Table of Contents

Part I Server Overview 4

1 ElectroServer 4 Overview ... 4

2 Common Applications of ElectroServer ... 4

Real-Time Isometric Environment ... 4
Lobby and Game System ... 4
Audio/Video Streaming ... 4

3 Server Versions ... 5

Standalone Mode ... 5
Distributed Mode ... 5

4 Web-based Administrator ... 6

Part II Installation 6

1 Hardware ... 6

2 Operating System ... 7

Windows ... 7
OS X ... 8
Linux ... 8
Other Unix ... 8

3 Initial Configuration ... 9

Registry and Standalone ... 9
Gateway ... 9

Part III Administration 9

1 Web Based Administrator ... 9

Accessing the Web-based Administrator ... 10
The Web Admin Sections ... 10
General Settings ... 10
Manage Licenses ... 10
Edit Server Settings ... 10
Edit Thread Settings ... 11
Restart or Shutdown Server ... 11
Security Settings ... 11
Manage Permission Sets ... 11
Manage Web Admin Users ... 12
Change Keystore Password ... 12
Gateways ... 12
View existing gateways ... 12
Edit a gateway ... 13
Create a new gateway ... 13
Filters ... 13
Manage Flooding Filters ... 13
Manage Language Filters ... 13
Manage Word Lists ... 13
Extensions ... 13
Upload Extensions ... 14
Create New Server-level Component ... 14

© 2008 Electrotank, Inc.

Persistent Rooms 14
Create new Persistent Room 14
Restart Server 14

Part IV Server in Detail 14

1 Chat 14
Public Messaging 14
Private Messaging 15
Extensions 15
Filters 15

2 Filters 15
Language Filter 15
Filter Match During Public Message 16
Filter Match During Login 16
Filter Match During Room Creation 16
Filter Match During Private Message 16
Flooding Filter 16
Filter Parameters 16
Assigning a Flooding Filter to a Room 17

3 Rooms and Zones 17
Zones 17
Rooms 17
Creating a Room 17
Joining a Room 18

4 Extensions 19

5 Security 19
User Permissions 19

6 Room Variables 20
Why Are Room Variables Useful? 20
Creating a Room Variable 20
Updating a Room Variable 20
Room Variables and Extensions 20

7 User Variable Types 20
User Variables 20
User Plugin Variables 21
User Extension Variables 21
User Server Variables 21

Part V Extensions 21

1 Types of Extension Components 22
Plug-ins 22
Managed Object Factories 22
Event Handlers 22
Login Event Handler 23
Logout Event Handler 23
User Variable Event Handler 23
Buddy List Event Handler 23
Private Message Event Handler 23
Extension Lifecycle Event Handler 23
Audio/Video Event Handler 23

© 2008 Electrotank, Inc.

2 Extension Structure and Deployment 23
Extension.xml 24
3 Extension Loading and Start-Up/Shut-Down Process 25
4 Extension Reloading Process 26
5 Server API 26

Part VI Advanced Topics 27
1 Understanding the EsObject 27
Where EsObjects can be Used 27
EsObjects Used During Data Exchange 27
EsObjects Used as Property Values 27
Supported Data Types 27
Creating an EsObject 28
2 Game Manager 28
Game Types 28
Game Details 29
Loading a Game List 29
Creating or Joining a Game 29
The Game's Plugin(s) 30
3 Buddy List 30
Buddies are Session-Based 31
Persisting the Buddy List 31
Client Requests and Events 31

Part VII Audio and Video Streaming 31

Part VIII Performance and Optimization 32
1 Adjusting ElectroServer's Memory 32
Windows 32
Executable 32
Service 33
Unix 33

Part IX Clients 34
1 ActionScript 34
ActionScript API Overview 34
The ElectroServer Class 34
Requests, Responses, and Events 35
Event Handling 35
Differences between the ActionScript 2 and ActionScript 3 APIs 35
Commonly Used Classes and Methods 36
Commonly Used Classes 36
Common ElectroServer Class Methods 36
2 Errors 36

Index 0

© 2008 Electrotank, Inc.

1 Server Overview

1.1 ElectroServer 4 Overview

ElectroServer 4 is multiplayer server product that facilitates interaction between many connected users. Having been tested with as many as 200,000 concurrent users, ElectroServer 4 is highly scalable. In addition to multiplayer capabilities, ElectroServer is used for real-time audio and video streaming and recording.

ElectroServer works by allowing client applications, such as Flash, Java, or Silverlight, to connect via socket to it and log in. This connection is persisted as long as the client wants to stay connected. While connected the server can push data to the client or the client can make requests of the server at any time.

Developers often find zones and rooms useful. A room is a collection of users that can all “see” each other. These users can easily communicate to achieve chatting or multiplayer game play. A zone is a collection of rooms. Chatting can occur as public messages sent to an entire room of users, or private messages that are sent to one or more specific users in any room. Users can add other users as buddies. Users are informed when their buddies join and leave the server.

The server is highly extensible using what are called extensions. An extension is a collection of 1 or more ActionScript or Java files/classes that are used to add more functionality to the server. These extensions can be used as server-level event handlers, such as the Login Event Handler, as Managed Objects which come in handy for things like database connection pooling, or as Plugins. Plugins are run at the server level or at the room level and are frequently used to execute game logic.

The server is configured and maintained via a powerful web-based administration tool. This allows for management of things like user permission sets, language filters, extensions, and persistent rooms.

1.2 Common Applications of ElectroServer

ElectroServer can be used to create any number of multiplayer experiences. But here are a few common types of applications that are created using ElectroServer.

1.2.1 Real-Time Isometric Environment

If you’ve seen Webkinz, Club Penguin, or Barbie Girls, then you’ve seen a real-time isometric environment application. These games allow thousands of users to manipulate a character in a virtual world, depicted in a 2.5D isometric view. ElectroServer has been used by companies since 2001 to power such environments. Now ElectroServer 4 brings an unprecedented scalability.

1.2.2 Lobby and Game System

One of the most common ways to play a multiplayer game on the Internet is by logging into a chat and then hopping into an open slot in a game. This approach works well for both turn-based and real-time multiplayer games. Through a system like this users can view the current list of games, create new games, or join existing games. See the **SimpleChat** example extension for chatting.

1.2.3 Audio/Video Streaming

ElectroServer 4 provides the ability to stream Flash video to connected clients. The video stream comes from a video file, another user’s web-cam, or streamed live event such as a TV show or concert. ElectroServer is one of the few solutions in existence that can meet this need. See these example extensions:

- RecordVideo
- VideoStreamingExample
- WebCamAndMicrophoneChat

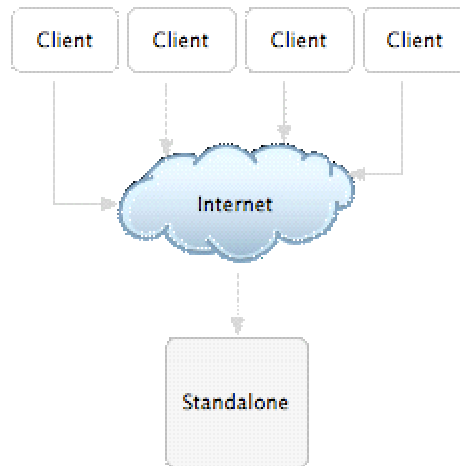
© 2008 Electrotank, Inc.

1.3 Server Versions

ElectroServer 4 is sold as ElectroServer 4 Pro and ElectroServer 4 Enterprise. ElectroServer 4 Pro can only be installed in standalone mode whereas ElectroServer 4 Enterprise can be installed in standalone mode or distributed mode.

1.3.1 Standalone Mode

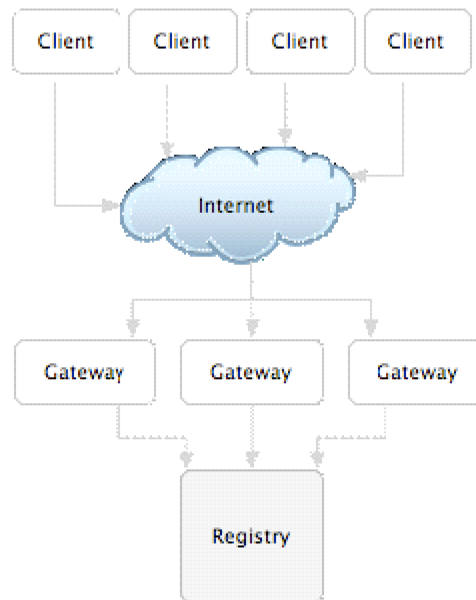
This mode is available in ElectroServer 4 Pro and Enterprise. The server is installed and runs on one physical machine. Clients connect to ElectroServer directly.



1.3.2 Distributed Mode

This mode is only available in ElectroServer 4 Enterprise. The server is installed on multiple machines. The installations are made up of one registry server and one or many gateway servers. The single registry server keeps track of the server state (users, rooms, extensions, etc). The gateway servers manage the client connections. This allows the system to scale into the hundreds of thousands.

© 2008 Electrotank, Inc.



1.4 Web-based Administrator

ElectroServer is maintained remotely through a powerful web-based administration tool. This allows for management of things like user permission sets, language filters, extensions, and persistent rooms. It is also used to install a license file which unlocks ElectroServer for higher numbers of concurrent users.

For information on more specific concepts or features, detailed articles and tutorials are available on the ElectroServer website www.electro-server.com.

2 Installation

ElectroServer is a Java application. This means that any computer that can run the Java Runtime Environment (JRE) 1.6, also known as the Java virtual machine, is capable of hosting ElectroServer successfully. Windows 98/ME/NT/2000/XP, Solaris, and Linux are among the supported operating systems.

Read the section relating to hardware to see suggestions on the physical servers you can use or on your chosen operating system below to learn how to install ElectroServer.

2.1 Hardware

While ElectroServer is flexible and can run on many different hardware configurations, it performs best when a few things are kept in mind.

