Firebase Administration Guide

Version: 1.9

Firebase Administration Guide 1.9

Introduction

Welcome to the Cubeia Firebase Administration Guide.

About Cubeia

Cubeia Ltd is a distributed systems expert company, registered in England and operating through our office in Stockholm, Sweden. We provide scalable, high availability systems and consultation based on our long experience in the gambling and Internet application industry.

Our main product, Firebase, is a game agnostic, high availability, scalable platform for multiplayer online games. It is developed by Cubeia Ltd and was built from the start with the gaming industry in mind. It provides a simple API for game development using event-driven messaging and libraries for point-to-point client to server communication. Firebase is a server platform for developing and running online games. It scales from small installations to extremely large, it is built to stand up to hard traffic and can be used for almost any type of game.

Contact

For further information, please contact Cubeia Ltd UK Filial in Stockholm. Bugs should be reported to the Cubeia online support forums. If you do not have access to these forums please contact Cubeia Ltd at the address below.

Cubeia Ltd, UK Fillial
Katarinavägen 19, 4tr
11645 Stockholm
Sweden
Email: info (at the cubeia domain)
Corporate Homepage: http://www.cubeia.com
Community Community: http://www.cubeia.org

Table Of Content

System Overview

This section briefly details the Firebase system and its design. It is intended as a high level introduction to the concepts involved. Understanding the fundamental design of the system is crucial in building reliable, high-performance applications.

Introduction

A Firebase cluster is made up of one or more independent servers communicating and sharing load in what is called the cluster topology. The following vocabulary is used when describing a Firebase cluster.

A server is a single instance of Firebase running inside a Java virtual machine. Normally there will be one server per physical machine in a cluster but it is quite possible to run several virtual machine on a single computer and thus simulating a full cluster.

The server is made up of two main components, services and nodes.

A service is an internal singleton module which is available publicly in the server. Services forms the backbone of the server capabilities. Internal services are used by the system and public services can be custom developed to support games. The set of services available on a server is called the service stack.

Each server is capable of supporting one or more node. Whereas the services are static, local and have their lifetime bound to their server, nodes are dynamic, distributed and can be started and stopped independently of the server. These nodes are described in following sections.

Normally one node is loaded per server and kept running for the server lifetime. However, if a server is started in so called 'singleton' mode all nodes are loaded on one server at the same time making it possible for Firebase to be run on a single machine. Nodes can be combined in any manner on a server but for stable systems one node per server is recommended.

A game is a module loaded by the game nodes which handles the action for a particular game in the system. It co-operates with Firebase in maintaining a set of tables where clients can 'sit', 'leave' and act. The list of tables for each game is called a lobby, and is a shared data structure within the Firebase system.

Cluster Nodes

The cluster nodes in a Firebase system is equivalent to 'roles', each node is responsible for a subset of the Firebase functionality. Nodes are identified used string ID's within the system. Currently the following nodes are in use in the system:

Master

The master node is responsible for the cluster topology. It's only responsibility is controlling the cluster layout, the participating nodes, and the cluster-wide configuration. It does not handle any traffic and will be dormant for most of its lifetime. Master nodes are not distributed as other nodes, rather they use failover techniques using one "primary" master and one or

more "secondary" masters.

A Firebase installation uses a set of configuration properties which are used cluster-wide, ie. is the same for all nodes. This configuration file is handled by the primary master and propagated to the nodes when they start.

Firebase requires at least one master to be alive at all times. The server carrying the master nodes must be started before any other servers. Also, should all master nodes fail in a clustered system, the result is unpredictable and is counted as a fatal error.

Game

Game nodes are used to handle game deployments. Servers which are configured with a game node will also have game deployments and will participate in the game traffic and lifetime. Game nodes are distributed over the cluster, sharing the current system load.

Client

Client nodes handle the socket connections from actual game clients and also has a small 'session' representing each connected client. Client nodes are distributed over the cluster, sharing the current system load.

Tournament

Tournament nodes handle the tournament deployments. They can also be called MTT, for 'multi table tournament', nodes. They execute the tournament logic but not the game logic, ie. the actual game play takes place on the game nodes irregardless if the current game is part of a tournament or not, but the tournament logic and control executes on the tournament nodes. Tournament nodes are distributed over the cluster, sharing the current system load.

Servers vs. Nodes

The relationship between servers and nodes is this: A server is a standalone Java VM and may contain one or more nodes. Normally a Firebase will be deployed in a cluster, in such case each server will most likely carry one node each. On the other hand, when developing a standalone server (usually called a 'singleton') will be used most of the time and such a server will contains all nodes necessary for the system to work.

