

Gridborg Server – Programming Tutorial

Uniqall, Inc.

June 30, 2006



Contents

1	About This Document	2
2	Talking With the Gridborg Server	2
2.1	Example 1: basic application-server communication	2
3	Messages	2
3.1	Message syntax	3
3.2	Example 2: sending and receiving messages	4
4	Commands and Responses	4
4.1	Responses are Synchronous	5
5	Events	5
5.1	Example 3: using the SocketClient class	5
6	Introduction to the Gridborg Server Model	5
6.1	Logging-in, Logging-out, Sessions	6
6.2	Resources	6
6.3	Calls	6
6.4	Media Playing and Recording	7
6.5	Audio Routing	7
6.6	Bottom Line	8
7	Programming	8
7.1	The Complete Example: simple voice-mail application	8
8	Change History	9

1 About This Document

This document is intended as a tutorial-style guide into developing applications for the Gridborg Server.

An archive file containing example code written in Java, Visual Basic .NET and Python languages can be downloaded from the Community section of the Uniqaall web site. The archive file also contains ready-made code that you can directly use in your program.

To be able to follow through this tutorial and at the same time try out provided example code (recommended), a running instance of Gridborg Server should be made available, with at least one Application Identity enabled for this use. If you have not yet installed and configured your copy of Gridborg Server, please do so. The “Gridborg Server - Installation and Configuration Guide” will aid you in this task and will also provide information on various terms that are used in this tutorial.

We will assume that Gridborg Server is running on a local machine (localhost), and that the control port is left in its default value (1234). An application identity called “user1” has been created, and its password is set to “user1pass”.

The “Gridborg Server - Protocol Reference” is the definite formal description of the Gridborg Control Protocol and all commands and events that exist. It is wise to have it near during any Gridborg-related programming activity.

2 Talking With the Gridborg Server

The Gridborg Server and the application talk to each other using a simple textual message-based protocol. Messages are carried from the application to the server, and the other way around, over a single TCP connection which stays open during the entire session.

To be able to talk to the Gridborg Server the first thing the application should do is to open a connection to the server’s control port. At that port the server is waiting for incoming connections. When the connection with the application is established, the server is ready to receive messages.

2.1 Example 1: basic application-server communication

In this example, after the application has established a TCP connection with the server, it sends one line of text to the server and receives two lines from the server, printing them to the standard output.

Output from the example program should look something like this:

```
#Gridborg Server 1.1.8 ready  
ROK 1 1 8
```

The first line is a comment string that the Gridborg Server sends to all applications upon connection establishment.

The second line is a response message to the “ProtocolVersion” command message that we sent to the server.

3 Messages

Flow of text between application and server is separated into individual lines that are terminated with the CRLF sequence (`\r\n`). Each line of text can contain a message part and a comment-string part. The comment-string starts after the first occurrence of the `'#'` character and spans to the end of the line. For example:

```
#Gridborg Server 1.1.8 ready
ROK 1 1 8
RFAIL #Missing parameter (resource ID)
```

The first line contains only the comment-string part, the second line only the message part, and the third line contains both.

In the previous section we already mentioned two message types: command and response messages. A complete message-type break-down goes like this:

- messages sent from the application to the server are command messages
- messages sent from the server to the application that start with the 'R' character are response messages
- messages sent from the server to the application that start with the 'E' character are event messages

Applications send command messages to the server. Response and event messages are always sent from the server to the appropriate application.

It was said that messages are textual. A couple of rules do apply:

- only printable seven-bit ASCII characters (values 32 to 126 decimal) are permitted in the message, others are forbidden
- characters ' ' ("space"), '=' ("equal sign") and '%' ("percent") are reserved
- the "space" character separates individual message elements
- the "equal sign" character separates name and value parts of a named parameter
- the "percent" character is used to encode forbidden or reserved characters into the message (or any other, for that matter)
- the '#' ("number sign") character is not reserved, but is used to end the message part and start the comment-string part

In the previous example we received one comment string and one response message.

Comment strings are intended only for human eyes, and the application program should probably ignore them altogether, except maybe by logging them. For example, the greeting message that the Gridborg Server sent when the application connected contains the program's version. It would be a mistake for the application to try to extract the server's version from that message (the example program gives us a hint on how to get server's version properly).

The response message from the example's output consists of four elements, which are individually separated by the delimiting "space" character. Individual elements are, in order of appearance, "ROK", "1", "1" and "8".

3.1 Message syntax

The previously mentioned message rules specify reserved characters which have a special role in a message. Those are: space, equal sign and percent sign.

The space character separates individual elements, breaking down a message to a list of message elements. Each message contains at least one element, and that element holds a special role in all message types.

For command messages, the first element specifies the command name. For event messages, it specifies the event name. For a response message, it specifies the response type – success or failure.

