loss of server connection or by `BigWorld.delBots()` method call. If a bot client is destroyed by `BigWorld.delBots()` method call, it will be disconnected from the server gracefully. We recommend client should execute additional log off procedure in this `onClientAppDestroy` callback function. An example is as following:

```
def onClientAppDestroyed( playerID ):
    bot = BigWorld.bots[playerID]
    if bot.isOnline:
        bot.entities[playerID].base.logOff()
```

The implementation above assumes that the player entity on the server has an exposed base method `logOff()`, which will then destroy the player cell and base entities on the server. The script must to be placed under the script folder `<res>/scripts/bot`.

7.3. Controlling Movement

The default movement controller Patrol moves the bots along a graph read from a file.

The name of the file containing the movement graph is specified in file `<res>/server/bw.xml`, on section `<bots>`, by the configuration option `controllerData`.

```
<root>
...
<bots>
  <controllerData>  server/bots/test.bwp  </controllerData>
  <controllerType>  Patrol                </controllerType>
  <password>        passwd                </password>
  <username>        Bot                  </username>
  <shouldLog>       true                  </shouldLog>
  <serverName>      </serverName>
  <randomName>      false                 </randomName>
  <scripts>         false                 </scripts>
  <port>            0                     </port>
  <publicKey>       loginapp.pubkey       </publicKey>
</bots>
...
</root>
```

Example file `<res>/server/bw.xml`

The file name might have the suffix random, in which case the start position of the bot is chosen randomly from the set of node positions in the graph.

The format specification for the movement graph file is illustrated below:

```
<controller>

  <nodeDefaults>
    ?NodeProperties
  </nodeDefaults>

  <nodes>
    *<node>
      <pos>    integer integer integer  </pos>
      ?<name>  string_node_name        </name>
      ?NodeProperties
      *<edges>
        <edge> string_node_name  </edge>
      </edges>
    </node>
  </nodes>

</controller>
```

Grammar of movement graph files

The list below describes the tags in the movement graph file:

- **edge (section nodes/node/edges)**

Name of one of the nodes that this node borders. Must be the same value specified for a name section in the file.

- **edges (section nodes/node)**

Tag specifying all the nodes that this node borders.

- **name (section nodes/node)**

Name used to refer to the node. If this tag is not specified for a node, then you can refer to it by an integer sequential number, with the first node having an index of zero.

- **node (section nodes)**

Tag for section specifying a specific node in the graph.

- **nodeDefaults**

Tag for section specifying default values for all nodes. These values can be overridden on a per-node basis in section nodes/node.

- **NodeProperties (section NodeDefaults and nodes/node)**

For details, see “NodeProperties Section” on page 98 .

- **nodes**

Tag for section specifying all the nodes in the graph.

- **pos (section nodes/node)**

XYZ position of the centre of the node. The node is the area around pos, extending for n metres around it (n being the value of NodeProperties' tag radius).

7.3.1. NodeProperties Section

The format specification for the NodeProperties section is illustrated below:

```
?<minStay>    float    </minStay>
?<maxStay>    float    </maxStay>
?<radius>     float    </radius>
?<minSpeed>   float    </minSpeed>
?<maxSpeed>   float    </maxSpeed>
```

Grammar of **NodeProperties** section

The list below describes the tags in the NodeProperties section:

- **maxSpeed**

Maximum speed of the bot.

- **maxStay**

Maximum number of seconds that the bot should stay in the node.

- **minSpeed**

Minimum speed of the bot.

- **minStay**

Minimum number of seconds that the bot should stay in the node.

- **radius**

Number of metres around the node's pos location to be considered as the node. When a bot is travelling between two nodes, it chooses a random point of arrival in the destination node. This way, the bots do not follow the exact same line. The point is chosen to lie within the radius of the destination node.

An example movement graph file is displayed below:

```
<patrolGraph>
  <nodeDefaults>
    <minStay>    5    </minStay>
    <maxStay>    10   </maxStay>
    <radius>     20   </radius>
  </nodeDefaults>
  <nodes>
    <node>
      <name>      Town1      </name>
      <minStay>   7          </minStay>
      <pos>       5000 0 0    </pos>
      <edges><edge> Town2      </edge></edges>
    </node>
    <node>
      <name>      Town2      </name>
      <pos>       5000 0 5000 </pos>
    </node>
```

```
</nodes>
</patrolGraph>
```

Example movement graph file

7.4. Extending Bots

7.4.1. Creating New Movement Controllers

The default movement controller is Patrol, but you might want to create one that better represents the expected traffic patterns and entity distributions of your game, or to stress test other aspects of the system.

To create a new movement Controller, derive its class from the MovementController class. Also derive a class from the MovementFactory class. The factory is used to parse the Controller data argument, and instantiate a new movement controller, while the movement controller moves the bot around.

The fragments below list the relevant methods in the base classes:

- **Class MovementController**

```
bool nextStep( float speed, float dTime, Vector3& position, Direction3D&
               direction)
```

- **Class MovementFactory**

```
MovementFactory(const char* name)

MovementController* create(
    const std::string& data,
    const Vector3&      startPosition)
```

7.5. Miscellaneous Bots Issues

7.5.1. Running out of File Descriptors

As the Bots process is designed to test a large number of client connections to a BigWorld server, it is possible that in doing so the Bots process may exceed the maximum number of open file descriptors allowed by the Linux kernel.

This issue will be identifiable by newly created bots not communicating with the LoginApp, as well as log output from the Bots process similar to the following:

```
Bots ERROR Endpoint::findDefaultInterface: if_nameindex returned NULL (Too
many open files)
Bots ERROR NetworkInterface::recreateListeningSocket: Couldn't determine
ip addr of default interface
```

If you encounter this issue, this can be worked around by changing your Linux system settings to increase the per-process limit for the maximum number of open file descriptors

To check the current limit allowed by your system, from a Linux shell type the following:

```
$ ulimit -n
```

```
1024
```

Here you can see the limit is 1024 open file descriptors per-process. In order to increase this limit for all processes we need to edit the file `/etc/security/limits.conf`. We do this on a per-user basis, so if we wanted to increase the limit for the `game_user` user account to 4096, we would add the following:

```
game_user soft nofile 4096  
game_user hard nofile 4096
```

If the user is currently logged in, you may need to log out and then back in for the changes to take effect on the sessions.

Chapter 8. Security

Security has been of paramount importance in the design and implementation of all parts of BigWorld.