For example, a small cluster with full redundancy, without tournament support, may look like this:

- server 1 -> master1 + client1
- server 2 -> client2 + game1
- server 3 -> game2 + master2

A singleton server looks like this:

- server -> master1 + client1 + game1 + tournament1

And finally, a full scale production cluster may look like this:

- server 1 -> master1 + client1
- server 2 -> master2 + client2
- server 3 -> game1 + tournament1
- server 4 -> game2 + tournament3
- server 5 -> game3 + tournament3

Services

A service is a software module, instantiated as a singleton on a Firebase server. All services together form the service stack on single servers. Services are local to a server, as opposed to games, or their container nodes. There is no built in support for distribution, clustering or remote service control. The service stack is a prerequisite for the server, without it the server will refuse to start. If any service reports an error at start-up the server will halt immediately. The Firebase server comes with a number of built in services which are not accessible from the games, but also some public services for accessing data sources, the transaction manager, etc.

Developers can also implement their own services to provide common functionality between the games.

Cluster Topology

With cluster topology is meant the composition of nodes that makes up an entire cluster and their relation to servers in which the nodes are hosted. A single server may host a number of nodes at the same time, making the topology a non-trivial issue. However, it is recommended that a production cluster should run only single nodes per server.

Single Server / Singleton

For development and demonstration purposes it is beneficial to run on a single server. This entails starting a server instance with all nodes running in parallel and can be easily done using the special keyword 'singleton' as a node/id identifier to the startup scripts. An example of how to start a singleton on a *nix machine, may look like this:

```
./start.sh -n singleton -i server1
```

The above arguments will start a server with the ID 'server1' which will contain one each of the following nodes (with ID): master ('mas1'), client ('cli1'), game ('gam1') and tournament ('mtt1').

Minimal Fail-Over Cluster

If the player base is quite small there may be no need for a large cluster installation, but fail-over safety is still required. This is called a 'minimal cluster' and consists of two servers as detailed below (using readable ID as nodes):

- server 1 -> master1 + client1 + game1 + tournament1
- server 2 -> master2 + client2 + game2 + tournament2

In effect the above configuration starts two parallel singleton servers. Any one of the two server can be stopped without affecting game play. The startup commands on *nix systems for the above servers could look like this (in order of appearance):

```
./start.sh -n master:mas1,client:cli1,game:gam1,mtt:mtt1 -i server1
./start.sh -n master:mas2,client:cli2,game:gam2,mtt:mtt2 -i server2
```

Medium Failover Cluster

For easier scalability a medium sized cluster can be considered (using readable ID as nodes):

- server 1 -> master1 + client1
- server 2 -> master2 + client2
- server 3 -> game1 + tournament1
- server 4 -> game2 + tournament2

The above configuration keeps a backup node for all roles. It also avoids mixing master and game/tournament nodes which could foul up the failover safety. The startup commands on *nix systems for the above servers could look like this (in order of appearance):

```
./start.sh -n master:mas1,client:cli1 -i server1
./start.sh -n master:mas2,client:cli2 -i server2
./start.sh -n game:gam1,mtt:mtt1 -i server3
./start.sh -n game:gam2,mtt:mtt2 -i server4
```

Fully Scalable Cluster

For production a fully scalable fail-over safe cluster is recommended. This consists of a minimum of eight servers. It provides excellent safety and scalability (using readable ID as nodes):

- server 1 -> master1
- server 2 -> master2
- server 3 -> client1
- server 4 -> client2
- server 5 -> game1 + tournament1
- server 6 -> game2 + tournament2
- server 7 -> game3 + tournament3
- server 8 -> game4 + tournament4

A down side of the above cluster is that the game nodes shares servers with the mtt nodes, a better setup would perhaps be to increase the servers to ten and run two dedicated mtt servers. Startup arguments and servers omitted here for brevity.

HTML5

Firebase comes with HTML5 support in the form of Web Sockets and CometD. These can be enabled or disabled in the cluster properties. By default the HTTP server is enabled and will listen to port 8080.

In order to simplify development of HTML5 application Firebase can also be configured to serve static content natively. If this is enabled the default location for said content is "game/web" under the installation directory.

Installation

Requirements

Firebase requires Sun JDK 1.6.0 or higher for optimal performance. A Linux operating system with kernel version 2.6.9 or higher is recommended. The kernel should have support for the 'epoll' programming interface. Firebase has been tested on the following distributions:

- CentOS 4.5 and 5.0, http://www.centos.org/
- Debian 3.1 and 4.0, http://www.debian.org/
- Ubuntu 7.04, http://www.ubuntu.com/

Firebase has not been fully tested on any Windows distributions although it is known to work trivially, for development etc.