ElectroServer is highly concurrent.
ElectroServer is designed to support many concurrent users. To do this, ElectroServer uses a highly-scalable thread pool internally. Along with using this pool for processing user requests, the server also uses this pool for many background tasks that are running at all times. This means that adding more processors (or cores) will invariably increase performance of the server.

Ultimately though, processor speed is the most important item for scalability. In both a standalone

© 2008 Electrotank, Inc.

instance and a registry instance, processor speed is number one with concurrency a close second. When dealing with gateways you can use smaller machines as you can simply scale by adding more gateways into the mix.

The more RAM the merrier.
The exact amount of RAM necessary on the server depends entirely on the number of processors and the type of traffic. In general, every core or processor should have at least one gig of RAM. This means a single processor dual core machine should have two gigs. A dual processor, dual core machine should have at least four gigs.

This rule of thumb is far from perfect but it's a good starting point. On Intel or AMD hardware, we would recommend against more then eight gigabytes of RAM as we haven't seen it give much improvement but when in doubt, more RAM is safer then less.

If you are running on Sun's Sparc hardware, please contact us and we can work with you to determine the optimal configuration.

Operating system concerns.
ElectroServer runs beautifully on any operating system that can run Java 1.6 or later. In practice though, the choice of operating system dramatically effects the performance you can expect from the server.

Despite the growing popularity, Windows appears to offer the worst performance with ElectroServer by a significant margin while the various Unix options all perform well. For the ultimate in performance and scalability, Solaris on Sparc is the clear choice due to the fact that it supports a great deal more processors than are available with Intel or AMD systems.

So which is best?
There is no "best" hardware/software platform for all situations but if we were to pick a single configuration that is both cost effective and performs great in all common scenarios we would suggest a dual-core Linux box with between two and four gigs of RAM. Increasing the box to a dual-processor/ dual-core system will literally double your performance as well. If growth is important, getting the box with only a single processor installed and half the RAM is a great way to leave room for growth without breaking the bank.

2.2 Operating System

2.2.1 Windows

To install and run ElectroServer 4 on Windows you will need a version of the Java Runtime Environment (JRE) 1.6 or newer. The required JRE is contained within the ElectroServer installer. So you don't need to worry about tracking it down on the Internet, the ElectroServer installer will do it for you.

1. To install ElectroServer, you first need to download the Windows installer from http://www.electro-server.com and save it to a known location.

2. Locate the file you just downloaded and double-click it. Follow the series of prompts to complete installation.

   During the installation process you will be asked a series of questions. The most important of these is if you are installing Professional or Enterprise. If you are not sure which to choose, use Professional. The rest of the prompts will be defaulted for you; only change them if necessary. One key thing to remember is the administrator username/password and the web server IP/port. You will need those to get into the web-based administration panel.

   Once the installer completes, you will have finished installing ElectroServer 4 successfully.

© 2008 Electrotank, Inc.

3. To start ElectroServer 4, click Start > All Programs (or Program Files) > ElectroServer 4 > Start ElectroServer.

   ElectroServer should start up without any problem. The console window will remain open as long as the server is running. If started properly, the last entry in the console window will say "ElectroServer has started successfully". If you install different versions of the server (Registry or Gateway) the command will be named appropriately but the server will start the same.

2.2.2 OS X

ElectroServer is based on the latest Java platform, 1.6. Unfortunately, Apple hasn't gotten around to actually supporting Java 1.6 on either Leopard or Tiger. As soon as 1.6 is available, we will provide an installer for OSX and update this section.

2.2.3 Linux

Installing ElectroServer on a Unix server that supports RPM files is quite simple. The server will deploy the correct version of the Java runtime for you automatically.

1. To install ElectroServer, you first need to download the RPM installer from http://www.electro-server.com and save it to a known location.

2. Pull up a command-prompt on your server.
   You can do this with telnet, SSH, or any number of other techniques.

3. Navigate to the folder where you saved the RPM file.

4. Run the following command, replacing <file name> with the actual name of the file you downloaded:

```
rpm -i <file name>
```

5. That's it!
   The server and JVM will deploy automatically into the "/opt/ElectroServer_<Version>" folder. To start the server, you simply need to execute "`./ElectroServer`" (without the quotes) from within that folder.

   To prevent the server from stopping when you close the console, you need to use the nohup command. It would look like this in that case:

```
nohup ./ElectroServer &
```

2.2.4 Other Unix

Installing ElectroServer on a Unix server using tar.gz is a straight-forward process. You will need to install the latest release of the 1.6 Java Virtual Machine for your platform before you can start the server. Go to http://java.sun.com for details on this.

If you install the VM and the server won't start, you will need to define the environment variable "INSTALL4J_JAVA_HOME" to point to your VM installation folder.

1. To install ElectroServer 4, download ElectroServer 4 and save it to a known location on your hard drive.

2. Pull up a command-prompt on your server.
   You can do this with telnet, SSH, or any number of other techniques.

3. Navigate to the folder where you saved the tar.gz file.

© 2008 Electrotank, Inc.

4. Run the following command, replacing <file name> with the actual name of the file you downloaded:

```
gunzip <file name>
```

This will unzip the file and remove the ".gz" extension as it's no longer "gzipped".

5. Run the following command, replacing <file name> with the actual name of the file without the ".gz":

```
tar xvf <file name>
```

This will extract the server into a sub-directory of your current working directory named "ElectroServer_<version>". To start the server, you simply need to execute "`./ElectroServer`" for Standalone mode, or "`./Registry`" for Registry mode and "`./Gateway`" for Gateway mode from within the folder.

To prevent the server from stopping when you close the console, you need to use the nohup command. It would look like this in that case:

```
nohup ./ElectroServer &
```

2.3 Initial Configuration

2.3.1 Registry and Standalone

Both the Registry and Standalone server instances use the same configuration file, found in the "server/config" folder: ES4Configuration.xml.

This file contains the name of the instance as well as the web server configuration. It's critical that the web server is configured with an available port and IP otherwise you will be unable to access the web administration.

2.3.2 Gateway

Gateway instances use their own configuration file in the "server/config" folder: GatewayConfiguration.xml

Like the registry/standalone configuration, this file also contains a "name". This name is used to identify the instance in the web administrator.

The file also contains an entry for a pass phrase. This value is necessary for the gateway to log into the registry server when it connects. This value must match the same value in the registry server otherwise the gateway will fail to log into the registry.

Finally, the file contains a pair of entries that contain the IP and port of the registry server. These entries need to be the same as the matching settings in the registry for distributed mode to work.

3 Administration

3.1 Web Based Administrator

ElectroServer comes with a full-featured web-based administrator. Almost all server configuration and maintenance is done through this tool. This includes managing extensions, creating server-level plugins, persistent rooms, configuration filters, setting up gateways, managing license files, and much more.

© 2008 Electrotank, Inc.

This section describes the web-based administrator tool and each major section at a high level. Further details on specific fields can be found in the help text on every page of the manual.

To use the web-based administrator ElectroServer must be running. The admin is served up using ElectroServer's built-in web server. The settings for the web server are the only server configuration settings that happen outside of the admin. The web server settings are configured here: [installation directory]\ server\config\ES4Configuration.xml

3.1.1 Accessing the Web-based Administrator

If all server default settings are used during installation, then you can access the admin here: https://localhost:8080/admin/

Once you arrive at the home page you will be prompted to log in. If you left the username and password as the default during installation then they are: administrator / password. You will be immediately warned to change these after logging in.

3.1.2 The Web Admin Sections

3.1.2.1 General Settings

This section of the admin is used for general server settings such as thread settings, timeouts, and licenses.

- Manage Licenses
- Edit Server Settings
- Edit Thread Settings
- Restart Server
- Shutdown Server

3.1.2.1.1 Manage Licenses

By default ElectroServer has a demo license (limit of 25 users, any IP address) installed. Through this section the properties of all installed licenses can be viewed and new licenses can be uploaded. More licenses can be installed, but only one license is active at any time.

Once you purchase a license, install it, then use manage licenses to enable the new license. A server restart is required for a license to become the active license. Inactive licenses can be deleted.

3.1.2.1.2 Edit Server Settings

The Server Settings page allows you to edit general server settings and communication settings. These include things like the server name, the IP/combo that the registry uses to list for gateway connections, and idle disconnect time.

- Server Name

© 2008 Electrotank, Inc.

- File Store Directory
- Server Temp Directory
- Gateway Listener IP. *Note: This refers to Registry mode, not Standalone mode.*
- Gateway Listener Port
- Concurrent User Limit
- User Idle Disconnect Time
- Client Idle Disconnect Time
- Maximum Queued Messages
- Timeout
- Maximum Inbound Message Length
- Maximum Outbound Message Length
- Enable Policy File
- Enable Custom Policy File

3.1.2.1.3 Edit Thread Settings

This page displays the current thread settings. They should only be modified by advanced users that understand what they mean.

- Processor Thread Count
- Minimum Pool Size
- Maximum Pool Size
- Scheduler Thread Count
- Use Executor Filter

3.1.2.1.4 Restart or Shutdown Server

The web based administrator's General Settings menu has options for restarting or shutting down the server. You may also shut down the server manually by closing its command window, then start it again later by running ElectroServer.exe.

3.1.2.2 Security Settings

ElectroServer provides several ways for you to secure your application. This can be done by restricting access to the web-based admin, restricting what users can do using permission sets, and by using secure certificates.

3.1.2.2.1 Manage Permission Sets

This section allows for permissions sets to be managed: create a new permission set or modify an existing one. A permission set is a list of a few dozen actions that a user could possibly perform when logged into the server. The permission set has a name and then a yes or no associated with each action. Permission sets are assigned to users when they log into the server.

To create a new permission set, click **Create a new permission set,** enter a unique name, then click **Yes** next to each permission that users in that set should have. Click **Save** when finished.

To edit an existing permission set, click the name of the set, then click either **Yes** or **No** on

© 2008 Electrotank, Inc.

any permission that needs to be changed. Click **Update** when finished with this permission set.

3.1.2.2.2 Manage Web Admin Users

Through this section new admin-level users can be added, updated, or removed. You can assign major activities that you want the admin user to be able to do such as managing extensions or managing filters.