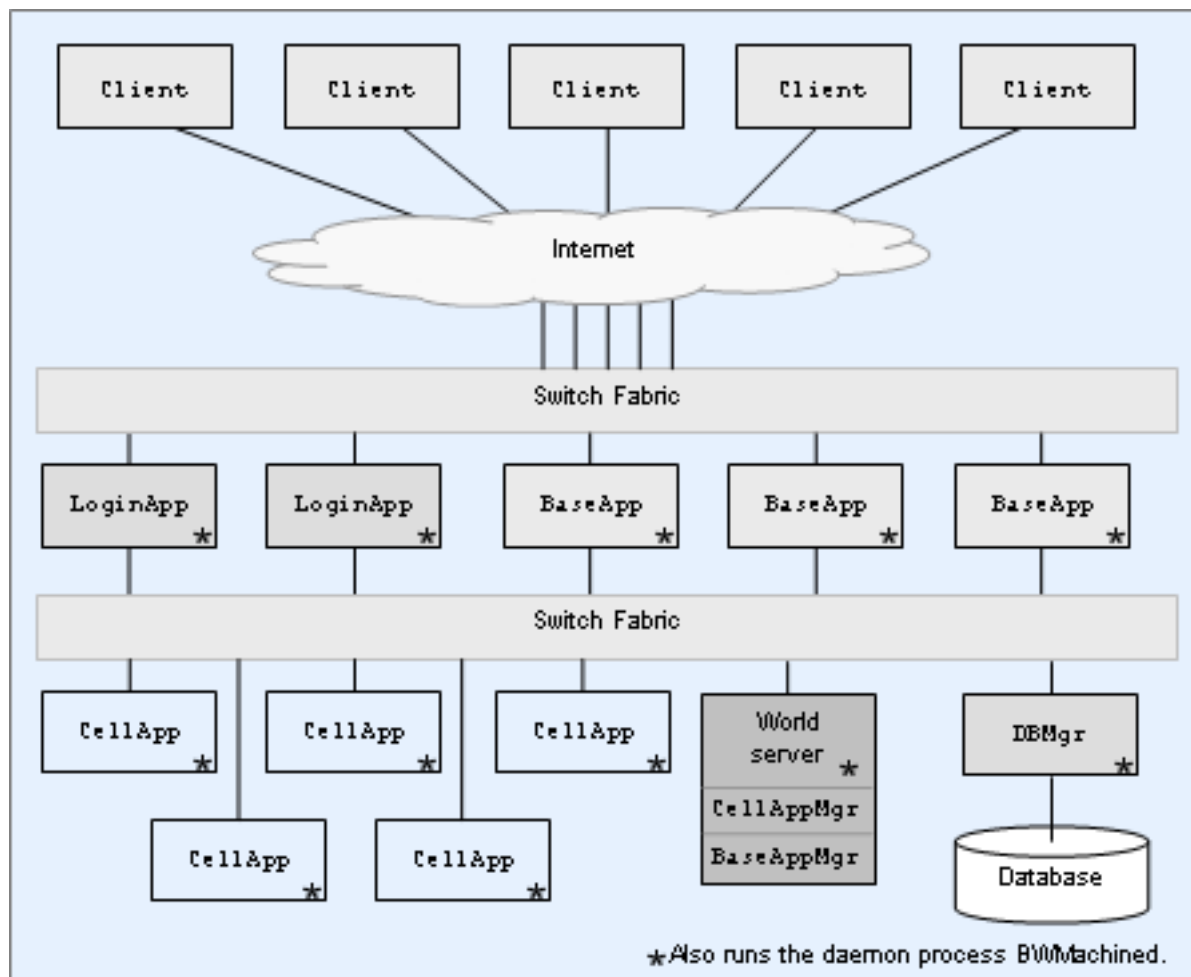
The basic philosophy is to always handle with care any client-initiated actions or messages. This should be accomplished in a way that does not unduly limit potential game designs.

For more details, see the document *Server Programming Guide*, chapter *Security*.

8.1. Security of the Server

The internal network is assumed to be secure — BigWorld does not implement security measures to safeguard processes against an attacker gaining access to the cluster's internal LAN. Operators should ensure that the usual protections for an internal network are in place. Remote access should be very strictly controlled.

The external points of contact are the area of most concern when running an exposed server. For BigWorld, these are LoginApp and BaseApp, as illustrated below:



BigWorld Server components

Due to the bandwidth needs of a massively multiplayer online game, LoginApp and BaseApp are intended to be run on machines with external access. In some sense they are the firewall. For more details, see "Blocking Ports and Related Security Considerations" on page 103.

LoginApp receives only fixed-length queries, making it easy and transparent to secure. This process is expected to be tailored by customers to suit their game, but care should be taken when doing so.

BaseApp receives more complex data, including script method calls, and is the gateway to the rest of the BigWorld Server. It has many checks in place to ensure the integrity of received data, and to discard (and warn about) corrupted data and hacking attempts. The string CHEAT is used in the log messages when BigWorld receives potentially malicious data that does not conform to its protocols (the CellApp may also use this indicator). It is advised that MessageLogger logs be monitored for messages containing that string. For details on MessageLogger, see "Message Logger" on page 55.

The security of the game-level logic rests to a certain extent with the Python scripts that implement it. For example, an entity should not be able to stab another entity that is 100 metres away. For more details on this topic and on server features such as physics checking, see the Server Programming Guide's chapter *Proxies and Players*, section "Physics Correction".

8.2. Server Ports

The list below describes the ports used by BigWorld server:

- **20013 (Protocol: UDP, Access: External)**

The default port used by LoginApp. This port can be overridden in `<res>/server/bw.xml` file's `loginApp/externalPorts` section, set the `port` configuration option. Multiple ports can be specified and the first available one is used.

- **20018 and 20019 (Protocol: UDP, Access: Internal)**

Used by BWMachineD.

- **40001-49999 (Protocol: TCP, Access: Internal)**

The Python server on BaseApp.

All BaseApps have a Python server that can be telnetted to. If no `baseApp/pythonPort` is specified in `<res>/server/bw.xml`, the port number is 40000, plus the BaseApp ID is used.

BaseApp ID numbers start at 1, so to talk to the third started BaseApp, telnet to 40003.

- **50001-59999 (Protocol: TCP, Access: Internal)**

The Python server on CellApp.

All CellApps have a Python server that can be telnetted to. If no `cellApp/pythonPort` is specified in `<res>/server/bw.xml`, the port number is 50000, plus the CellApp ID is used.

CellApp ID numbers start at 1, so to talk to the third started CellApp, telnet to 50003.

- **32768-61000 (Protocol: TCP, Access: Internal)**

Used by CellAppMgr and CellApp for viewer applications such as Space Viewer. Automatically assigned by the kernel (see UDP entry).

- **32768-61000 (Protocol: UDP, Access: Internal [External for BaseApp])**

Used by the server components: CellApp, CellAppMgr, BaseApp, BaseAppMgr, DBMgr, MessageLogger and StatLogger. Only the BaseApp has an external port.

Automatically assigned by the kernel, in the range of the kernel setting `/proc/sys/net/ipv4/ip_local_port_range`, which defaults to 32768-61000.

The BaseApp external port may be exempted from random assignment, by specifying it in the `<res>/server/bw.xml` file's `baseApp` section's `externalPorts/port` configuration options.

The port chosen can be displayed via the Watcher interface under:

- `nub/address` for internal interfaces.
- `nubExternal/address` for the external interfaces on BaseApp and LoginApp.

8.3. Blocking Ports and Related Security Considerations

Since TCP/IP is not used externally, you can block all TCP traffic. Leave all UDP/IP ports 32768 and above open, as well as the login port (20013 by default). This range can be reduced if you specify possible ports in `baseApp/externalPorts/port` settings in `bw.xml`.