On Linux, the following system property must be set by default, or configured:

- JAVA_HOME

This should point to the Java installation directory.

Fast Installation

For a quick check or development, a simple installation can be made. However, for large scale testing or deployment, a full installation is recommended. To install and start Firebase on a development machine, do the following:

1. Unzip the Firebase distribution in a location of your choosing.
2. Change to the installation installation directory in a terminal and run the "`start`.sh" script on Linux, or the "start.bat" on Windows.
3. To stop Firebase, use the "`stop`.sh" script on Linux, or close the CMD window on Windows.

Full Installation Example

For more permanent installations it is important that the installation can be updated smoothly. It is recommended to use symlinks for this purpose, in effect to create an installation directory for configuration and logs, and the symlink executables to the specific Firebase version used. Also, Firebase should be run as a dedicated user.

Create User and Directories

- Create a user that Firebase should run as, root is not allowed by the start scripts:
  `useradd <user>`
- Create directory '/usr/local/firebase':
  `mkdir -p /usr/local/firebase`

- Change ownership of '/usr/local/firebase' to the newly created user:
  ```
  chown <user> /usr/local/firebase
  ```

Create Permanent Directories

This creates directories for configuration, logging and game deployment. These are semi-static in regards to a Firebase distribution and change slowly. We will later use symlinks to a real Firebase directory for all executables.

- Become the Firebase user and change directory to '/usr/local/firebase':
  ```
  su <user>
  cd /usr/local/firebase
  ```
- Create directories 'conf', 'logs' and 'game/deploy':
  ```
  mkdir -p conf logs game/deploy
  ```

Create Distribution Symlinks

By using symlinks for the executables, upgrading from one Firebase version to another is simple a matter of switching the symlinks.

- Become the Firebase user and change directory to '/usr/local/firebase':
  ```
  su <user>
  cd /usr/local/firebase
  ```
- Unzip the Firebase distribution here:
  ```
  unzip /<source dir>/firebase-<version>.zip
  ```
- A new directory with version information has been created (firebase-<version>):
- Copy all the files from 'firebase-1.0.0/conf' to 'conf':
  ```
  cp firebase-<version>/conf/* conf
  ```
- Make symbolic links to the bin and lib directories:
  ```
  ln -s firebase-<version>/lib lib
  ln -s firebase-<version>/bin bin
  ```
- Copy the start/stop scripts to the root folder:
  ```
  cp firebase-<version>/*.sh .
  ```

Upgrading

Given the above symlink installation, upgrading Firebase is a matter of unzipping and switching symlinks.

- Unzip the new distribution in '/usr/local/firebase' and change the 'bin' and 'lib' symlinks to the new destination directory that was created by the unzip operation.
- Check the release notes, they will tell you if there has been any changes to the configuration files.
- The configuration files ('conf/cluster.props' and 'conf/server.props') may have to be manually edited if a change is required.
- Make sure that the ownership of all files/directories in '/usr/local/firebase' is set correctly.

Scripts

The start scripts consist of the following files:

```
/start.sh
/stop.sh
/bin/gameserver.sh
/conf/config.sh
```

Script Configuration

The start and stop scripts of Firebase assume that the system variable 'JAVA_HOME' is set and pointed to the Java installation directory. But 'JAVA_HOME' and other default values can also be configured. This is done via the 'conf/config.sh' script which contains the following configuration variables:

- JAVA_HOME - Java installation directory.
- FIREBASE_HOME - Firebase installation directory, defaults to current directory.
- SILENT - Non-verbose mode?
- MAX_MEMORY - Java VM heap size, default to 512M.
- GC_ARGUMENTS - Java VM garbage collection arguments, defaults to '-XX:+UseParallelGC'.
- HOST_IP - Host IP used for JMX connections, defaults to the first value in `hostname -i`.
- HOST_NAME - Host name used for the server ID, defaults to `hostname`.
- DISABLE_EPOLL - Disable EPOLL?
- JMX_PORT - JMX port to use, defaults to 8999.
- MAX_WAIT - Max wait for start/stop in seconds, defaults to 90.
- DEBUG_OPTS - Debug options for the VM

Start / Stop

The "start.sh" and "stop.sh" scripts can be used from the installation directory. For example:

```
cd /usr/local/firebase
./start.sh
```

Startup Parameters

With no arguments to the start script a singleton will be started, to control which nodes are started the following syntax is used:

```
-n <node-type>:<node-id>[,<node-type>:<node-id>]*
```

The available node types are:

- “master” - Master node.
- “game” - Game node.
- “client” - Client gateway node.
- “mtt” - Tournament node

The node ID should be unique across the cluster, so for example, to start a master node, a client node and a tournament node at the same time:

```
./start.sh -n master:mas1,client:cli1,mtt:mtt1 -i server1
```

As show above you should also provide a "-i" switch which sets an ID for the entire server, this should be set to the server name.