To create a new web admin user, click **Create a new admin user** and enter each of the following:
- Username
- Full Name
- Password
- Repeat Password

Next, check any or all of the following boxes to give this admin user the privileges needed:
- Can manage users
- Can manage extensions
- Can manage filters
- Can manage persistent rooms
- Can manage stat server

Click **Save** when finished.

To edit an existing web admin user, click the username, then make any modifications needed. Click **Update** to save the changes, or **Delete** to delete the user.

3.1.2.2.3 Change Keystore Password

This page allows you to change the keystore password. The keystore is where ElectroServer stores its certificates.

3.1.2.3 Gateways

When run in distributed mode the server is made up of one to many gateway servers and one registry server. A gateway server manages client connections.

When run in standalone mode the server has a single gateway server and no separate registry server.

3.1.2.3.1 View existing gateways

This section shows all gateways that exist. Click the + symbol on a gateway to view more information. Multiple gateways are only used in distributed mode, which is an enterprise-level feature.

When you first arrive at this page in the admin you will see one gateway running called StandAlone. If you are running in standalone mode, which most developers will, then this is the only gateway you will ever need. To support more protocols or IP/port combos, you can just use this gateway and edit it.

© 2008 Electrotank, Inc.

3.1.2.3.2 Edit a gateway

When you edit a gateway you can add more listeners (IP/port combos) and assign an expected protocol to each. To add a new listener just enter the IP or hostname into the empty field, give it a port, select a protocol (text is the most common), and click save.

3.1.2.3.3 Create a new gateway

This option is only available when running in distributed mode, which is an enterprise feature. To create a new gateway click on the 'Create a new gateway' link. Enter a name for it and the registry pass phrase. Set the number of connections to the registry to something reasonable, like 100. Click save.

3.1.2.4 Filters

ElectroServer provides flexible support for flooding and language filters. Filters are used to help automatically restrict abuse in your applications. The language filters can be used to whitelist or blacklist word usage. The flooding filters can be used to restrict the rapid firing of messages or too many duplicate messages.

3.1.2.4.1 Manage Flooding Filters

Through this section you can view, add, and edit flooding filters. You can also edit the default flooding filter and the default flooding filter settings. Flooding filter settings are used to determine what actions should be taken when a violation is detected. Please see Filters for more detailed information.

3.1.2.4.2 Manage Language Filters

Through this section you can view, add, and edit language filters. A Language Filter is either inclusive or exclusive. If inclusive, all words filtered must be in the associated word list, if exclusive then no words in the associated word list can be found in the string. Language filters contain other properties as well. Please see Filters for more detailed information.

3.1.2.4.3 Manage Word Lists

A word list is just a named list of words. A Language Filter must point to a named word list. You can create or edit word lists through this section.

3.1.2.5 Extensions

The server is highly extensible using what are called **extensions**. An extension is a collection of 1 or more ActionScript or Java files/classes that are used to add more functionality to the server. These extensions can be used as server-level event handlers, such as the Login Event Handler, as Managed Objects which come in handy for things like database connection pooling, or as Plugins. Plugins are run at the server level or at the room level and are frequently used to execute game logic.

Multiple example extensions and plugins are installed in the [installation directory]/server/examples folder.

© 2008 Electrotank, Inc.

3.1.2.5.1 Upload Extensions

New extensions can be added by manually placing them into the [installation directory]/server/extensions directory, or by uploading a zipped extension through the admin. After the extension has been uploaded a server restart is required to access it.

3.1.2.5.2 Create New Server-level Component

New server components can be created. They include event handlers, such as the Login Event Handler, server-level plugins, and managed objects. You select the extension name, give it a name that you will use to access it later, and choose the plugin in the extension that you want to use. Optionally you can also add some XML that will be used to pass information in at start up. A server restart is necessary for changes to take effect.

3.1.2.6 Persistent Rooms

A persistent room is a room that is not removed when the user list is empty. Most rooms are dynamic and are removed when empty.

3.1.2.6.1 Create new Persistent Room

New persistent rooms can be created and published or unpublished at run-time. For information on all of the room properties see Creating a Room.

3.1.2.7 Restart Server

Most of the settings that you change in the admin will not take effect until the server is restarted. You can restart the server using this button. TBI

4 Server in Detail

4.1 Chat

ElectroServer 4 provides a very simple, intuitive, and flexible way for users to chat. There are two primary ways for users to exchange information:

- Public messaging
- Private messaging

4.1.1 Public Messaging

ElectroServer 4 supports rooms. A room is a convenient collection of users that can see each other. A public message is a message sent by a user in a room to that room. All users in that room receive this message.

A public message is made up of two pieces:

- The message body – This is most typically what you would see appear in a chat window.
- An optional EsObject – An EsObject is a highly flexible serialization format supported all over ElectroServer. This provides an easy way for users to send simple or complex data to other users. Example: you want to send a chat message to the room but an audio clip, animation, or other non-text action needs to be performed when the message is displayed. This can easily be achieved by sending extra data via EsObject along with the public message.

© 2008 Electrotank, Inc.

4.1.2 Private Messaging

Private messaging occurs between one and many users. A user can send a message directly to 1 or more other users. These users don't have to be in the same room, or in any room for that matter.

A private message is made up of three pieces:
- The list of users – This is the list of users that should receive the message.
- The message body - is most typically what you would see appear in a chat window.
- An optional EsObject – An EsObject is a highly flexible serialization format supported all over ElectroServer. This provides an easy way for users to send simple or complex data to user users. Example: you want to send a chat message to the room but an audio clip, animation, or other non-text action needs to be performed when the message is displayed. This can easily be achieved by sending extra data via EsObject along with the public message.

4.1.3 Extensions

Public messages and private messages can be captured by server extensions and processed before being delivered. For example, a message sent in English may be captured by the server and translated into Spanish before being delivered to another user. Note: in this example, the translation would need to be implemented by the developer. Other common extensions of chat will expand abbreviations or comicify profanity. See PluginPigLatin in the [installation directory]/server/examples folder.

Extensions can also initialize public and private messages. See Extensions for more information.

4.1.4 Filters

Public and private messaging are affected by language filters and flooding filters. See Filters for more information.

4.2 Filters

ElectroServer provides flexible support for flooding and language filters. Filters are used to help automatically restrict abuse in your applications. The language filters can be used to whitelist or blacklist word usage. The flooding filters can be used to restrict the rapid firing of messages or too many duplicate messages.

4.2.1 Language Filter

ElectroServer supports multiple named language filters and word lists. Language filters and word lists are created and managed in the web-based administrator. A default language filter exists with default actions to take when filtering is applied.

Any language filter can be applied to the following:
- Usernames, during the login process
- Zone and room names, during room creation
- Public messaging, when a message is sent to a room
- Private messaging, when a private message is sent to one or more users

A language filter is identified by its name and can be configured in the following ways:
- Language filter name
- Word list – A list of words to use as either a white list or black list. Words in this list must appear exactly as they are in the list for an action to be taken.
- Root word list – A list of words to use as a white list or black list. Words in this list can appear anywhere for an action to be taken. For instance, the “f-word" contains four characters that should never appear naturally anywhere.
- Language filter type

© 2008 Electrotank, Inc.

  - Exclusive – If this type is chosen, then the word lists are used as black lists. That means that if a match is detected between what is being filtered and one of the words, then an action is taken.
  - Inclusive – If this type is chosen, then the word lists are used as white lists. That means that if a word is found in the filtered string that is not in these lists, then an action is taken.
- Strip HTML – If this is enabled, then HTML is first stripped from the string to be filtered.
- Strip Punctuation – If this is enabled, then punctuation is first stripped from the string to be filtered.

4.2.1.1 Filter Match During Public Message

The following actions are defined by default in the web-based administrator. These only apply to public message filter matches and can be overridden during the room creation process.
- Deliver on failure – If set to yes, then a message that fails the language filter check is still delivered. If set to no, and a failure is detected, the message is suppressed.
- Failures before kick – This is the number of language filter matches that can occur before the user is kicked. When a user is kicked, he is just removed from the room but not the server.
- Kicks before ban – This is the number of times the user can be kicked before he is disconnected from the server and banned from returning.
- Ban duration – This is the amount of time (in seconds) the user is banned.
- Resets after kick – If this is set to yes, then every time the user is kicked, it resets the process as if the user has never been kicked.

4.2.1.2 Filter Match During Login

If the filter finds a match during login, then the user receives a login response informing them that their login attempt was unsuccessful and told that their user name failed the language filter check.

4.2.1.3 Filter Match During Room Creation

If the filter finds a match during the creation of a room, then the room fails to create. The user is informed that the language filter identified text that is not allowed.

4.2.1.4 Filter Match During Private Message

If the filter finds a match during the sending of a private message, then the user is informed of the language filter match. If the **Deliver on failure** option is set to true, then the filtered message is delivered anyway. If set to false, any message that is caught by the language filter is suppressed.

4.2.2 Flooding Filter

Often times users of a chat will cause disruption by using profanity or by sending a high density of messages to the room to hijack the chat window. Through the use of language filters (described in more detail in the Language Filters section) and flooding filters, common abuse behaviors can be suppressed and handled.

ElectroServer allows the creation of custom named flooding filters through the web-based administrator. When creating a flooding filter, certain parameters can be created to detect flooding. The flooding filter of choice is applied to a room during the room creation process. The action to take (such as kicking the user) is also established upon room creation.

4.2.2.1 Filter Parameters

- **Filter name** - The unique name referenced when applying a filter to a room.
- **Maximum duplicate messages** – The maximum number of allowed identical messages sent consecutively to the server.
- **Sliding window duration** – The amount of time (in milliseconds) used when considering if a group of messages is flooding. This is used in conjunction with the parameter for maximum messages in a window.
- **Maximum messages in a window** - The number of messages allowed during the given interval.

© 2008 Electrotank, Inc.

This is best explained by example. Consider that the sliding window duration is set to 1000ms and maximum messages in a window is set to five. If a user ever sends five messages or more within a 1000ms duration, then it will be considered flooding.