Use of a separate firewall machine is discouraged. The BaseApp machines are designed to be the firewalls themselves, and perform a very similar proxying function for clients. Their amount of processing is small enough so that they can handle a whole network adapter's worth of Internet traffic. Adding another machine would only be a waste of hardware, maintenance time, and latency. BaseApps only listen on one UDP port — and so the whole TCP stack can be disabled on the external interface. The use of standard firewall software such as `iptables` may be an appropriate way to accomplish this.

See chapter *Cluster Configuration* in the Server Installation Guide for examples of using `iptables`.

Network tools such as **lsof** and **netstat** should be consulted, to ensure that you are not running any daemons listening on the external (or common) interface. Apart from `BWMachined`, you should not need to run any daemons with BigWorld, but if you wish to, then you should ensure that their ports are blocked. `BWMachined` should not listen on the external interface, only the internal interface.

Barring all TCP packets greatly improves the security of a BaseApp machine. TCP is a complicated protocol, and requires many tables and buffers to implement or firewall. By this rationale, the security of a BaseApp machine may be considered even better than that of an ordinary firewall, which must conditionally pass TCP packets.

To reduce exposure to DDOS attacks, it is recommended that the BaseApp be left to choose, within the range allocated by the operating system, a random port. This way, if an attacker discovers the IP and port of one of the BaseApp machines, then it does not mean he will automatically know those details for the other BaseApp machines.

Chapter 9. BigWorld Server Across Multiple Machines

To start a server cluster consisting of multiple machines, you can start each process in any of the following ways:

- the command line
- the `control_cluster.py` script
- WebConsole's ClusterControl module

CellApps, BaseApps, and LoginApps can have multiple instances running, while the other processes can have only one. The clients should connect to the IP address of a machine that is running a LoginApp.

9.1. How To Start

The following sub-sections describe how to start server components.

9.1.1. WebConsole

WebConsole can be used to easily start, stop and control server components.

For an outline of WebConsole, see “WebConsole” on page 51 . For operational behaviour, see the online WebConsole documentation in the Cluster Control module.

9.1.2. Auto Configuration Via `control_cluster.py`

The script `bigworld/tools/server/control_cluster.py` is generally the easiest way to start a multi-machine BigWorld Server. For details, see “Control Cluster” on page 67 .

9.1.3. Hard-Coded Scripts

Create your own shell scripts using the script `bigworld/tools/server/misc/startsingle.sh` as a guide.

9.1.4. Manual Start

During development and testing, it is feasible to manually start processes individually.

9.2. How To Stop

The method used to stop the system depends on the method used to start it. For example, calling the script `control_cluster.py` with the option `stop` stops a system started with that script. Stopping LoginApp triggers other processes to terminate, unless this feature is disabled in configuration file `<res>/server/bw.xml`.

9.3. How To Monitor

Use WebConsole¹ and `control_cluster.py`² for monitoring and recording real-time statistics. You can also get profiling data from the watchers (via WebConsole's ClusterControl module). Use WebConsole's LogViewer module to monitor the centralised log.

¹For details, see “WebConsole” on page 51 .

²For details, see “Control Cluster” on page 67 .

9.4. LoginApp and Scalability

Just as you can use multiple CellApps and BaseApps in a large cluster, you can also run multiple LoginApps in a load sharing arrangement.

To do that, DNS has to set up so that it returns multiple DNS addresses for the login server's name. If you are using BIND, then you can simply put multiple A addresses in the zone configuration file. This means that when a client looks up the DNS address of the login server, it receives a login server randomly selected from the available pool.

For example, if you are using BIND as your DNS server (most Linux distributions come with BIND as their default DNS server), the zone file can have configuration similar to the ones in the table below:

Name	TTL	CLASS	TYPE	ResourceRecord
login	600	IN	A	10.0.0.1
	600	IN	A	10.0.0.2
	600	IN	A	10.0.0.3

Example configuration on a BIND zone file

Because there are multiple A records with different IP addresses under one name, the IP address returned to a client when it looks up the address for the login server will be picked up from the IP address list, in a round robin manner.

Chapter 10. Multiple BigWorld Servers in a Single LAN

10.1. Keeping Processes Separate

All server components in a BigWorld server cluster run under the same UID. If you want to run multiple server clusters on the same network, as can be required for different stages of testing, then make sure that you use different UIDs.

You also need to make sure that the LoginApps are on different machines. This is because by default each LoginApp opens the same specific port, and therefore only one could succeed in binding to it. Alternatively, multiple LoginApps can be run on a single machine as long as they have different port settings in <res>/server/bw.xml.

Note

It is important the username and UID mapping is consistent across all machines on the cluster. It is highly recommended to use centralised logins using LDAP or similar.

10.2. Centralised Cluster Monitoring

The best way to keep track of processes is via WebConsole's StatGrapher module.

The module, which can be run on any web browser, uses log files generated by StatLogger daemon to display statistics, and has a variety of options to filter the output by user.

For more details, see "StatGrapher" on page 52 , and "StatLogger" on page 59 .

10.3. Auto-Detection of LoginApps

When running multiple servers, it is convenient for the client to auto-detect LoginApp processes (this is only suitable for a development environment).

The LoginApps have this functionality built in, and do not require any configuration to perform it — they register themselves with BWMachined, and the client broadcasts a 'find' message to retrieve their contact details. A probe message is then sent to each server to determine who, where and what it is running. This functionality is implemented on the client by class ServerDiscovery.

Chapter 11. MySQL Support

Following is described the minimal set of steps required to enable MySQL support for a server cluster. They assume that a MySQL database has been installed and configured as outlined in the Server Installation Guide's chapter *Installing the BigWorld Server*, section "Configure MySQL Server".

11.1. Compiling DBMgr with MySQL Support

The DBMgr binary provided with the default BigWorld package does not have MySQL support built-in. This allows users to quickly start using BigWorld Technology without having to configure a production database to test a simple package. See the `licensing_information.txt` document for MySQL licensing considerations.

First, to enable compilation of DBMgr, make sure that the MySQL development files are installed on the build system. To do this, as root, run the following command:

```
# yum install mysql-devel
```

Next, the DBMgr makefile needs to be modified, to enable MySQL support. Edit `bigworld/src/server/dbmgr/Makefile` and change the variable definition from `USE_MYSQL=0` to `USE_MYSQL=1`. When enabling MySQL the `USE_XML` option can be left enabled without any adverse side effects.

After these alterations, it will be possible to rebuild DBMgr. From within the `bigworld/src/server/dbmgr` directory, issue the following command:

```
$ make
```

11.2. Update `bw.xml` To Use MySQL

Once DBMgr has been compiled to communicate with a MySQL server, the game resource configuration file `<res>/server/bw.xml` needs to be updated with details on username, password and the host machine the MySQL server is running on.

The example below illustrates FantasyDemo configuration (via `fantasydemo/res/server/bw.xml`):

```
<dbMgr>
  <type>      mysql                      </type>
  <host>      my_mysql_server_machine </host>
  <username>  bigworld </username>
  <password>  my_pass </password>
  <databaseName>  fantasydemo </databaseName>
</dbMgr>
```

Example `fantasydemo/res/server/bw.xml`

For details on these fields and other relevant configuration options for your production environment, see *Server Configuration with bw.xml* on page 9.