Startup Notification

The server currently uses a "system log" file to print a message to when it considers itself started. This unfortunately means that it is possible to get the scripts to misbehave by changing the log configuration. The following logging configuration is required in for this startup message to appear:

```
<!-- The Firebase System Log, used by the start scripts -->
<appender name="SYSTEM_LOG" class="org.apache.log4j.RollingFileAppender">

  <param name="File"   value="logs/system.log" />
  <param name="Append" value="false" />
  <param name="MaxBackupIndex" value="5" />
  <param name="MaxFileSize" value="10MB" />
  <layout class="org.apache.log4j.PatternLayout">
    <param name="ConversionPattern" value="%d{ISO8601} - %-5p - %m%n"/>
  </layout>
</appender>
<category name="SYSLOG" additivity="false">
  <priority value="info"/>
  <appender-ref ref="SYSTEM_LOG"/>
</category>
```

The server will print three information lines to the system log when it is started. The start scripts waitforserver method is monitoring the above 'system.log' for these lines to appear before it exits. Below is an example startup message as it appears in the system log.

```
2008-10-28 ... - INFO  - System [Java HotSpot(TM) ...]
2008-10-28 ... - INFO  - Server started [Tue Oct 28 10:45:16 CET 2008]
2008-10-28 ... - INFO  - Firebase Server Platform [version: 1.5-RC.4]
```

Configuration

Secure Sockets

Normally a gaming system should implement SSL support via certificates at the firewalls or load balancers. However, for a simple setup or testing Firebase can be configured to use SSL certificates for client socket connections. This configuration is done in the server properties on each client gateway node. There are three properties used:

- “ssl-enabled” - This is a boolean true or false value. SSL is disabled by default.
- “ssl-key-store-password” - This is a required property if SSL is enabled, it should be the password for the key store file.
- “ssl-key-store” - This is a required property if SSL is enabled, it should point to the key store file.
- “ssl-key-store-type” - This is the type of the key store, 'jks' or 'pkcs12', which defaults to 'jks' if not specified.

MCast Addresses

> **Note:** The following configuration applies to Firebase Enterprise Edition only.

Firebase uses a number of MCast addresses for its communication. Primarily for discovering members in cluster communication. These MCast addresses are automatically generated from the cluster communication channel address which is configured in the server properties, default value:

```
cluster-mcast-address=224.0.23.23:9785
```

The above address is configured on all servers in order for a cluster to start. Using this as a start address, Firebase will generate mcast addresses for its internal services. Currently, seven addresses will be used. These addresses can also be manually configured in the cluster properties. Below is the seven mcast addresses, their services, and the configuration property which can be used in the cluster properties to manually set them to another value:

```
224.0.23.24:9786 - Space replication / MTT \
service.space.mttspace.mcast-address

224.0.23.25:9787 - Space replication / Tables \
service.space.tablespace.mcast-address

224.0.23.26:9788 - Message bus / Game events \
service.mbus.dqueue.game.mcast-address

224.0.23.27:9790 - Message bus / Client events \
service.mbus.dqueue.client.mcast-address

224.0.23.28:9791 - Message bus / Chat events \
service.mbus.dqueue.chat.mcast-address

224.0.23.29:9792 - Message bus / MTT events \
service.mbus.dqueue.mtt.mcast-address

224.0.23.30:9793 - System state replication \
systemstate.mcast-address
```

Configuration Properties

Files and Formats

Firebase uses two main configuration files:

- server.props - This file contains configuration properties for a single server and must be installed on each server in a cluster. Crucially it contains hardware specific information such as IP addresses.
- cluster.props - This file contains configuration properties for the entire cluster. It need only to be placed on master servers in a cluster and will propagate from the masters to the rest of the nodes.

The properties files uses a normal Java property file format.

For Firebase CE or a server in singleton mode, both the above files must be used. In a cluster, the 'cluster.props' can safely be removed from servers not running master nodes.

Some properties use the node ID to differentiate between nodes of the same type. In the tables below this will be indicated using '<id>'. In such case the '<id>' should be replaced with the ID of the actual node.