4.2.2.2 Assigning a Flooding Filter to a Room

When creating a room via web-based administrator or from a client, a flooding filter can be optionally selected. If a flooding filter will be used, then the following parameters must be specified:

- Flooding filter name (string) – A unique name that identifies a previously created flooding filter.
- Deliver on failure (boolean) – If true and a failure (flood) is detected, the message is delivered. If false, and a failure is detected, the message is suppressed.
- Failures before kick (integer) – The total number of failures allowed by a user before that user is kicked.
- Kicks before ban (integer) – The total number of times to kick a user before that user is banned.
- Ban duration (integer) – The amount of time in seconds to ban the user. If set to -1, then the user is banned until a server reboot.
- Reset after kick (boolean) – If set to true, then the user's offenses are erased and reset after kicked, so the user would never get banned automatically. If set to false then number of times the user has been kicked is stored may eventually lead to a ban (depending on the ban settings).

4.3 Rooms and Zones

ElectroServer supports the concept of rooms and zones. A **room** is a collection of users. A **zone** is a collection of rooms. These simple features allow for a tremendous amount of flexibility when organizing users.

4.3.1 Zones

A zone is a collection of rooms. Every room must exist in a zone. A room cannot exist in more than one zone. A zone is defined by a unique name and has no other properties. A zone is not created by itself. During the room creation, a zone is specified. If that zone does not exist, then it is created.

Users joined to a room in a zone have access to the list of rooms in that zone.

4.3.2 Rooms

A room is a collection of users. Rooms are most commonly used for chatting and playing games, but can be used in any situation where multiple users need to interact. Users can exist in many rooms and many zones. Public messages (chat messages) are sent to a room by users or by extensions. All users in a room receive public messages sent to that room. There are many events that a user can receive by belonging to a room. These events are configured by the user during the room join process and are described below.

4.3.2.1 Creating a Room

There are two types of rooms: dynamic and persistent. Dynamic rooms are more common than persistent rooms.

Dynamic rooms are created at any time by a user or extension. A dynamic room is automatically removed when there are no more users in that room. A zone is removed when there are no more rooms in that zone.

Persistent rooms are created using the web-based administrator and always exist, which is the only differentiator from dynamic rooms.

Whether a room is persistent or dynamic, its properties are the same and assigned during the create process. Some of the properties can be modified at run-time.

© 2008 Electrotank, Inc.

Properties assigned during room creation:

- Zone name (string) - The name of the zone in which to create the room. If the zone does not exist, then it is created.
- Room name (string) – A unique name that can be referenced by users or the server to identify the room.
- Room description (string) – Optional text description of a room viewable by any user in the same zone.
- Room capacity (integer) – The total number of users allowed to join the room. If set to -1 then there is no limit.
- Room password (string) – A password used to secure the room. If a room has a password, then users must provide this to successfully join the room.
- Hidden (boolean) – If hidden, the room does not show up in the room list for a zone.
- Use flooding filter (boolean) – If true, a flooding filter is used in the room and these flooding filter properties are needed. See Flooding Filter for more details.
  - Flooding Filter Name
  - Deliver on Failure
  - Failures before kick
  - Kicks before ban
  - Ban duration
  - Reset after kick
- Use language filter (boolean) – If true, a language filter is used in the room and these language filter properties are needed. See Language Filter for more details.
  - Language filter name
  - Deliver on failure
  - Failures before kick
  - Kicks before ban
  - Ban duration
  - Reset after kick

4.3.2.2 Joining a Room

When a user wants to join a room, the following information must be supplied:

- Room name – The name of the room
- Zone name – The name of the zone that holds that room
- Password (optional) – If the room is password protected, the user joining that room must supply the correct password
- Receive zone list updates (boolean) – If true, the user will receive updates for this zone when something notable changes. A zone update is a minimal update sent when one of the following events happen in the zone:
  - A new room is created
  - An existing room is removed
  - A room's externally viewable details have changed (name, description, capacity, password status, user count, description)
- Receive user list updates (boolean) – If true, the user will receive user list updates for the room. A user list update is fired when one of the following events happen:
  - A new user joins the room
  - A user leaves the room
  - The user starts or stops sending a live audio or video stream to the server. Receiving this event is configurable - see next property
- Receive streaming events (boolean) – If true, the joining user will receive a user list update event when someone in the room starts or stops streaming a live video to the server
- Receive user variable updates (boolean) – If true, the joining user will receive user variable update events when user variables are updated. See also User Variables.
- Receive room variable updates (boolean) – If true, the joining user will receive room variable updates when room variables are updated. See also Room Variables.

© 2008 Electrotank, Inc.

4.4 Extensions

The server is highly extensible using what are called **extensions**. An extension is a collection of one or more ActionScript or Java files/classes that are used to add more functionality to the server. These extensions can be used as server-level event handlers, such as the Login Event Handler, as Managed Objects which come in handy for things like database connection pooling, or as Plugins. Plugins are run at the server level or at the room level and are frequently used to execute game logic.

See the Extensions section for more information on extensions.

4.5 Security

ElectroServer provides several ways for you to secure your application. This can be done by restricting access to the web-based admin or restricting what users can do using permission sets.

4.5.1 User Permissions

When a user logs into the server, he is assigned a permission set. By default the user is assigned the Default permission set. Permission sets can be assigned to users based on custom logic by the use of a login event handler, which is described in detail in Extensions.

ElectroServer exposes more than twenty key actions that can be permitted or denied via a permission set. If a user attempts to perform an action that they are not permitted, then that user is delivered a server error message indicating that access of the feature is denied.

Here is the list of actions that can be permitted or denied through the use of permission sets:
- Create user variables
- Update user variables
- Delete user variables
- Create rooms
- Join rooms
- Leave rooms
- Change room details
- Create room variables
- Update room variables
- Delete room Variables
- Add operators
- Remove operators
- Invoke room plugins
- Invoke server plugins
- Send public messages
- Send private messages
- Kick user
- Ban user
- Kick users (server level)
- Ban users (server level)
- Kick users without operator status
- Ban users without operator status
- Add operators without operator status
- Remove operators without operator status

Note that *without operator status* permissions allow users who are not operators themselves to kick or ban other users. Similarly, the *add/remove operators without operator status* permissions will allow

© 2008 Electrotank, Inc.

non-operator users to give themselves operator status and remove it from other operators. Use these *without operator status* permissions with extreme caution.

4.6 Room Variables

Room Variables are name/value pairs stored on the server and scoped to a room. These variables are seen by users in that room and are accessible by server extensions. Room variables can be created during the room creation process, by users in the room, or by server extensions. The value of a room variable is an EsObject.

When a user joins a room that user specifies if they want to receive Room Variable Update Events. The default is to receive these events. All users that have indicated they want to receive these events are given the list of room variables when they enter and receive events when a room variable is added, updated, or deleted.

4.6.1 Why Are Room Variables Useful?

Room Variables provide a convenient way for developers to accomplish simple things easily. For instance, Room Variables can be set in a room to specify the background music to play, the chat skin to use, etc.

4.6.2 Creating a Room Variable

When a client creates a room variable the following are specified:

- Room Id (integer) – The id of the room in which to create the Room Variable.
- Zone Id (integer) – The id of the zone that owns the room.
- Name (string) – The name of the Room Variable.
- Value (EsObject) – The EsObject value of the Room Variable.
- Locked (Boolean) – If true then the variable cannot be modified until it is unlocked. If false then the variable value can be modified.
- Persistent (Boolean) – If true then when the user that creates the Room Variable leaves the room the variable remains. If false then when the user that created the Room Variable leaves the room the variable is removed.

4.6.3 Updating a Room Variable

A Room Variable can be updated by changing its locked status or by changing its value.

4.6.4 Room Variables and Extensions

A server plugin can create, update, or remove Room Variables with the same capabilities as a client – except for the persistent property. The persistent property can tie the Room Variable to a user.

Plugins can implement a Room Variable event handler so that they can tell when a Room Variable has been modified.

4.7 User Variable Types

In multiplayer games, chats, and other types of interactive applications, it is often useful to attach information to a user that stays with that user at all times. ElectroServer supports that concept through User Variables, User Plugin Variables, User Extension Variables, and User Server Variables.

4.7.1 User Variables

A User Variable is a name/value pair stored on the server and scoped to a user. A user can have many User Variables. The value of a User Variable is an EsObject. A user can create, update, or delete his own User Variables. Server extensions can create, update, or delete User Variables on any user.

© 2008 Electrotank, Inc.

User Variables of a user are seen by other users in the same room. Users in that room can receive User Variable Update events when User Variables are changed. See Joining a Room for more information on configuring the User Variable Update event.

4.7.2 User Plugin Variables

User plugin variables along with the next two are the same except for the scoping. They are not visible to the connected clients. These have a string as the variable name, and any object as the value. User Plugin Variables are scoped to the plugin and associated with the user. So if a room has three plugins, each could have a User Plugin Variable called 'score' with different values.

4.7.3 User Extension Variables

User Extension Variables are scoped to an extension and associated with a specific user. An extension can contain many plugins.

4.7.4 User Server Variables

User Server Variables are scoped to the server and associated with a specific user. A common use for a User Server Variable is storing a user ID that would be used to locate database information about that user.

5 Extensions

When developing multi-user applications, it's often useful or even necessary to push some logic to the server rather than the client. There are many good reasons for doing this but here are a few of the most important:

- Security – The server is significantly more secure than the client.
- Performance – In most cases, the server will provide much higher performance than the client.
- Manageability – Maintaining code deployed on the server is simpler than maintaining code deployed on many clients.
- Resolving Conflicts – Making important decisions in one central location allows the clients to agree.
- Maintaining State – Keeping track of the application state ensures that new users entering can properly view the state.

With all these reasons, it seems that most logic should be pushed to the server, but this isn't the case at all. If you push everything to the server, then it will need to do that same work for each client. Ultimately, while the server might be faster for a few calls, it will get bogged down with many hundreds of thousands of calls. Because of this, it's critical to know what should and should not be done on the server.