11.3. Synchronise Database With Entity Definitions

DBMgr requires the MySQL database table structure to be synchronised with the current entity definitions. To initialise the MySQL database with the correct table structure run the `sync_db` tool:

```
$ $MF_ROOT/bigworld/bin/Hybrid64/commands/sync_db
```

This tool will not run if there are unconsolidated secondary databases. For details on data consolidation, see “Data Consolidation Tool” on page 85 .

Note

A SyncDB binary is not shipped by default with BigWorld releases. In order to use SyncDB you will need to compile the binary from `bigworld/src/server/tools/sync_db`, or have already built DBMgr with MySQL support enabled.

11.4. Enabling Secondary Databases

Secondary databases can only be enabled when MySQL support is enabled. Please see the document Server Programming Guide's chapter “Secondary Databases” for details about secondary databases.

To enable secondary database support, you must:

- Enable MySQL support (see above).
- Build the data consolidation tool by issuing the **make** command from within the `bigworld/src/server/tools/consolidate_dbs` directory.
- Set the `bw.xml` option `<baseApp>/<secondaryDB>/<enable>` to `true`.

11.5. Privileges

Below are the MySQL privileges required for each BigWorld component that communicates with a MySQL database.

Component	Required Privileges	User
DBMgr	SELECT, INSERT, UPDATE, DELETE	dbMgr/username (bw.xml)
sync_db	UPDATE, ALTER, CREATE, DROP, INDEX	dbMgr/username (bw.xml)
snapshot.py	SELECT	dbMgr/username (bw.xml)
snapshot_helper	RELOAD	dbMgr/username (bw.xml)
StatLogger	SELECT, INSERT, UPDATE, DELETE, CREATE	preferences/options/dbUser (bigworld/tools/server/stat_logger/ preferences.xml)
WebConsole	SELECT, INSERT, UPDATE, DELETE, CREATE	sqlobject.dburiame (bigworld/tools/server/web_console/ [dev prod].cfg)

Use MySQL's GRANT command to set privileges for a user.

```
$ mysql -u root
mysql> GRANT SELECT, INSERT, UPDATE, DELETE ON fantasydemo.* TO
'bigworld'@'localhost' IDENTIFIED BY 'bigworld_passwd';
```

11.6. The ClearAutoLoad tool

Over the course of a server run, entities may be marked for auto-loading so that subsequent server startups will automatically create these entities and the spaces that they inhabited, with the entity data and space data that was present in those spaces at the time that the server was shut down. Additionally, the server game time is persistent.

It is sometimes desirable to clear that data so that the server starts from a clean slate without auto-loading any entity data, and reset the server game time to 0. There is a command tool called **clear_auto_load** which can clear this data. It is located in `bigworld/bin/Hybrid64/commands/clear_auto_load`. Running it without command line arguments will clear the auto-load data in the database specified in the `bw.xml` for the current user.

```
$ bigworld/bin/Hybrid64/commands/clear_auto_load
```

You cannot run this tool while there is an instance of DBMgr or other BigWorld tool accessing the database.

Note

A ClearAutoLoad binary is not shipped by default with BigWorld releases. In order to use ClearAutoLoad you will need to compile the binary from `bigworld/src/server/tools/clear_auto_load`.

Chapter 12. RPM

BigWorld provides a RPM implementation which allows the creation of binary RPM package for BWMachined. The RPM implementation uses the RPM system available in Linux distributions such as CentOS, RHEL and Fedora to generate RPM packages.

This chapter provides the following information:

- Directory structure and files related to BigWorld RPM implementation.
- How to generate BigWorld binary RPM packages.
- Customising RPM Packages.
- Setting up a yum repository.
- Install, upgrade and uninstall using `yum` command.
- How to obtain version number of an installed package.

12.1. Directory Structures and Files

The RPM implementation is located in the `bigworld/tools/server/rpm` directory:

- The `Makefile` is used to generate RPM packages.
- The `generate.py` is used to facilitate the creation of RPM packages.
- The `binary_rpms` directory is where generated RPM packages are placed.
- The `rpm` directory also contains package specific directories. For example, the `bwmachined` directory contains all the files specific to BWMachined.

12.2. How to Generate Binary RPM Packages

Binary RPM packages must be generated using a normal Unix user account. Binary RPM packages must not be generated using the root user.

In order to allow RPM's to be built into a user writeable directory, it is necessary to modify the user's `.rpmmacros` file. This file will contain the location of the directories in which the RPM files will be built. Firstly we need to create the directory structure to use. For this example we will create a new directory in the user's home directory called `rpmbuild`. We also require an `RPMS` subdirectory where package files will be placed.

```
$ mkdir -p ~/rpmbuild/RPMS
```

Next we need to let the rpm build environment know how to use that directory. This simply involves creating a new user specific rpm macro file as follows.

```
$ vim ~/.rpmmacros
```

Copy and paste the following content into the `.rpmmacros` file.

```
%_topdir /home/username/rpmbuild
%_tmppath %{_topdir}/tmp
```

Write and close this file. To confirm the change you made has been successful, run the following command to ensure the path looks correct.

```
$ rpm --eval %_topdir
```

To generate RPM package for BWMAchined, go to the `bigworld/tools/server/rpm` directory and run the following command:

```
$ make
```

The generated RPM package will be placed in the `bigworld/tools/server/rpm/binary_rpms` directory.

The generated RPM package will have the following file name:

```
bigworld-bwmachined-<version>-<release>.<architecture>.rpm
```

- The `<version>` field is the version number of the BigWorld release that the package was generated from. This field is based on the version number in `bigworld/res/version.xml`.
- The `<release>` field identifies the specific build of this RPM package. By default, it is not used and is mapped to patch number in `bigworld/res/version.xml`.
- The `<architecture>` field will be either `i386` for 32 bit systems, or `x86_64` for 64 bit installations.

12.3. Customising RPM Packages

The most common customisation of a RPM package is the configuration file(s) installed by the package. That is, one RPM package may include the default configuration file and another may include a customised configuration file. These RPM packages can be distinguished by using the `release` field in the filename of these packages.

For example, the configuration file included in a BWMAchined RPM package is located in `rpm/bwmachined/bwmachined.conf`.

To generate a BWMAchined RPM package with the default configuration file and assign it the `release` number 0:

- Use the `bwmachined.conf` provided by default.
- Update the Release tag in `rpm/bwmachined/bwmachined_template.spec` to the following:

```
Release: 0
```

- Generate the RPM package.

The file name of the RPM package generated will be, for example, `bigworld-bwmachined-1.9.1.0-0.i386.rpm`.

To generate a BWMAchined RPM package with customised configuration file and assign it the `release` number 1:

- Update the `bwmachined.conf` as required.

- Update the Release tag to 1.
- Generate the RPM package.

The file name of the RPM package generated will be, for example, `bigworld-bwmachined-1.9.1.0-1.i386.rpm`.

12.4. Setting up a Yum Repository

For a large environment where RPM Packages, e.g. the BWMachined RPM Package, need to be installed on many machines, we recommend that a Yum repository to be set up to provide these RPM packages. This allows an RPM package to be installed on any machine in the cluster by running the **yum** command without the need to manually copy the RPM package to each machine in the cluster.

The Yum repository created will need to be made available through a web server such as the Apache HTTP Server. The setup required is outside the scope of this document. Please consult relevant web server documentation on how to achieve this.