Property Inheritance

In the tables below, almost every property 'inherits' values from their parent namespace. If for example a service is looking at the following property...

```
service.<public-id>.jta-enabled
```

... to determine if it is to use JTA transactions for it events, it will take its public ID, let's say 'myWallet', and lookup the following properties in order, and use the first one it finds:

1. service.myWallet.jta-enabled
2. service.jta-enabled
3. jta-enabled

From this follows that if you want to enable JTA for all services, you can set 'service.jta-enabled' to make it global. All inheritance is 'descendant first', so that the most specific property will be used in each instance.

If a property *does not* use inheritance, this will be noted in the comments.
In the configuration tables below, the following data types are used:

- string - Any string, note that leading a trailing spaces will be truncated.
- directory - String interpreted as a directory path, absolute or relative to the current directory.
- file - String interpreted as a file path, absolute or relative to the current directory.
- integer - 32 bit signed Java integer.
- long - 64 bit signed Java integer, normally used for millisecond values.
- boolean - Boolean 'true' or 'false'.
- string-list - A list of strings, separated by commas.
- ip address - (see below)
- thread pool properties - (see below)

IP Address

Addresses in string format are expressed as <name>:<port>. Usually an IP address requires both port and name, if the port is optional '-1' can often be used, however, the port must be available. In other words: 'localhost' is not a legal IP address, whereas 'localhost:8080' or even 'localhost:-1' is. The 'name' portion of the address can be either a host name or an IP address, but IP addresses are to prefer to avoid DNS problems. No wildcards or subnets are supported. And currently only IPv4 addresses are supported.

Thread Pool Properties

There are a number of thread pools used with the system. They are configured via a comma separated list of values. The values are, in order:

1. Core / Minimum size of the thread pool.
2. Maximum size of the thread pool.
3. Time to live in milliseconds for idle connections.
4. Enable queueing, optional boolean value.
5. Queue size, optional size of queue.

Value four and five are optional, but value five must be provided if four is. Here follows some examples:

- “1,2,60000” - Core size 1, max size 2, 60 seconds TTL, no queueing
- “1,5,20000,true,5” - Core size 1, max size 5, 20 seconds TTL, with a 5 element queue

Thread pool properties are currently only used for cluster communications.

Property Substitution in Server Properties

The server properties may contain syntax for string replacement against system properties. This is done via a normal curly brackets substitution. For example, you may want to say something like this in the server properties...

```
cluster-bind-address=${internal.ip}
```

... which will cause Firebase to try to replace "${internal.ip}" with the system property name "internal.ip". In this case, you would probably modify your startup script to include something like this:

```
-Dinternal.ip=`hostname -i`
```

Server Properties

Table 4.1. Core Server

Property	Data Type	Default Value	Comment
cluster-mcast-address	ip address	null	cluster communication channel - *mandatory* - no inheritance
cluster-ping-timeout	integer	500	timeout for cluster discovery in milliseconds
cluster-bind-address	ip address	null	bind address, may default to localhost if unspecified - no inheritance
cluster-bind-interface	string	null	bind interface name (eg. 'eth1') - no inheritance
cluster-bind-	integer	7800	bind NIC port range start - no inheritance

address-start-port			
cluster-bind-address-end-port	integer	7900	bind NIC port range end - no inheritance
client-bind-address	ip address	0.0.0.0:4123	bind address for client nodes - no inheritance
web-client-bind-address	ip address	0.0.0.0:8080	bind address for web client nodes (WebSocket/Comet)- no inheritance
web-client-ssl-bind-address	ip address	0.0.0.0:8443	bind address for web client nodes (WebSocket/Comet) with SSL - no inheritance
autostart-services	string-list	null	comma separated list of service ids for auto starting - no inheritance
log-directory	directory	./logs	service log directory - no inheritance
ssl-enabled	boolean	false	enable / disable SSL - no inheritance
ssl-key-store-password	string	null	key store password, mandatory if SSL is enabled - no inheritance
ssl-key-store	file	null	key store file, mandatory if SSL is enabled - no inheritance
ssl-key-store-type	string	jks	key store type, either 'jks' or 'pkcs12' - no inheritance

Table 4.2. Message Bus

Property	**Data Type**	**Default Value**	**Comment**
service.mbus.dqueue.game-channel-bind-address	ip address	null	bind address for game multicast messages
service.mbus.dqueue.client-channel-bind-address	ip address	null	bind address for client multicast messages
service.mbus.dqueue.chat-channel-bind-address	ip address	null	bind address for chat multicast messages

service.mbus.dqueue.mtt-channel-bind-address	ip address	null	bind address for tournament multicast messages
service.mbus.dqueue.game-channel-bind-interface	string	null	bind interface name (eg. 'eth1') for game multicast messages
service.mbus.dqueue.client-channel-bind-interface	string	null	bind interface name (eg. 'eth1') for client multicast messages
service.mbus.dqueue.chat-channel-bind-interface	string	null	bind interface name (eg. 'eth1') for chat multicast messages
service.mbus.dqueue.mtt-channel-bind-interface	string	null	bind interface name (eg. 'eth1') for tournament multicast messages