Fortunately, there are some rules of thumb on what should be handled by the server that can be used to make this decision easier:

- Anything that affects the game state directly
- Anything that is critical to game-play
- Any area of an application where clients might disagree on the outcome of an action
- Any area of an application where there are concerns for security

Below are a few examples what should be handled on the server:

- In an RPG, determining if the player was hit by the monster and calculating how much damage and remaining health is left to the player
- Picking up an object in the world if the object is important to other players
- Any place where cheating could be done by changing strength, angle, speed or any other variable in the client

© 2008 Electrotank, Inc.

5.1 Types of Extension Components

To support adding this custom logic, ElectroServer 4 includes the concept of extensions. An extension is a grouping of code used to extend the server. Extensions only run on the registry or stand-alone server but never on gateways. There are multiple ways to extend the core functionality of ElectroServer, described in the following sections.

5.1.1 Plug-ins

A plug-in is the most generic and flexible way to add functionality to ElectroServer. Simply put, a plug-in is a piece of code that can be tied to the server itself or a room and can then be called via clients to directly ask it to perform an action. As a special case, room-level plug-ins have the ability to listen to many events that occur in a room such as room variable changes or public messages. These are the bread and butter of multi-player game development.

See the [installation directory]/server/examples folder for multiple examples of plug-ins.

5.1.2 Managed Object Factories

A managed object factory is a special case object that can be used to create objects that need to be tracked over time. Any piece of code in an extension can call a managed object factory and request or return an object to it. An excellent use for managed object factories is accessing a database connection pool. In this case, the connection itself is the managed object and the factory is the pool manager.

See the ConnectionPool example in [installation folder]/examples/ConnectionPool . To use this in your own extension you will need to edit the ConnectionPool's Extension.xml file to provide the correct url, user, and password information for connecting with your database. The database driver will need to go in the extension's lib folder. For example, a mysql database would need both DBPool_v4.8.3.jar and mysql-connector-java-5.0.4-bin.jar in the lib folder, and this section of ConnectionPool's Extension.xml would need to be edited:

```
        <Variable name="poolname" type="string">mysqlpool</Variable>
        <Variable name="url" type="string">jdbc:mysql://localhost:3306/db
</Variable>
        <Variable name="user" type="string">username</Variable>
        <Variable name="password" type="string">password</Variable>
```

Replace the red text with the correct information for your specific database. Merge the `ManagedObjects` section of the Extension.xml with the rest of the parts needed for your extension.

Any plugin or event handler in the same extension will be able to use lines such as these to connect to the database:

```
        EsObject esDB = new EsObject();
        esDB.setString("poolname","mysqlpool");
        Connection c =
(Connection)getApi().acquireManagedObject("ManagedConnectionPoolExample",
esDB);
```

If you rename the handle for the ManagedObject in Extensions.xml, you will need to edit the above line to match.

5.1.3 Event Handlers

Event handlers are used to implement specific code. All of the available event handlers are covered in the following sections.

© 2008 Electrotank, Inc.

5.1.3.1 Login Event Handler

A login event handler is used to provide custom logic during the login process. The login event handler is designed to allow a developer complete control over how a user logs into the server. Multiple login event handlers can be tied to the server at one time. The order in which they are specified dictates their execution order. Each login event handler has the ability to fail the login at any time. A very common use for a login event handler is to look up a username and password in the database before letting the user login to the server.

See **LoginEventHandler** in the [installation directory]/server/examples folder for an example. After creating a login event handler, you will need to use the web admin to add it as a server level component of the extension it is in.

5.1.3.2 Logout Event Handler

A logout event handler is used to execute code when the user logs out or disconnects from the server. Unlike the login event handler, these do not work in a chain. Each one is fired in the order specified but with no ties to their peer handlers. A common use for a logout event handler is to track the time the user spent logged in by updating a database with the exit time.

5.1.3.3 User Variable Event Handler

A user variable event handler is used to listen for changes to user variables. It has the ability to perform an action when a user variable is updated, deleted, or created. A good use for this type of handler is to persist the variable to a database table so when a user next logs in, a login event handler can automatically create the user variables for the user.

5.1.3.4 Buddy List Event Handler

A buddy list event handler is used to execute custom logic when a user adds, removes, or updates a buddy on their buddy list. Like user variable event handlers, these are most commonly used to store the data to the database. TBI

5.1.3.5 Private Message Event Handler

A private message event handler is used to listen for private messages sent between users and is most often used for message logging or custom filtering. TBI

5.1.3.6 Extension Lifecycle Event Handler

An extension life cycle event handler is a special case event handler that is tied to a given extension, not to the server itself. It listens for both the extension start up and shut down notifications. It is generally used to initially load configuration data for the extension and provide that data to other components in the extension. TBI

5.1.3.7 Audio/Video Event Handler

An audio/video event handler is another multi-purpose event handler. It is used to approve someone attempting to publish or subscribe to a video stream much like a login event handler. It also serves as a notification that a stream has stopped playing or that a user is no longer subscribing. TBI

5.2 Extension Structure and Deployment

Extensions follow a common directory structure regardless of what language is used to create them. In fact, different languages can be matched inside of a single extension. The structure looks like this:

```
<extension name>
<extension name>/config
<extension name>/classes
<extension name>/lib
<extension name>/scripts
<extension name>/Extension.xml
```

© 2008 Electrotank, Inc.

Replace with diagram

Everything for an extension goes in a single folder. It's recommended that the extension name for this folder is used, but it isn't necessary as the Extension.xml file (described below) contains the extension name the server will use. Technically the Extension.xml file is the only thing required to create an extension, but wouldn't be a very useful extension by itself.

The **config** folder is optional and is used to contain any configuration files specific to your extension. The contents of the configuration folder will be added to the classloader for this extension. A good use for the config folder is the applicationContext.xml file used for Spring or HBM files used for Hibernate.

The **lib** folder is optional and contains any libraries you need for your extension. In practice, these will be jars and other dependencies. They will all be added to this extension's classloader instance.

The **scripts** folder is optional and contains all the script files needed by the extension. These files can be in any language the server supports.

The **classes** folder is optional and is the equivalent to the scripts folder but for Java classes. Any compiled classes will go in this folder as it will be added directly to the classloader.

5.2.1 Extension.xml

This file defines the contents of the extension and how it should be loaded by the server. The file is broken down into a series of sections that define each component of the extension. The layout looks like this:

```
<Extension>
    <ManagedObjects>
        <ManagedObject>…</ManagedObject>
    </ManagedObjects>
    <EventHandlers>
        <LoginHandlers>
            <LoginHandler>…</LoginHandler>
        </LoginHandlers>
        <LogoutHandlers>
            <LogoutHandler>…</LogoutHandler>
        </LogoutHandlers>
    </EventHandlers>
    <Plugins>
        <Plugin>…</Plugin>
    </Plugins>
</Extension>
```

Each section supports multiple entries. For example, many plug-ins could exist inside the Plugins node as long as each is enclosed in its own Plugin node. The actual data definition of the extension component is the same for all components and would look like this for a Java plug-in (if it had all options enabled):

```
<Plugin>
    <Handle>ExampleJavaPlugin</Handle>
    <Type>Java</Type>
    <Path>com.electrotank.electroserver4.testextension.SpringTest</Path>
    <Synchronized>true</Synchronized>
    <Variables>
        <Variable name="Variable1Name" type="string">variable 1 value</Variable>
        <Variable name="Variable2Name" type="string">variable 2 value</Variable>
```

```
    </Variables>
</Plugin>
```

Each node breaks down as indicated below.

- **Handle**: This is the “handle” of the component and is used as a name for the component's definition. When creating a new instance of a component, the handle is needed for the server to identify which component is being referenced.
- **Type**: The type node defines what language was used to create the component. It currently must be either Java or ActionScript (with other languages soon to follow in future versions).
- **Path**: This is the path to the location of the component’s code. In the case of a scripting-language component, this is a relative path from the scripts directory of the extension. In the case of a Java component, this will be the fully-qualified path.
- **Synchronized** (optional): This node is used to determine if the server should automatically synchronize all access to the component. In the case of Java components, this is often a poor approach because the server must synchronize aggressively due to the fact that it doesn’t know the details of the component, only the entry points. Generally a developer can synchronize the component more efficiently on their own. In the case of an ActionScript component, this flag is crucial because ActionScript doesn’t directly support the ability to synchronize threads.
- **Variables** (optional): All components in ElectroServer 4 support a series of life cycle methods that include being initialized and destroyed. The variables node allows configuration information to be passed into the component’s initialize method directly as an EsObject. Each variable entry will become a single variable in the EsObject. The name attribute will be the name of the variable and the type will be the data type used. All data types are supported with the exception of arrays.

As noted before, every component uses the same structure in the Extension.xml file. Here is an example ActionScript login event handler without variables:

```
<LoginHandler>
    <Handle>ExampleActionScriptEventHandler</Handle>
    <Type>ActionScript</Type>
    <Path>login/AsLoginHandler.as</Path>
    <Synchronized>true</Synchronized>
</LoginHandler>
```

5.3 Extension Loading and Start-Up/Shut-Down Process

Extensions are sophisticated so it’s important to describe how they are loaded and the order in which various components will execute.

Each extension is loaded with its own classloader as part of the overall classloader hierarchy to ensure separation between extensions and the server. On the surface, this may seem an unnecessary complexity, but it provides the most flexibility and convenience for extension developers. For instance, by using a separate hierarchy we are able to ensure that extensions don’t interfere with each other. This allows the server to run multiple instances of the same extension in parallel or even different versions of the same extension at the same time. This also ensures that if one extension uses a different version of the same library there will be no conflict between them.

The server loads all extensions together at start up but in no specific order. Any initialization process that throws an error will prevent the server from starting the external listeners. This is to ensure that a secure system isn’t brought online in an insecure manner accidentally. All start up errors will be in the log files as well as in the web-based administrator.