Steps to set up a Yum repository:

- On the machine that will be hosting the Yum repository, create a directory for the BigWorld RPM packages. The location of this directory will depend on the web server chosen and the web server configuration which is outside the scope of this document.
- Copy the RPM packages to the directory created.
- Install the `createrepo` package using the following command:

```
$ yum install createrepo
```

This is a utility that will generate a common metadata repository from a directory of RPM packages.

Note

You will need to be the `root` user to install a package.

- Run the following command to create the necessary metadata for the yum repository:

```
$ createrepo <path_to_dir>
```

where `<path_to_dir>` is the path to the directory created.

For example, if the directory created is `/mnt/bigworld_repo`, then the command to run is the following:

```
$ createrepo /mnt/bigworld_repo
```

This creates a `repodata` directory in the `/mnt/bigworld_repo` directory.

- The `/etc/yum.conf` on a machine that will access the repository created will need to be updated to include the following setting:

```
[<repo_name>]
name=<repo_name>
baseurl=<url_to_repo>
```

```
enabled=1
```

where `<repo_name>` is replaced by the name given to the repository and `<url_to_repo>` is replaced by the URL that refers to the repository.

Note

By default, the `/etc/yum.conf` contains a `[main]` entry. The entry above should be added after the `[main]` entry.

When the yum repository is updated, for example, a new version of BWMAchined RPM package is added, then the `repodata` directory should be deleted and the `createrepo` command should be run again to generate up-to-date metadata.

If a new RPM package was added to the repository but is not displayed on a machine when running a query command such as

```
$ yum info bigworld-bwmachined
```

run the following command to delete the metadata used by Yum to determine the remote availability of packages:

```
$ yum clean metadata
```

When the **yum** command is run the next time, it will download the up-to-date metadata.

12.5. Install, Upgrade and Uninstall using Yum Command

This section describes how to install, upgrade and uninstall using the **yum** command. The BWMAchined package is used as an example.

12.5.1. Install and Upgrade using a RPM Package Directly

- To install a RPM Package directly without using yum repository:

```
$ yum --nogpgcheck install bigworld-bwmachined-<version>-<release>.i386.rpm
```

where `<version>` and `<release>` are replaced by the actual version and release number.

This assumes that you are running the Yum command in a directory containing the specified `.rpm` file.

Note

The RPM packages created by BigWorld RPM implementation are not signed, since these packages are from trusted source. Therefore, during installation and upgrade, GPG check is disabled.

- To upgrade a package:

```
$ yum --nogpgcheck upgrade bigworld-bwmachined-<version>-<release>.i386.rpm
```

12.5.2. Install and Upgrade using Yum Repository

- To install a package from a Yum repository:

```
$ yum --nogpgcheck install bigworld-bwmachined
```

- To upgrade a package:

```
$ yum --nogpgcheck upgrade bigworld-bwmachined
```

12.5.3. Remove an Installed Package

To remove an installed package:

```
$ yum remove bigworld-bwmachined
```

12.6. How to Obtain Version Number of an Installed Package

To obtain the version number of an installed package such as the BWMachined package, run the following command:

```
$ yum info bigworld-bwmachined
```

Chapter 13. First Aid After a Crash

If a BigWorld server component fails there are some steps that you should follow to assist BigWorld in identifying and resolve the problem as quickly as possible. These steps are outlined briefly below.

- Change your coredump output directory, if necessary.
- Determine the first process(es) that crashed.
- Generate a stack trace of the process that crashed.
- Retrieve relevant log information.
- Back up the crash information.
- Notify BigWorld Support of the crash.

A crash may be either an intentional event such as an `assert`, or unintentional event such as segmentation fault. Whenever a crash occurs a core file should be written to the `bigworld/bin/Hybrid64` directory (or equivalent directory in your system). These files provide a complete dump of the memory in use by the process at the time the failure occurred and allows deep investigation as to the state of the program which is often necessary to determine the cause of a problem.

13.1. Change your coredump output directory, if necessary

By default, processes that crash will output a coredump into their current working directory, which is generally the directory where the binaries reside. This directory may need to be changed if the user running the server does not have sufficient permissions to write to that directory. This may be the case if you have installed using RPM packages.

The output path can be temporarily changed by writing the new output path and the core file pattern to `/proc/sys/kernel/core_pattern`. Note that this will change the coredump output path for all processes (not just BigWorld processes) dumping core on that machine.

For example, a command similar to the following can be used (executed as the root user):

```
$ echo "/your/path/here/core.%e.%h.%p" > /proc/sys/kernel/core_pattern
```

Note that the change is only temporary and will not persist across machine restarts.

If a more permanent setting is required, there is the `CORE_PATH` shell variable defined in the `BWMachined` init script that can be set to a suitable output directory. For example, it can be set like this:

```
CORE_PATH=/your/path/here
```

The init script is located at `/etc/init.d/bwmachined2`. Note that after modifying `CORE_PATH`, the `BWMachined` service will need to be restarted for the changes to take effect.

13.2. Determine the first process(es) that crashed

When diagnosing a crash it is important to find the first process that crashed if there have been multiple process failures. In most scenarios multiple crashes occur due to a single failure that then propagates through the cluster, so it is important to identify where the failure started. There are two methods which can be used to achieve this.

- BigWorld logs.

- Chronologically listing core files.

13.2.1. BigWorld logs

Whenever a crash occurs the logs should always be investigated as a first step to check for any known problems or obvious errors that may have occurred. When an issue has been identified, searching around the time period for `CRITICAL` and `ERROR` messages can help quickly identify the set of processes that failed. Searching through the logs can be performed either via `WebConsole` and `LogViewer`, or via the command line utility `mlcat.py`. Using the command line utility provides more flexibility for saving logs to a file which can then be collected and sent as part of a BigWorld support ticket.

13.2.2. Chronologically listing core files

If a process failure occurs due to a non-assertion failure case you may find no obvious `CRITICAL` or `ERROR` messages appear in the server logs. In this circumstance you will need to search all of the server binary directories within your cluster to find the core files relating to a crash. This sort of operation will generally require a custom script that is suited for your environment as a cluster with hundreds of machines can be time consuming to examine.

The easiest way to perform this kind of search is to simply `ssh` to all the server machines in the cluster and perform a directory listing. Below is the output of a simple chronologically sorted directory listing showing a sequence of core files.

```
$ cd bigworld/bin/Hybrid64
$ ls -lt
...
-rw----- 1 game Company 124305408 Jan 7 10:17 core.cellapp.pc242.3030
-rw-r--r-- 1 game Company      275 Jan 7 10:16 assert.cellapp.pc242.3030.log
-rw----- 1 game Company 67174400 Jan 7 10:16 core.baseapp.pc242.3024
-rw----- 1 game Company 21635072 Jan 7 10:16 core.cellappmgr.pc242.3018
-rw-r--r-- 1 game Company      167 Jan 7 10:15 assert.baseapp.pc242.3024.log
-rw-r--r-- 1 game Company      166 Jan 7 10:15
assert.cellappmgr.pc242.3018.log
-rw----- 1 game Company 125972480 Jan 6 09:48 core.cellapp.pc242.600
-rw-r--r-- 1 game Company      275 Jan 6 09:48 assert.cellapp.pc242.600.log
-rw----- 1 game Company 21753856 Jan 6 09:47 core.cellappmgr.pc242.596
-rw-r--r-- 1 game Company      165 Jan 6 09:47
assert.cellappmgr.pc242.596.log
-rw----- 1 game Company 62255104 Jan 5 15:49 core.baseapp.pc242.601
-rw-r--r-- 1 game Company      275 Jan 5 15:49 assert.baseapp.pc242.601.log
```