Table 4.3. Table Space

Property	Data Type	Default Value	Comment
service.space.tablespace.bind-address	ip address	null	bind IP address for multicast messages
service.space.tablespace.bind-interface	string	null	bind interface (eg. 'eth1') for multicast messages

Table 4.4. Tournament Space

Property	Data Type	Default Value	Comment
service.space.mttspace.bind-address	ip address	null	bind IP address for multicast messages

Table 4.5. System State

Property	Data Type	Default Value	Comment
systemstate.bind-address	ip address	null	bind IP address for multicast messages

Table 4.6. Routable Services

Property	Data Type	Default Value	Comment
service.<public-id>.router-pool-	thread pool	1,3,60000	router thread pool, for

properties	properties		incoming events
service.<public-id>.sender-pool-properties	thread pool properties	1,3,60000	sender thread pool, for out going events
service.<public-id>.jta-enabled	boolean	false	enables or disables JTA for the service

Cluster Properties

Table 4.7. Core Cluster

Property	Data Type	Default Value	Comment
node.client.<id>.name	string	null	readable node name
node.client.<id>.client-reconnect-timeout	integer	60000	millisecond timeout for re-connections
node.client.<id>.acceptor-io-threads	integer	4	underlying number of I/O threads
node.client.<id>.event-daemon-threads	integer	16	internal event executor pool
node.client.lobby-broadcast-period	integer	2000	interval between lobby updates, in milliseconds - no inheritance
node.client.local-packet-max-fixed-access-frequency	integer	n/a	max access frequency for non-logged in packets, in milliseconds
node.client.local-packet-interval-access-frequency-length	integer	n/a	frequency access interval for non-logged in packets,, in milliseconds
node.client.local-packet-interval-access-frequency	integer	n/a	max accesses within interval for non-logged in packets
node.master.<id>.name	string	null	readable node name
node.master.<id>.scheduler-pool-init-size	integer	1	internal scheduling pool
node.master.<id>.resume-command-delay	integer	2000	resume grace period for lingering events, in millis
node.master.<id>.name	string	null	readable node name

node.game.<id>.event-daemon-threads	integer	64	internal event executor pool
node.game.<id>.activator-halt-on-init-error	boolean	true	should halt system on activation errors?
node.game.<id>.player-reconnect-timeout	integer	120000	player reconnection timeout in milliseconds
node.game.<id>.player-reservation-timeout	integer	30000	player reservation timeout in milliseconds
node.mtt.<id>.event-daemon-threads	integer	16	internal event executor pool
node.client.<id>.enable-http-server	boolean	true	enable or disable HTTP client access

Table 4.8. HTTP

Property	Data Type	Default Value	Comment
node.client.<id>.disable-static-http-content	boolean	true	enable or disable serving of static HTTP content
node.client.<id>.static-web-directory	string	"game/web"	static web content directory, relative to server roots
node.client.<id>.allow-static-web-directory-listing	boolean	false	enable or disable directory listings
node.client.<id>.json-max-text.message-size	long	512000	max JSON string message size
node.client.<id>.web-socket-max-idle-timeout	long	300000	idle timeout for web sockets
node.client.<id>.comet-poll-timeout	long	300000	comet poll timeout in milliseconds
node.client.<id>.comet-idle-timeout	long	5000	timeout between polls after which the connection is considered idle, in milliseconds
node.client.<id>.enable-http-cross-origin-filter	boolean	true	enable or disable the cross origin filter

Table 4.9. HTTP Cross Origin

Property	Data Type	Default Value	Comment
node.client.http.cross-origin.<id>.allowed-origins	string list	"*"	comma separated list of allowed origins, "*" means all
node.client.http.cross-origin.<id>.allowed-methods	string list	"GET,POST"	comma separated list of allowed methods
node.client.http.cross-origin.<id>.allowed-headers	string list	"Origin,Content-Type,Accept"	comma separated list of allowed headers
node.client.http.cross-origin.<id>.preflight-max-age	int	1800	the number of seconds a preflight request can be cached by a client
node.client.http.cross-origin.<id>.allow-credentials	boolean	true	allow or deny requests with credentials

Table 4.10. Login Service

Property	Data Type	Default Value	Comment
service.loginmanager.number-of-threads	integer	1	number of threads per login handler type