In practice, this is a very valuable attribute. For instance, let’s assume a multiplayer game uses a database to authenticate players. A login event handler uses a managed object factory to get access to the database connection. The managed object factory uses its initialization method to create the connection pool initially. On start up, a connection string is incorrect and the factory can’t connect to the database. Rather than starting the server and possibly letting users log in without being

authenticated or logging errors every time the login event handler runs, the server simply flags the extension as an error and ensures the external listeners don't start.

The server loads each extension sequentially in an undetermined order. Each extension runs through the following process:
1. The extension life cycle event handler is constructed and initializes.
2. Managed object factories are constructed and initialized in the order they are defined.
3. Server-level plug-ins are constructed and initialized in the order they are defined.
4. Login event handlers are constructed and initialized in the order they are defined.
5. Logout event handlers are constructed and initialized in the order they are defined.
6. Private message events are constructed and initialized in the order they are defined.
7. User variable event handlers are constructed and initialized in the order they are defined.
8. Audio/video event handlers are constructed and initialized in the order they are defined.

Only when all extensions have started successfully does the server begin opening external listeners and allow users to connect. As stated before, any exceptions while initializing will prevent listeners from coming online and will be displayed in the logs and as start up errors in the web-based administrator.

The extension shut down process works in exact reverse of the start-up process with the exception that while errors are logged, they do not stop the process from continuing. The only two things that can cause an extension to shut down are the server getting shut down (registry or stand-alone) or the extension reloading which is described in the following section.

5.4 Extension Reloading Process

Note: This is not working fully yet. A server restart is the only way to guarantee a proper reload today.

Extensions support the ability to dynamically reload if any changes are made to the contents of the extension directory. Reloading an extension is a multi-step process that warrants a more detailed description.
1. The server scans the extension directory looking for changes to existing extensions or newly added extensions.
2. If a change is found, the server rescans the extension on short intervals waiting for changes to settle out. This is to prevent an extension from being deployed while it is still being copied.
3. The extension is copied into the server's temporary directory. If the extension is zipped up currently, it is extracted during the copy.
4. If the extension is a duplicate of an existing extension, the existing extension is shut down via the process described previously. To ensure the integrity of the system and prevent data mismatch problems with the new extension, some users must be disconnected, according to the following rules:
   - Any user that has gone through a login event handler associated with this extension is disconnected. This is to ensure that they will be forced to login through the newly deployed extension.
   - Any user that has had an associated extension-bound user-server-variable via the extension will be disconnected. This is necessary because these users could have had an object associated with them that may not be compatible with the new extension code base.
   - Any user that is in a room and has a room-level plug-in from the extension will be kicked from the room and the room will be destroyed. This is to ensure the room can be re-created with the new code.
4. The new extension is started up using the procedure described above.

5.5 Server API

ElectroServer 4 Java API is used by developers to implement extensions. Any server-side, custom-code in any langauge supported will have access to all methods defined in this interface. Any specific extension component from plugin to event-handler will be able to access a specific

© 2008 Electrotank, Inc.

implementor of the ElectroServerApi by calling the getApi() method on the component. The api will be available to the component by the time the lifecycle "init" method is invoked.

6 Advanced Topics

6.1 Understanding the EsObject

There are many components that make up a multiplayer application, including clients, server extensions, and ElectroServer itself. Clients can be created as Flash applications, Java, BREW for cell phones, or any number other platforms. Server extensions are written in Java or ActionScript. All of these different parts of the overall multiplayer application need to communicate seamlessly for the system to work. To this end, we created EsObject, designed to facilitate the exchange of data between all layers of a multiplayer application.

EsObject is an object that supports a large number of data types. It allows the server and client, or clients and other clients, to exchange simple or complex object structures that retain data types. In addition, EsObjects are used in many places when dealing with extensions and values for room variables, user variables, and user server variables.

One of the most attractive benefits of an EsObject is that it allows for data to be sent across languages cleanly and easily. A plugin written in ActionScript can create an EsObject and send it to a Java client and both sides will treat it consistently. There is no need to perform any messy conversions. For example: A Flash client can create an EsObject, populate it with strings, numbers and arrays of information, and then just send it to a server plugin. The plugin receives this EsObject and can access all of that information.

6.1.1 Where EsObjects can be Used

EsObjects are used in the exchange of custom data and as the value of User Variables and Room Variables. Below are lists of instances when an EsObject is used for data exchange and data storage.

6.1.1.1 EsObjects Used During Data Exchange

- Public and private messages contain an optional EsObject property. This allows for custom data to be attached to a chat message. For instance, which sound to play when the chat message arrives or where to show the message on the screen.
- To log into the server, a client sends a LoginRequest. This request contains an optional EsObject property. In general, this EsObject property would be captured and used by a custom Login Event Handler installed on ElectroServer and written for custom login behavior.
- Plugin-to-plugin and client-to-plugin communication requires an EsObject to carry information.

6.1.1.2 EsObjects Used as Property Values

- User variables allow the storage of information on the server associated with that user in a name/value pair format. The name is a string and the value is an EsObject. As a user travels from room to room, other users see these user variables. See User Variables for more information.
- Room variables are a simple way to create information in a room that may stay there after the player that created it leaves. Room variables are useful in games, and can be used for things like a white board or room welcome message, etc. The value of a room variable is the EsObject. See Room Variables for more information.

6.1.2 Supported Data Types

- String, StringArray
- Boolean, BooleanArray
- Integer, IntegerArray

© 2008 Electrotank, Inc.

- Double, DoubleArray
- Float, FloatArray
- Short, ShortArray
- Long, LongArray
- Number, NumberArray
- Byte, ByteArray
- Char, CharArray
- EsObject, EsObjectArray

6.1.3 Creating an EsObject

To use an EsObject, you must first create a new instance and then set values on it using the provided methods. Here's how to create an EsObject using the client-side ActionScript API:

```
import com.electrotank.electroserver4.esobject.EsObject;
var esob:EsObject = new EsObject();
esob.setString("FirstName", "Jobe");
esob.setInteger("Age",32);
esob.setBoolean("isCool", false);
esob.setFloatArray("FavoriteFloats", [3.141, 2.718]);
```

6.2 Game Manager

When creating a multiplayer game the following question must be answered: How do players match up with each other to play a game? It's likely that a system would be devised that shows a list of games, allows players to join one of those games, and provides a way for a player to create a new game with custom input parameters. A system that does this is a Game Manager (sometimes called Game Matching System). ElectroServer has a highly flexible Game Manager built-in. The primary features of the Game Manager are:

From the client perspective:
- Load game list based on search criteria
- Quick join – Joins any open game or creates one if none exists.
- Join a specific game
- Create a game

From the server perspective:
- Multiple game type support
- Register a new game type
- Tell the Game Manager to create a game with multiple players

From the game's perspective (Plugin):
- The game can lock itself
- The game can hide itself
- The game can update its game details

Note: A multiplayer game (with or without the Game Manager) usually happens inside of a room. A room is a very convenient collection of users that allows the users to interact and so is ideal for nearly all multiplayer games. The Game Manager manages games, and so it manages rooms that happen to have extra properties associated with them. But at the heart of it a game is nothing more than a room.

6.2.1 Game Types

The Game Manager supports an unlimited number of game types. A game type is uniquely identified by a name, such as Chess. A game type also takes a GameConfiguration object. This defines the list of plugins to be used in the room, the default room configuration, default game details, and the default listeners to be applied to the players joining the game (such as receiving user list events).

© 2008 Electrotank, Inc.

Game types are registered by an extension. Typically this is done by a server-level plugin at server startup time. See the server API documentation for exact information.

6.2.2 Game Details

Every game has an EsObject that represents the game details. Default game details are defined when the game type is registered. Additional properties can be supplied by a player that creates the game during the create process. The default game details and those supplied by the creator of the game are merged. If there are property conflicts then the ones supplied by the creator override the default ones.

The game can modify its own game details at any time. The game details are returned in the list of games whenever a list of games is loaded. A game of Texas Hold em poker might have game details that specify the pot limit, number of seats available, and number of people currently playing. These properties are seen by anyone inspecting the game list are completely custom per game.

6.2.3 Loading a Game List

A client loads a list of games from ElectroServer 4 based on search criteria by using the FindGamesRequest. It is really a search that is applied. At a minimum the game type is provided to the server when requesting this search to narrow the results. Additional search criteria include the game's locked property and values of the properties found on the custom game details EsObject. So, you can search for all Texas Hold em games that have a locked status of false, a pot limit of more than 10,000, and maximum seats of 8.

A list of games that match the search criteria is returned (see FindGamesResponse). Each game in the list is represented on the client by a ServerGame class. For each game the following information is available:

- Game ID
- If the game is password protected
- If the game is locked
- Game details.

6.2.4 Creating or Joining a Game

There are three requests included in the client-side API that allow a client to get into a game. They are: QuickJoinRequest, CreateGameRequest, and JoinGameRequest. Sending one of these requests will always get a CreateOrJoinGameResponse. Joining a game is not guaranteed so CreateOrJoinGameResponse contains the success status of the request and possible error information.

QuickJoinRequest
This is the easiest way for a user to get into a game. The client provides search criteria to limit the potential list of games that they can join. If there is an open game that meets the search criteria specified by the client, then the client is joined to that game. If there is no open game then a new game is created and the client is joined to that game. With quick join a client is guaranteed to get into a game (existing or new) as long as the game type specified has been registered.

To perform a quick join, a QuickJoinRequest object is created and populated with information. Some of the information is used for find a game to join and some of it is used if no game is found and a new one needs to be created.

- Game Type (string) – The type of game the client wants to join.
- Game Details (EsObject) – This is an optional property. If it exists and no game is found, then this EsObject is used as the new game's game details.
- Search Criteria (SearchCriteria) – This is an optional property. When the server searches for an open game for the user to join it uses the provided SearchCriteria to limit the results.
- Zone Name (string) – This is used if no game to join is found and a new one is being created. The new game will be created in the specified zone.

© 2008 Electrotank, Inc.

- Password (string) – This is an optional parameter. If specified it will be used against games that the server finds that the user can join. If a new game is created then this password is used in that game.