From the above example you can see there are 3 separate core files that have been generated on Jan 7th. As we are interested in the earliest crash we look at the two assertions from 10:15 which are for `CellAppMgr` and `BaseApp`. We then use these assertion files to find the corresponding core files using the PIDs 3018 and 3024. Thus the core files we are interested in investigating are:

- `core.baseapp.pc242.3024`
- `core.cellappmgr.pc242.3018`

13.3. Generate a stack trace of the process that crashed

As you can see from the directory listing in the previous section, core files can be extremely large. One of the most effective ways of notifying BigWorld of a crash is to generate a simple text stack trace of the crash. This can be performed by using the GNU Debugger program `gdb`. For example using the `BaseApp` core file from the previous section we generate a stack trace as follows:

Load the `BaseApp` binary and core file into `GDB`.

```
$ gdb baseapp core.baseapp.pc242.3024
GNU gdb (GDB) 7.0
Copyright (C) 2009 Free Software Foundation, Inc.
...
Core was generated by `/home/game/game_dir/bigworld/bin/Hybrid64/baseapp
-machined --res /home/gam'.
Program terminated with signal 11, Segmentation fault.
#0  DebugMsgHelper::criticalMessageHelper (this=<value optimized out>,
    isDevAssertion=<value optimized out>, format=<value optimized out>,
    argPtr=<value optimized out>) at debug.cpp:321
321          *(int*)NULL = 0;
(gdb)
```

Use the bt (backtrace) command to generate a stack trace.

```
(gdb) bt
#0  DebugMsgHelper::criticalMessageHelper (this=<value optimized out>,
    isDevAssertion=<value optimized out>, format=<value optimized out>,
    argPtr=<value optimized out>) at debug.cpp:321
#1  0x000000000087882d in DebugMsgHelper::criticalMessage
    (this=0x2aaab4000020, format=0x2aaab4000078 "\020\326_\265\252*") at
    debug.cpp:128
#2  0x000000000007ce56c in Mercury::Channel::checkOverflowErrors
    (this=0x1b2e3120) at channel.cpp:460
#3  0x000000000007d16f1 in Mercury::Channel::addResendTimer (this=0x1b2e3120,
    seq=<value optimized out>, p=0x2aaab55fc240, roBeg=0x0, roEnd=<value
    optimized out>) at channel.cpp:898
#4  0x000000000007f8808 in Mercury::NetworkInterface::send
    (this=0x7fff6b280320, address=..., bundle=..., pChannel=0x1b2e3120) at
    network_interface.cpp:1095
#5  0x000000000007d060b in Mercury::Channel::send (this=0x1b2e3120,
    pBundle=0x1b2e2f50) at channel.cpp:769
#6  0x000000000005373c0 in Mercury::ChannelOwner::send (this=0x7fff6b27b950,
    arg=<value optimized out>) at channel_owner.hpp:33
#7  BaseApp::handleTimeout (this=0x7fff6b27b950, arg=<value optimized out>) at
    baseapp.cpp:4693
#8  0x000000000007da7cd in TimeQueueT<unsigned long>::Node::triggerTimer
    (this=0x1b2a7400, now=254536585659057) at time_queue.ipp:402
#9  TimeQueueT<unsigned long>::process (this=0x1b2a7400, now=254536585659057)
    at time_queue.ipp:184
#10 0x000000000007d9b16 in Mercury::EventDispatcher::processTimers
    (this=0x7fff6b27c1c0) at event_dispatcher.cpp:408
#11 0x000000000007d9c38 in Mercury::EventDispatcher::processOnce
    (this=0x7fff6b27c1c0, shouldIdle=true) at event_dispatcher.cpp:580
#12 0x000000000007d9c54 in Mercury::EventDispatcher::processContinuously
    (this=0x2aaab4000020) at event_dispatcher.cpp:564
#13 0x000000000007d9c69 in Mercury::EventDispatcher::processUntilBreak
    (this=0x2aaab4000020) at event_dispatcher.cpp:598
#14 0x0000000000054ddbe in BaseApp::run (this=0x7fff6b27b950, argc=<value
    optimized out>, argv=<value optimized out>) at baseapp.cpp:1125
#15 0x0000000000051a0eb in doMain (dispatcher=<value optimized out>,
    interface=<value optimized out>, argc=4, argv=0x7fff6b280b58) at main.cpp:47
#16 0x0000000000051aa65 in bwMain (argc=4, argv=0x7fff6b280b58) at main.cpp:70
#17 0x0000000000051b048 in main (argc=4, argv=0x7fff6b280b58) at main.cpp:60
```

Quit gdb.

```
(gdb) quit
```


The stack trace can then be copy and pasted into a separate text file and stored along with the other report information for the crash.

It is also possible to perform this step in one quick operation by issuing a command similar to the following:

```
$ gdb -ex 'bt' -ex 'q' baseapp core.baseapp.pc242.3024 >
core.baseapp.pc242.3024.backtrace
```

13.3.1. Troubleshooting

It is useful to identify when a bad / invalid stack trace has been generated. These can be caused by a number of issues such:

- Process binary has been modified since the core file was generated.
- Core file or binary is corrupt.
- Architecture of the core file and process binary do not match.

Below is an example of an invalid stack trace that has been generated by GDB.

```
$ gdb baseapp core.baseapp
...
warning: exec file is newer than core file.
[New Thread 7384]
[New Thread 7368]
Program terminated with signal 11, Segmentation fault.
#0 0x00000000008753d1 in MemberWatcher<double, StatWithRatesOfChange<unsigned
int>, double>::getAsString (this=0x7fff13a189bf, base=0x1, path=<value
optimized out>, result=..., desc=..., mode=@0x2e99d)
    at watcher.hpp:1846
1846                                     RETURN_TYPE value = (useObject.*getMethod_());
(gdb)
```

You can see here that the initial loading of the file is warning us that the BaseApp executable is newer than the core file that is being examined. This can indicate that the binary has been recompiled and no longer corresponds to the core file. If we continue to examine the core file we see the following output:

```
(gdb) bt
#0 0x00000000008753d1 in MemberWatcher<double, StatWithRatesOfChange<unsigned
int>, double>::getAsString (this=0x7fff13a189bf, base=0x1, path=<value
optimized out>, result=..., desc=..., mode=@0x2e99d)
    at watcher.hpp:1846
#1 0x616d732030373039 in ?? ()
#2 0x71655374754f6c6c in ?? ()
#3 0x383039313d5f7441 in ?? ()
#4 0x65646c6f202c3737 in ?? ()
#5 0x656b63616e557473 in ?? ()
#6 0x38313d5f71655364 in ?? ()
#7 0x6e69772032383736 in ?? ()
#8 0x5f657a6953776f64 in ?? ()
#9 0x6d202c363930343d in ?? ()
#10 0x6c667265764f7861 in ?? ()
#11 0x74656b636150776f in ?? ()
#12 0x000a323931383d73 in ?? ()
#13 0x0000000000c11440 in ?? ()
#14 0x000000000fa4dc00 in ?? ()
#15 0x000000000fa84050 in ?? ()
```

```
#16 0x00000000fa840ce in ?? ()
#17 0x00000000fa840d0 in ?? ()
```

What we are seeing here is GDB being unable to match the information in the core file with the executable and then presenting us with '?? ()' to indicate something is wrong. If you see this kind of output while generating a stack trace please identify the reason for this occurring and generate a new correct stack trace before sending it to BigWorld Support.