Table 4.11. Space Service

Property	Data Type	Default Value	Comment
service.space.buddy-reorg-delay	integer	500	buddy group grace delay on startup, in milliseconds
service.space.table-space-object-warn-size	integer	10240	table debug state size threshold, in bytes
service.space.mtt-space-object-warn-size	integer	10240	tournament debug state size threshold, in bytes

Table 4.12. Ping Service

Property	Data Type	Default Value	Comment
service.ping.ping-enabled	boolean	false	enable / disable ping failure detection
service.ping.initial-ping-delay	integer	20000	allowed client idle time before ping start, in millis
service.ping.ping-interval	integer	5000	ping interval, in millis
service.ping.ping-timeout	integer	3000	ping timeout, in millis
service.ping.failure-threshold	integer	1	number of times a client may miss a ping
service.ping.number-of-threads	integer	1	number of threads used for pinging

Table 4.13. Table Space

Property	Data Type	Default Value	Comment
service.space.tablespace.mcast-address	ip address	null	multicast address for communication, auto generated
service.space.tablespace.cache-mode	string	REPL_SYNC	REPL_SYNC | REPL_ASYNC
service.space.tablespace.lock-aquisition-timeout	integer	10000	cache lock aquisition timeout in millis
service.space.tablespace.synch-replication-timeout	integer	20000	timeout for synchronous global RPC calls
service.space.tablespace.jta-enabled	boolean	false	enables or disables JTA for the space

Table 4.14. Tournament Space

Property	Data Type	Default Value	Comment
service.space.mttspace.mcast-address	ip address	null	multicast address for communication, auto generated

service.space.mttspace.cache-mode	string	REPL_SYNC	REPL_SYNC | REPL_ASYNC
service.space.mttspace.lock-aquisition-timeout	integer	10000	cache lock aquisition timeout in millis
service.space.mttspace.synch-replication-timeout	integer	20000	timeout for mttspace global RPC calls
service.space.mttspace.jta-enabled	boolean	false	enables or disables JTA for the space

Table 4.15. System State

Property	Data Type	Default Value	Comment
systemstate.mcast-address	ip address	null	multicast address for communication, auto-generated by default
systemstate.cache-mode	string	REPL_ASYNC	(see config file for allowed values)
systemstate.isolation-level	string	REPEATABLE_READ	(see config file for allowed values)
systemstate.use-repl-queue	boolean	false	use a queue for replication?
systemstate.repl-queue-interval	integer	100	interval in milliseconds
systemstate.repl-queue-max-elements	integer	50	max queue elements
systemstate.lock-aquisition-timeout	integer	5000	cache lock aquisition timeout in millis
systemstate.synch-replication-timeout	integer	20000	timeout for synchronous global RPC calls

Table 4.16. Message Bus

Property	Data Type	Default Value	Comment

service.mbus.dqueue.game.mcast-address	ip address	null	comm channel for game events, auto-generated by default
service.mbus.dqueue.client.mcast-address	ip address	null	comm channel for client events, auto-generated by default
service.mbus.dqueue.chat.mcast-address	ip address	null	comm channel for chat events, auto-generated by default
service.mbus.dqueue.mtt.mcast-address	ip address	null	comm channel for tournament events, auto-generated by default

Table 4.17. Client Node

Property	Data Type	Default Value	Comment
client.gateway.mina-logging-enabled	boolean	false	log all communication (packets)
client.gateway.handshake-enabled	boolean	false	use handshake before accepting client session
client.gateway.handshake-signature	int	1128351298	signature used to verify handshake
client.gateway.max-packet-size	int	32768	max packet size in bytes, for binary clients
client.gateway.encryption-enabled	boolean	false	use simple non-SSL encryption or not
client.gateway.encryption-filter	string	null	uses an AES crypto algorithm by default, if enabled
client.gateway.max-number-of-sessions	int	16384	max number of binary client on a gateway node, does not include web clients

Table 4.18. Statistics

Property	Data Type	Default Value	Comment

server.statistics.level	string	PROFILING	PROFILING | DEPLOYMENT

Miscellaneous

Flash Cross Domain Service

Firebase comes with a bundled service for service a cross domain policy file to connecting Flash clients. The service is configured by two files in the 'conf' directory.