CreateGameRequest
This request is used to create a new game. First a CreateGameRequest object is created and then it is populated with the following information.
- Game Type (string) – The type of game that the client wants to create.
- Game Details(EsObject) – The game details object that is accessible by the game and seen externally in the game list.
- Zone Name (string) – The name of the zone in which to create the game.
- Password (string) – This is an optional parameter. It should be specified if the client wants to create a password protected game.

JoinGameRequest
This request is used by clients that want to join a specific game. When a game list is loaded the game id for each game is available (so that is how a client would know a game id). To join a specific game a JoinGameRequest is created and populated with the following information.
- Game Type (string) – The type of game that the client is joining.
- Game Id (number) – The id of the game as found in a list of games loaded using the FindGamesRequest.
- Password (string) – This is an optional parameter. If the game specified is password protected then the password field needs to be specified.

CreateOrJoinGameResponse
This response is always received after trying to create or join a game. It contains the success status information. If the join was successful then it contains information about the game. If unsuccessful then it contains an error.
- Successful (Boolean) – True if the client joined a game, false if not.
- Zone Id (integer) – The id of the zone that contains the game.
- Room Id (integer) – The id of the room that is the game.
- Game Details (EsObject) – The EsObject that is used to store the game's details.
- Error (EsError) – If the successful property is false then no game was joined. This error tells you why. See the full list of client errors for the most up-to-date information. But the following three errors are at least some of what you can expect from an unsuccessful join: GameDoesntExist, FailedToJoinGameRoom, GameIsLocked.

6.2.5 The Game's Plugin(s)

It is not required to have a plugin associated with a game type. However, because plugins are so useful, it is likely that at least one per game will be used by the game developer. A plugin can lock or unlock the game. If the game is locked, the Game Manager will not attempt to join a player to that game. Typically a plugin will keep track of the number of players in the game and lock or unlock itself based on that number.

A plugin can get and modify its own game details EsObject.

Plugins have a userEnter event that is fired when a player is joining the room. Plugins have the ability to stop this player from joining. Even if the game manager decides to put a player in the game, the game can still reject the player. The ability of the userEnter event handler to reject a user from joining a room is always there in a plugin – it is not a game manager feature, it is a plugin feature.

6.3 Buddy List

When creating a gaming community, allowing users to have buddies is an attractive feature. When User X adds User Y as a buddy, User X receives events when User Y logs into or out of the server. The end result is that a user can show a list of buddies and view whether they are online.

© 2008 Electrotank, Inc.

6.3.1 Buddies are Session-Based

When a user logs into ElectroServer, many buddies can be added. When the user logs out and then back in again, the buddies need to be added again.

6.3.2 Persisting the Buddy List

If a user's buddy list should be preserved, then ElectroServer must be extended. An extension must be created that allows clients to communicate with it (a server-side plugin will do). The client will tell the plugin to add or remove a buddy to the database. The database used for this can be selected by the developer.

A login event handler extension is also required. When a user logs into the server, the extension will retrieve their buddy list from the database and then add each buddy to that user for this session. This technique gives the developer the ability to do a lot with buddies without being limited to a specific database.

6.3.3 Client Requests and Events

The client requests to add or remove a buddy: AddBuddyRequest, RemoveBuddyRequest. The client receives events when a buddy logs into the server or leaves the server: BuddyStatusUpdatedEvent.

7 Audio and Video Streaming

The Flash player supports a built-in protocol called RTMP. ElectroServer 4 supports full audio/video streaming capabilities via RTMP. As such, Flash clients can stream audio and video using ElectroServer. This includes:

- Clients publishing audio/video for live streams or to be recorded to a file.
- Clients subscribing to live audio/video streams.
- Clients subscribing to audio/video streams that come from FLV files.

In addition to the usual audio/video streaming capabilities ElectroServer provides a convenient room-level event for clients to capture. When clients in your room start or stop publishing a stream other users in that room receive an event reporting this information along with the stream name.

By default, published streams that are set to record save to the [installation directory] \server\video directory. The file name is that of the stream.

To stream audio and video using ElectroServer the client must already be logged into ElectroServer. Then an RTMP connection can be established via a new NetConnection class. The ActionScript API handles creating this NetConnection class instance for you.

Once a NetConnection is established with ElectroServer the client has access to it. The client can use the NetConnection to create NetStreams for publishing or subscribing to audio/video streams. You can use the Flash documentation for all of the details on how to use NetConnection and NetStream.

Here is some basic code that you can use to:
1. Establish a new RTMP connection.
2. Subscribe to an existing stream.

```
import com.electrotank.electroserver4.ElectroServer;
import com.electrotank.electroserver4.message.MessageType;
import com.electrotank.electroserver4.message.event.*;
import com.electrotank.electroserver4.message.request.*;
//
private var input:NetStream;
private var es:ElectroServer; //Assume this has been created and you are already logged in
//
public function init():void {
        es.addEventListener(MessageType.RtmpConnectionEvent, "onRtmpConnection", this);
```

© 2008 Electrotank, Inc.

```
		es.addEventListener(MessageType.RtmpConnectionClosedEvent, "onRtmpCloseEvent", this);
		es.addEventListener(MessageType.RtmpOnStatusEvent, "onRtmpOnStatusEvent", this);
		//
		es.createRtmpConnection("127.0.0.1" 1935);
	}
	public function onRtmpConnection(e:RtmpConnectionEvent):void {
		if (e.getAccepted()) {
			input = es.getRtmpConnection().getNewNetStream();//creates a new NetStream using the existin
			input.play("SomeVideo");
			subscribed_mc.attachVideo(input);//play video in a video symbol
		} else {
			trace("RTMP connection failed");
		}
	}
```

8 Performance and Optimization

8.1 Adjusting ElectroServer's Memory

ElectroServer is a Java application. All Java applications run inside of a "virtual machine". This virtual machine is very gracious of your computer and won't allow any application inside of it to take too much memory by default. ElectroServer can need more memory than the default 64 megabytes that's available as you start to get into higher-load environments. Because of this, you need to tell the virtual machine that it's OK for it to use more memory. This is done via a command-line switch. Specifically:

```
-Xmx<amount>M
```

Simply replace the <amount> with the number of megabytes you want to make available. ElectroServer will use this as the maximum available and will not exceed it.

The following sections cover how to adjust memory on different operating systems and environments.

8.1.1 Windows

8.1.1.1 Executable

If you are running the server as a stand-alone instance, the easiest way to set the command-line parameter is to create a "batch" file. This is a file that Windows can use to run programs with settings.

1. Shut down ElectroServer if it is currently running.
2. Navigate to the server installation folder. This is the folder you chose when you installed the server.
3. Right-click within the folder, choose "New" and then choose "Text Document".
4. Name the file "Start.bat" and say "Yes" when Windows asks you about changing the file extension.
5. Right click on your new file and choose "Edit".
6. Place the following two lines in your file:

```
<type> -J-Xmx<amount>M
pause
```

7. Replace <type> with the name of the executable you use for the server. It might be ElectroServer, RegistryServer, or GatewayServer depending on which you installed.
8. Replace <amount> with the amount of memory you want to use.
9. That's it! Now when you want to support the increased memory you just need to run "`Start.bat`" when you want to start the server.

Here's a specific example, asking for 1024 megabytes and using the standalone server:

```
ElectroServer -J-Xmx1024M
pause
```

© 2008 Electrotank, Inc.

8.1.1.2 Service

If you are running the server as a service, then you need to update the service start parameters.

1. Open the Windows Control Panel.
2. Double-click on "Administrative Tools".
3. Double-click on "Services".
4. Right-click on ElectroServer service and choose Properties. The service will be named ES_<version>_<type>_Service where <version> is the version of the server that you installed and <type> is either Standalone, Registry, or Gateway, depending on what was installed.
5. Add the following line to the "Start parameters" field, replacing <amount> with the amount of memory you want to use.

   ```
   -J-Xmx<amount>M
   ```

6. Press "OK". Next time you restart the server service or the server itself, it will use the increased memory settings.

8.1.2 Unix

Adjusting the server to support more memory while running on Unix involves creating a very simple shell script.

1. Stop ElectroServer if it is currently running.
2. Pull up a command-prompt on your server. You can do this with telnet, SSH, or any number of other techniques depending on the server.
3. Navigate to the server installation folder. For RPM installations, this defaults to /opt/ElectroServer_<version>/
   otherwise it will be where ever you extracted the tar.gz file.
4. Type "`vi Start.sh`" (without the quotes). Yes, this is the cryptic VI editor. Just follow these directions and you will be fine.
5. Press "`i`" to enter "insert" mode.
6. Enter in the following line but replace <amount> with the amount of RAM you want to allocate:

```
INSTALL4J_ADD_VM_PARAMS="-Xmx<amount>M"
```

7. On the next line, enter the command you normally use to start ElectroServer.
8. Press "escape".
9. Type in "`:wq`" (as always, ignore the quotes) and press enter.
10. You should be back and the command-prompt again. Run this command to allow others to execute the file:

    ```
    chmod 755 Start.sh
    ```

11. That's it! Now when you want to support more memory you just need to execute "`./Start.sh`" from this folder to start the server.

Here's a specific example of a Start.sh file that sets the amount of RAM at 1024 megabytes and runs the standalone server in a way that continues to execute even after the command window closes:

```
>nohup.out
INSTALL4J_ADD_VM_PARAMS="-Xmx1024M"
nohup ./ElectroServer &
```

© 2008 Electrotank, Inc.

9 Clients

9.1 ActionScript

ElectroServer 4 Flash ActionScript API is the integration point between the developer and ElectroServer. Developers use the API to:

- Create requests to send to the server
- Capture responses and events sent from the server
- Manage users, rooms, and connections

This section covers the general structure and concepts behind the ElectroServer 4 ActionScript API. It also discusses some differences between the ActionScript 2 and ActionScript 3 APIs, and finishes with a list of commonly used classes and methods. For additional help, check out the tutorials section of the ElectroServer website.

Note: Electrotank does not maintain an ActionScript 1 API for use with ElectroServer 4. If you want to use ActionScript 1 with ElectroServer 4 you can, but you will need to write it yourself.