13.4. Retrieve relevant log information

Most server clusters tend not to keep all server logs for a long period of time due to the data storage requirement. If server logs are being deleted on a semi-frequent basis it is important to make a backup of the relevant log files so they can be referred to during crash analysis by either your own team or BigWorld support.

13.4.1. Generating a log summary from a core file

In order to produce a quick log summary of a crash based on a core file you can use the MessageLogger tool `mlcat.py`. For example using the BaseApp core file from previous sections we can quickly generate a log summary using the following command:

```
$ mlcat.py --around core.baseapp.pc242.3024 --context=50 >
core.baseapp.pc242.3024.log
```

This command has queried the server logs based on the timestamp of the core file and saved 50 lines of context from before and after this time into the log file `core.baseapp.pc242.3024.log`.

Note

As core files can be extremely large and take a long time to write to disk, the time stamp of a core file may be significantly later than the server logs (sometimes in the order of minutes later).

13.4.2. Archiving all logs

It is also recommended to archive the complete set of logs surrounding a crash. This can be easily performed by using the `mltar.py` MessageLogger utility program as follows:

```
$ mltar.py -zcf 20100107_logs.tar.gz

<server/message_logger>
Please select the segments to archive (e.g. 0,1,5-10):

#   Time                               Duration      Entries      Size
0   2010-01-06-09:47:07                 7h           83490        4.5MB
1   2010-01-07-10:15:31                 4h           97268        5.5MB

Enter segments to archive [all]:
```

In this scenario we have already identified that the crash has occurred at around Jan 7th at 10:16am so it would be useful to archive both log segments.

We now have an archive of the relevant server logs in the file `20100107_logs.tar.gz` which can be used by your internal team or by BigWorld support as required.

13.5. Back up the crash information

Having successfully identified all the files that are relevant to a crash, backing up the files is the next step to ensure that all data remains available should it be needed during the investigation process. It is recommended to copy all files relating to a crash into a new directory on a non cluster machine that is well named to help identify the data at a later point. For example using the core file and crash information from previous sections a new directory might be created such as `/home/game_admin/crashes/20100107_baseapp`.

You can now copy all the relevant files mentioned in the previous sections using scp or other network file transfer mechanisms suited to your environment. Below is a complete list of files recommended to keep.

- Core files (e.g., `core.baseapp.pc242.3024`)
- Binaries (e.g., `baseapp`)
- Stack trace summaries (e.g., `core.baseapp.pc242.3024.backtrace`)
- Log file summaries (e.g., `core.baseapp.pc242.3024.log`)
- Complete log files (e.g., `20100107_logs.tar.gz`)

13.6. Notify BigWorld Support of the crash

You should now have a collected set of information regarding the crash which can be sent to BigWorld Support if you find you need assistance with the analysis and diagnosis of the crash.

When reporting a process crash to BigWorld Support please include as much information as possible. After you have reviewed the core files and log output summary, if you believe the crash is related to any specific section of Python game script or other custom game resources, providing these files with the initial bug report can greatly speed up the time it will take BigWorld Support to identify an issue and assist in finding a solution.

When creating a support ticket please include the following information:

- Exact BigWorld server version used (e.g., 1.9.4.3)
- All relevant stack traces (see “Generate a stack trace of the process that crashed” on page 120)
- All relevant logs (see “Retrieve relevant log information” on page 123)
- Is the crash reproducible?
- How frequently does the crash occur?

By providing all the above information in the original support ticket it can be possible to save a number of days by allowing the support team to start work on the issue as quickly as possible.

13.6.1. Uploading large files

If you need to upload large core files and logs to assist with the support process BigWorld provides an FTP location for you to upload this data to.

`ftp://ftp.bigworldtech.com` Username: `bwguser`

 Password: `3f7eepE3`

When using this FTP location please note that as it is in use by numerous customers, naming your files as specifically as possible greatly assists BigWorld support. The FTP is only writable by customers so feel free to use file names with customer and game information, for example `customer_cellapp_report_20100128.zip`.

13.6.2. Providing BigWorld access to your server cluster

In certain scenarios it can be necessary for BigWorld support to require access to your server cluster in order to fully assist with an issue. While this is not a common occurrence it can save time if your staff is aware of this possibility and has a plan in place should it be required.

Chapter 14. Common Log Messages

This section provides the description of common log messages.

14.1. Warnings

```
CellApp    WARNING  WatcherDoc::initWatcherDoc: unable to load
watcherdoc/cellapp.xml
```

This WARNING is due to an incomplete feature to add documentation to our watchers. We apologise if it caused any confusion. You can silence this warning by creating `cellapp.xml` in `bigworld/res/watcherdoc`. The file should contain an empty root section:

```
<root>
</root>
```

```
CellApp    WARNING  InterfaceElement::expandLength: Received a message
longer than normal length
```

We try to optimise network traffic by using the smallest integer size that can encode the expected length of the message. For example, if we expect the message to be always less than 255 bytes, we will use just one byte to encode the length. We use 2 bytes for messages that are expected to be less than 65535 bytes long, etc. However, if the message exceeds the expected size, we will fallback to using 4 bytes to encode the length. The above WARNING means that a message has exceeded the expected message length.

```
CellApp    WARNING  CellApp::handleGameTickTimeSlice: Last game tick took
0.21 seconds
```

This WARNING means that the game tick took longer than the normal 0.1 seconds. This can be due to many reasons but usually it is due to a script function taking a long time. You should ensure that your script code does not do anything that takes significant amount of time. You may have to break your processing into several runs by using a timer.

```
CellApp    WARNING  CellApp: Scaling back!!
```

In response to a game tick exceeding 0.1 seconds, the CellApp will skip some of its usual processing to help decrease the load, e.g. it will reduce the amount of updates sent to the client. This WARNING is to inform you that clients may receive less updates e.g. entity movements may be jerky.

```
CellApp    WARNING  cellapp.cpp:1918: Profile CALL_TIMERS took 0.12 seconds
```

This means that the total time of all `onTimer()` callbacks in that tick took 0.12 seconds. This is bad because one tick should be less than 0.1 seconds.

```
CellApp    WARNING  cellapp.cpp:1572: Profile GAME_TICK took 0.12 seconds
```

This means the game tick took more than 0.1 seconds. This time includes the `CALL_TIMERS` time above so almost all of the game tick was in `onTimer()` callbacks.

```
CellApp    WARNING  Witness::update: 474 has a deficit of 2370 bytes (10.68
packets)
```

This is related to the `<bitsPerSecondToClient>` configuration in `bw.xml`. If `<bitsPerSecondToClient>` is 20000, then it means that the maximum number of bytes we can send to the client every tick is $((20000/\text{BitsPerSecond})/\text{UpdateHertz}) - \text{UDPOverhead} = ((20000/8)/10) - 28 = 222$ bytes. The above WARNING means that we are sending 2370 bytes to the client in one tick. This is 10.68 times the allowed data rate.