- crossdomain.props -main service configuration
- crossdomain.xml - the policy file, the file name may be changed in the crossdomain.props file

Example: crossdomain.props.

```
####################################################
# Properties for Flash cross domain policy service #
####################################################

# name of the file containing the cross domain policy
# note: this file must be in the server config directory
CrossDomainPolicyFile=crossdomain.xml

# cross domain policy server port
CrossDomainServicePort=4124

# use mina logging filter
UseMinaLoggingFilter=false

####################################################
#                      E O F                       #
####################################################
```

Example: crossdomain.xml.

```
<?xml version="1.0"?>
<cross-domain-policy>
  <allow-access-from domain="*" to-ports="4124"/>
</cross-domain-policy>
```

HTTP Server Cross Domain

If the HTTP server is enabled Firebase also enables a cross domain filter for all web socket or comet requests. The configuration of this filter is listed under the cluster properties above and is equivalent of configuring the Jetty cross origin filter[1].

---

[1] http://wiki.eclipse.org/Jetty/Feature/Cross_Origin_Filter

Management

Monitoring

The preferred way of monitoring a Firebase installation is through the Java Management Extension (JMX). This is a standard interface for enterprise Java applications and Firebase publishes a number of interfaces (MBeans), primarily for:

- Monitoring and Statistics
- Cluster Topology and Control
- Diagnostics and Debugging

The Java VM also exposes valuable internal information such as memory allocations, processes and CPU usage.

For debugging and initial monitoring all Sun Java distributions comes with a tool, JConsole, which can be used to monitor JMX interfaces and VM statistics.

Operations

DOS Protection

Firebase includes rudimentary support for battling DOS attacks for non-logged in users. Ie. all packets that can be sent to a Firebase installation without logging in, are subjected to optional access rules. These rules are:

- fixed - Interval in which there can only be one request, ie. 'max one request every MaxFixedAccessFrequency milliseconds'.
- frequency / interval - A number of requests with a given interval, ie. 'max IntervalAccessFrequency requests within IntervalAccessFrequencyLength milliseconds'.

These two values are set via configuration. By default, no protection is configured. The configuration is separated into two areas, 'lobby' and 'gateway'. The 'lobby' protection regards only lobby queries and subscriptions as these are data and computation heavy, whereas 'gateway' protects the rest.

Lobby

The lobby protection is guarding the following packets:

- LobbyQueryPacket
- LobbyObjectSubscribePacket
- LobbySubscribePacket

Protection is configured via the 'service.lobby' namespace in the cluster properties. If a packet is dropped for a client a warning will be logged the first time it happens.

Gateway

The gateway protection is guarding the following packets:

- VersionPacket
- GameVersionPacket
- LobbyObjectUnsubscribePacket
- LobbyUnsubscribePacket
- LoginRequestPacket

Protection is configured via the 'node.client' namespace in the cluster properties. If a packet is dropped for a client a warning will be logged the first time it happens.

Updating

Rolling Updates

A 'rolling update' is as update of the server software in a cluster which does not halt the game play. It is done by stopping, updating and restarting one server at the time. Since Firebase handles topology changes gracefully a server can be stopped and later started transparently to the user.

Caveats

- A rolling update can only be performed as long as the serialized form of the involved Java classes has not changed. In other words, logic may change (to a certain degree of course), but data format must not.
- An update should be performed when the system has spare capacity. When a server is brought down, its 'load' (CPU, band width etc) is distributed to the remaining servers. If then the load is greater that the remaining servers can handle, the system as a whole may become unstable.
- Rolling updates should only ever be performed after having been comprehensively tested in a staging environment.
- You must have a redundant topology to perform a rolling update. In other words, you must have at least one of each node type running at all times.

How-To

Given the above caveats, performing a rolling update is simple. It involves updating the software on all cluster machines, and the sequentially stopping and starting each machine.

1. Update the software. Firebase loads binaries eagerly, which means you can overwrite the binaries even if the server is currently running, this makes it easier to script the update.
2. Sequentially stop and start each server. Each server should be completely stopped and completely started before the next server is approached, and it is recommended that you start with the master nodes, primary followed by secondary. This is because the master nodes are not immediately involved in game play, so should the master nodes fail you stand a better chance of either notify all players and perform a system wide reboot or rolling back the master to previous versions.

What If Something Goes Wrong?

If an update fails for some reason it will most likely involve changed data format (serialized forms) or external integrations (databases, 3rd party wallets etc). Given this, it is very hard to give any general advice.

- If a node refuses to start: Verify game play on remaining nodes and make sure the error logs are 'silent'. If every thing looks correct, rollback to previous version and attempt to start. If the rollback fails or the remaining nodes display errors you may have to restart the entire cluster.

- If a node starts but displays errors or inconsistent behaviour: You may still stand a chance of rolling back the change, however since the risk of cascade failures have increased significantly you must seriously consider the option of a cluster wide restart.