9.1.1 ActionScript API Overview

A Flash application must first open a socket connection to ElectroServer. Once the socket has been established, the client and server can communicate. How the client and server communicate (the form that the messages take) is the message protocol. The ElectroServer 4 ActionScript API provides a series of useful classes that handle communication with ElectroServer so developers don't need to understand the protocol used. The API also internally manages a range of other classes for things like user lists and room lists.

Both ActionScript 2 and ActionScript 3 APIs are available. These APIs are nearly identical. To use the API in your application, you need to download it and add it to your application's classpath. These APIs are used to create content for the following Flash players and later: Flash 6, Flash 7, Flash 8, Flash 9, Flash Lite 2.1, Flash Lite 3.

There are three important things to know about the ActionScript API, described in detail below.

Note: It is strongly recommended that Flash Develop or other ActionScript development tool that supports Intellisense be used when developing ElectroServer applications. Intellisense (intelligent code hinting) is invaluable when working with any rich API such as the ElectroServer 4 ActionScript APIs. If you choose to code on frames or using the Flash IDE then you may find yourself needing to look up information constantly. You can find Flash Develop at http://www.flashdevelop.org

9.1.1.1 The ElectroServer Class

The most important class in the API is the ElectroServer class which provides useful methods that allows developers to:

- Create one or more socket connections to ElectroServer
- Send requests
- Create an RTMP connection for audio/video streams
- Create a binary connection for transfer of files such as images
- Access the internally managed data such as rooms, zones, users, buddies, and connections

```
import com.electrotank.electroserver4.ElectroServer;
var es:ElectroServer = new ElectroServer();
es.createConnection("127.0.0.1",9875); //ip, port
```

Figure 1: How to create a connection to ElectroServer 4 using the ElectroServer class.

```
var zone:Zone = es.getZoneManager().getZoneByName("Game Zone");
var room:Room = zone.getRoomByName("Lobby 13");
var usersInRoom:Array = room.getUsers();
for (var i:int=0; i<usersInRoom.length;++i) {
        var user:User = usersInRoom[i];
        trace("name: "+user.getUserName());
}
```

Figure 2: How to use the API to get the zone list, find a room in the zone, and trace the user names of users in that room.

9.1.1.2 Requests, Responses, and Events

Nearly every time the Flash client sends something to the server, it is a request such as a LoginRequest with a specific user name, or a JoinRoomRequest to join a room. Some requests will generate a response from the server. For example: A LoginRequest will always cause a LoginResponse. A response is nearly always sent from the server to the client (on rare occasion one is sent from the client to the server). Things that happen on the server that the user needs to know about are called events. User list updates, chat messages, and room list updates are examples of events. Events are always sent from the server to the client.

```
var loginRequest:LoginRequest = new LoginRequest();
loginRequest.setUserName("coolio");
es.send(loginRequest);
```

Figure 3: LoginRequest object is created and then sent to ElectroServer using the ElectroServer class.

9.1.1.3 Event Handling

The API captures client-bound server responses and events. The developer must use event listeners to tie their own code to these events. The MessageType class contains every request, response, and event the ElectroServer API uses. Every request, response, and event has its own class. This makes for a larger but cleaner and less ambiguous API.

```
import com.electrotank.electroserver4.ElectroServer;
import com.electrotank.electroserver4.message.MessageType;
import com.electrotank.electroserver4.message.event.ConnectionEvent;
var es:ElectroServer = new ElectroServer();
es.createConnection("127.0.0.1",9875);
es.addEventListener(MessageType.ConnectionEvent, "onConnectionEvent", this);
function onConnectionEvent(e:ConnectionEvent):Void {
        if (e.getAccepted()) {
                //login
        } else {
                //tell the user login failed
        }
}
```

Figure 4: Importing the ElectroServer class, the MessageType class, and the ConnectionEvent class. A new instance of ElectroServer class is created and used to create a server connection. An event listener is used to capture the ConnectionEvent.

9.1.2 Differences between the ActionScript 2 and ActionScript 3 APIs

The ActionScript 2 and ActionScript 3 APIs are nearly identical. So much so that documentation is only provided for the ActionScript 3 API. The documentation outlines the rare places where the APIs differ. The documentation contains example code written in ActionScript 3. In general, this code is usable in ActionScript 2 applications as long as you convert ‘void’ to ‘Void’ and ‘int’ to ‘Number’.

Here are the places to look out for differences in the APIs:
- The ActionScript 2 API does not support binary protocol.
- The ActionScript 2 API EsObject supports get/set Integer, but uses Number.

© 2008 Electrotank, Inc.

9.1.3 Commonly Used Classes and Methods

The API is extensive, so we've provided some common classes and methods for quick reference.

The full API documentation can be found at
http://electro-server.com/documentation/docs/es4/as3/index.html

9.1.3.1 Commonly Used Classes

ConnectionEvent: Fired when an attempted connection to ElectroServer succeeds or fails
LoginRequest: Used to log in to the server
LoginResponse: Message with success information that is sent back to a client requesting to log in
CreateRoomRequest: Provides a way to create a room with many configurable options
JoinRoomEvent: Fired when a logged-in user joins a room
PublicMessageRequest: Used to create a chat message to send to a room that the logged-in user belongs to
PublicMessageEvent: Triggered when a chat message is sent to a room that the logged-in user belongs to
UserListUpdateEvent: Fired when the user list changes in a room that the logged-in user is in; contains useful information about what changed

9.1.3.2 Common ElectroServer Class Methods

createConnection(ip:String, port:Number): Attempts to create a socket connection between the Flash client and ElectroServer. The ConnectionEvent event is fired with the results of the request.

send(request): When sending something to ElectroServer you must first create a request object (such as LoginRequest), populate it with custom information if needed, and then use the send method to send to ElectroServer.

createRtmpConnection(ip:String, port:Number): An RTMP connection must be created to stream live audio and video using ElectroServer. When the connection has succeeded or failed the RtmpConnectionEvent is fired.

9.2 Errors

The word *error* has a wide meaning here. When working with ElectroServer a client can perform actions that indicate the client did something inappropriate (such as using vulgarity in a public message) or that the server could not do what was asked of it (such as joining a room that doesn't exist).

Most errors are received by the client as a GenericErrorEvent. The specific error that occurred is contained within the event details. For instance, if the user reaches the idle time out (they have been inactive for too long) then they receive a GenericErrorEvent that contains a more specific IdleTimeReached error.

If a user sent a request that requires a response, and that request caused an error, then the error is contained within the response object. For instance, if a user sends a LoginRequest to the server that contains a user name that is already in use, then the LoginResponse object contains a UserNameExists error.

Error List

- **UserNameExists** – The user name used during login already exists.
- **UserAlreadyLoggedIn** – The user trying to login is already logged in.
- **InvalidMessageNumber** – The API sent a message to the server with an invalid message number.
- **InboundMessageFailedValidation** – The API sent a message to the server that did not pass validation.

© 2008 Electrotank, Inc.

- **MaximumClientConnectionsReached** – The user trying to login cannot successfully login because the maximum number of client connections for the server license has been reached.
- **ZoneNotFound** – The zone that the client is trying to work with was not found.
- **RoomNotFound** – The room that the client is trying to work with was not found.
- **RoomAtCapacity** – The room that the client is attempting to join is already at capacity.
- **RoomPasswordMismatch** – The password that the client submitted when joining a room is not correct.
- **GatewayPaused** – The gateway to which a client is connected is paused.
- **AccessDenied** – The client does not have permission to perform the requested action.
- **RoomVariableLocked** – The room variable that a client is trying to update cannot be updated because it is locked.
- **RoomVariableAlreadyExists** – The room variable that a client is trying to create cannot be created because it already exists.
- **DuplicateRoomName** – The room that the client is trying to create fails because the room name is already in use in the specified zone.
- **UserVariableAlreadyExists** – The user variable the client is trying to create fails because it already exists.
- **UserVariableDoesNotExist** – The user variable that the client is trying to update or delete fails because it does not exist.
- **UserBanned** – The client was banned.
- **UserAlreadyInRoom** – The client trying to join a room is already in that room.
- **VulgarityCheckFailed** – Content submitted by the client failed a language filter check. This applies to chat messages, room names, and user names.
- **ActionCausedError** – The client sent a message that caused an error. The client is disconnected.
- **ActionRequiresLogin** – The client has tried to perform an action that requires login.
- **PluginNotFound** – The client tried to communicate with a plugin that could not be found.
- **LoginEventHandlerFailure** – An exception happened in a LoginEventHandler during the login process.
- **InvalidUserName**– The client specified a user name that does not exist.
- **ExtensionNotFound** – The client tried to use an extension that was not found.
- **PluginInitializationFailed** – The client created a room with a plugin and that plugin failed to initialize.
- **FloodingFilterCheckFailed** – The client sent too many messages that were detected as flooding.
- **UserNotJoinedToRoom** – Client attempted to interact with a room in a way that only a user in that room can.
- **ManagedObjectNotFound** – Client attempted to communicate with a ManagedObject that was not found.
- **IdleTimeReached** – The client has been idle (no activity) for longer than the allowed idle time. The client will be disconnected.
- **ServerError** – The client performed an action that caused an error.
- **OperationNotSupported** – The client called a plugin and that plugin did not implement the request event.
- **InvalidLanguageFilterSettings** – The language filter settings supplied during the room creation process were not valid.
- **InvalidFloodingFilterSettings** – The flooding filter settings supplied during the room creation process were not valid.
- **ExtensionForcedReload** – See Extension Reloading Process.
- **UserLogOutRequested** – The server is requesting that the client log out.
- **OnlyRtmpConnectionRemains** – The socket connection has been lost and only an RTMP connection remains.
- **GameDoesntExist** - The game specified by the client does not exist.
- **FailedToJoinGameRoom** – The game that a client is attempting to join fails.
- **GameIsLocked** – The game that a client is attempting to join is locked.
- **PublicMessageRejected** – The server captured a public message and decided to kill it. This error

is sent back to the sending client.

© 2008 Electrotank, Inc.