This is usually caused by method calls to the entities located on the client i.e. entities in the AoI of the player or the player itself. If many method calls are made, or if the method calls are passing large arguments, then we will exceed the bandwidth allocation for the client. When bandwidth allocation is exceeded, the position updates send to the client is reduced so entity movements will become jerky.

```
CellApp    WARNING  controller.cpp:158: Profile SCRIPT_CALL took 0.63
seconds
CellApp    WARNING  Controller::standardCallback: method = onTimer; id =
17648; controllerID = 270; userArg = 260
CellApp    WARNING  timer_controller.cpp:156: Profile ON_TIMER took 0.62
seconds
```

The above 3 WARNINGS are generated together. They mean a single `onTimer()` callback took 0.62 seconds. This is very bad since it means a single entity is using 6 times a normal game tick.

```
CellApp    WARNING  TimerController::New: Rounding up initial offset to 1
from 0 (initialOffset 0.000000)
```

This means that you are adding a timer that is less than 0.1 seconds long. Our minimum timeout is 1 tick.

14.2. Errors

```
CellApp    ERROR    CellApp: Scale back exhausted. spareTime 0.016686
tickPeriod 0.124365, real/entities 887/887, recvs 4504 trys 3958 good 0
errs (546 sels), nub timers 753 calls 752 rescheds
```

This means that the game tick has exceeded 0.1 seconds for many ticks. The CellApp has already reduced updates to the clients to a minimum but still cannot keep the game tick shorter than 0.1 seconds.

```
CellApp    ERROR    Chunk::load: Failed to load chunk
001xxxxx/sep/0010ffco: unknown error in item 'model'
```

This means that there is an error loading a model in the chunk 0010ffco. Unfortunately, there is not enough information to identify what is causing the error. You can try loading the space in World Editor and see if the same error occurs. World Editor may be able to provide more information about the error.

```
CellApp    ERROR    Received message id 41 for nonexistent entity 2844 from
192.168.50.21:49140
```

Message 41 is the `destroyEntity` message. This means that entity 2844 was already destroyed when the `CellApp` received this message. Make sure that you have not called "`self.destroy()`" in the cell and "`self.destroyCellEntity()`" in the base at the same time.

This may also be an indication of a BigWorld Server bug. If you are not able to resolve the issue, please send us more log output around the time of this error.

```
BaseApp    ERROR    BaseApp::setClient: Could not find base id 2881
BaseApp    ERROR    getProxyForCall: No proxy set!
BaseApp    ERROR    getProxyForCall: No proxy set!
```

The above messages means that the entity (2881) has been destroyed but is still receiving messages. This may be due to another entity calling a method using a mailbox of the destroyed entity. You have to make sure that when an entity is destroyed, all other entities holding a mailbox to the destroyed entity is informed so that they can reset the mailbox.

This may also be an indication of a BigWorld Server bug. If you are unable to resolve the issue please send us more logs around the time of this error.

```
BaseApp    ERROR    0.0.0.0:0: Exception was thrown: REASON_GENERAL_NETWORK
```

This means that there was an error with the network. Either the machine has been disconnected from the network, or there is some sort of hardware error.

```
BaseApp    ERROR    10.40.3.17:56859: Exception was thrown:
REASON_NO_SUCH_PORT
```

This means that either the machine has been disconnected from the network, or the destination process has crashed.

Chapter 15. Clock

The BigWorld server relies heavily on having an accurate clock for timers and load calculation. Unstable clocks (clocks that may move backward) can cause undesired behaviour and even crashes. Therefore properly configuring how BigWorld reads the time is essential for a stable, efficient server.

15.1. BigWorld Timing Methods

BigWorld provides three timing methods to select between. This can be configured by editing the file `/etc/bwmachined.conf`. For example, to use the `_gettime` timing method, include a section like this:

```
[TimingMethod]
_gettime
```

- **rdtsc** reads the time stamp counter register on the CPU.

It is known to become unstable in multi-core machines where a process moves between cores or if frequency scaling is used. In many single core systems it is completely stable and could be used without any issues.

- **_gettime** uses the kernel's clock driver to access the current time.

This method is far slower than `rdtsc` but has the ability to draw on multiple time sources to ensure a good compromise between speed, accuracy and stability is reachable. For additional system level configuration see "Linux Clock Source" on page 131 .

- **_gettimeofday** is a deprecated timing method.

This method of timing is similar to `_gettime` but has a lower theoretical maximum resolution and may be disrupted by NTP.

15.2. Linux Clock Source

Linux should automatically select an appropriate clock driver for its internal timekeeping, however at times the system administrator may need to select a different driver. If `_gettime` is used as the timing method this will have a direct impact on how BigWorld behaves.

A user may check which clocksource they are using with the following command run as root:

```
# cat /sys/devices/system/clocksource/clocksource0/current_clocksource
```

You may check which options available with:

```
# cat /sys/devices/system/clocksource/clocksource0/available_clocksource
```

Both the `tsc` and `jiffies` options are known to be unstable. There have also been reports that `acpi_pm` is unstable with some motherboard chipsets. We suggest that the `hpet` option is used if it is available, or `acpi_pm` if it is not.

Chapter 16. Machine Groups and Categories

16.1. Introduction

Machine groups are a method in which machines in a server cluster can be allocated to one or more named sets of machines. These groups can then be utilised by MessageLogger, WebConsole or your own custom utilities. Note that the use of groups is optional.

16.2. Configuration

Machine groups are created by placing a category [Groups] in the `/etc/bwmachined.conf` file, with entries containing the machine groups that the machine is assigned to. For example, two machines in the BigWorld office are allocated separately to the client and server team for each teams testing purposes during the week, but also belong to an office wide cluster group that can be used for larger scale weekend testing.

Machine A's `/etc/bwmachined.conf`:

```
[Groups]
bw_client
bw_weekend_tests
```

Machine B's `/etc/bwmachined.conf`:

```
[Groups]
bw_server
bw_weekend_tests
```

In WebConsole when starting a new server the option "On a group of machines" can then be chosen and the server will start across all machines contained in the selected group. For information on how to use Machine Groups with MessageLogger to assist scalability within a production environment please refer to the "Production Scalability" on page 58 .

Start		
<input type="radio"/>	On a single machine:	sgi02
<input type="radio"/>	From a saved layout:	
<input checked="" type="radio"/>	On a group of machines:	bw_server
	Restrict components by tags:	<input checked="" type="checkbox"/>
<input type="button" value="Go!"/>		

Starting a server using a machine group

16.3. User Defined Categories

The global configuration file `/etc/bwmachined.conf` can contain entries for user defined categories, and a list of items pertaining to it. The syntax for these categories is as follows (* meaning 0 or more occurrences):

```
*[category]  
*item
```

Server tools can query BWMAchined to find the items associated with user categories. Categories may be used by future tools or customer-created tools. The following sections outline some usages of categories currently in use by BigWorld.