Other elements in a message are said to be "message parameters". We separate message parameters to positional and named parameters. Message parameters that contain the special equal sign are considered

named, where the part before the equal sign specifies parameter name, and the part after the equal sign is a parameter value. Parameters that do not contain the equal sign are positional. Positional parameters are counted from the leftmost to the rightmost.

Take for example the following command message.

```
PlayFile 1 greetings.vap Type=VAP_ULAW Index=42
```

“PlayFile” is the first parameter, the special one, which specifies the command name – PlayFile in this example. “Type=VAP_ULAW” and “Index=42” parameters contain the equal sign, and are therefore considered to be named parameters. “Type” and “Index” are parameter names, while “VAP_ULAW” and “42” are values.

Parameters “1” and “greetings.vap” are positional parameters, and, counting from left to right, “1” is the first one while “greetings.vap” is the second one.

The same command message could as well be written like this:

```
PlayFile 1 Index=42 greetings.vap Type=VAP_ALAW
```

Nothing is changed – “1” is still the first positional parameter, and “greetings.vap” is the second one. But it is a convention to first specify all positional parameters, and then specify named parameters.

In command messages positional parameters are also called required parameters, while named parameters are called optional parameters.

The final special character to explain is the percent sign. It is used to encode a character that could not be placed verbatim in a message because it is either reserved or forbidden, by specifying its ASCII value instead.

This is done by following the percent sign with exactly two characters from the hexadecimal alphabet. For example:

```
PlayFile 1 hello%20world.wav
```

The file on the disk is called “hello world.wav”, and contains a reserved space character. The ASCII value of the space character is 20 hexadecimal.

The hexadecimal alphabet includes digits from 0 to 9, and letters ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, and their upper-case equivalents.

This way you can include a character of any value between 0 and 255 in the message.

3.2 Example 2: sending and receiving messages

The downloadable archive contains two classes – MessageParser and MessageFormatter, that provide a convenient way to parse messages received from the Gridborg Server and construct messages that are to be sent to the server.

The second example program demonstrates their use.

One object of the MessageFormatter class is used to create command messages that are sent to the server. At first, individual values are added to the object, and then a properly formatted and encoded message text is obtained from the object. This text is then written to the server.

Each line of text received from the server is separated into a message and a comment part. For the message part an object of the MessageParser class is created and used to decompose the message into individual elements.

4 Commands and Responses

Command and response messages are tied together – for each command message that the Gridborg Server receives from one of its applications, it sends back a response message to that exact application.

There are exactly two types of response messages:

- positive one, confirming that the command was understood by the server and that the appropriate action for that command will be taking place
- negative one, indicating an error, in that the server was either unable to understand the command message, or the command message requested an illegal operation

Positive responses, also known as ROKs, besides signalling a command success, often return one or more response values embedded in the message parameters. The `GetVersion` message that we used in earlier examples returned three numbers.

The number of possible response values and their respective types are specified for each command separately. For the `GetVersion` command, the “Gridborg Server - Protocol Reference” document specifies that the positive response returns exactly three integer numbers, that represent the major, minor and build number of the running Gridborg Server.

4.1 Responses are Synchronous

What it means when we say that responses are synchronous, is that the application will receive response messages from the server in the same order that it sent command messages to the server.

If the application sends a number of command messages in succession, then the first response message received will relate to the first command message sent, the second response message to the second command message, and so on. That does not eliminate the possibility that some other message-type can find its way in-between consecutive response messages, but all response messages will be received in the right order.

5 Events

Event messages generated by the Gridborg Server can be a direct result of some command that the application issued, or a notification about some external event, like a reception of an incoming call or a user pressing a key on the telephone keyboard.

We say that event messages are asynchronous. That means that they can be received at any time, and the application should always be listening out for them if it wants to be able to react to them in a timely fashion.

5.1 Example 3: using the `SocketClient` class

In this example the use of `SocketClient` class is shown.

The asynchronous nature of events gives us the requirement to eagerly read on the incoming side of the connection to the server. For this we use a separate thread, whose role is to read everything coming in from the socket and separate synchronous response messages from asynchronous event messages.

When an asynchronous event message is received, it is put in a special queue where a separate thread can get it and handle it. Moving all event handling to a separate thread is necessary so the event-handling code does not block the entire communication with the server.

6 Introduction to the Gridborg Server Model

The “Gridborg Server - Resource Model Description” document explains in great detail the Gridborg Server’s underlying design and application development model. While this chapter is a gentle introduction to those topics, that document explains them in depth.

In following subsections, example communication between application and server is shown. Lines starting with the '>' character are traffic sent from the application to the server, while lines starting with the '<' character are sent from the server to the application.

6.1 Logging-in, Logging-out, Sessions

Now that we know all about commands, responses and events, we can begin using the Gridborg Server in some real-life scenarios.

Before the application can start to create and use Gridborg Server's resources, it must assume one application identity that is configured on the server, or colloquially said, it must log-in to the server.

The Login command is used just for that.

```
> Login user1 user1pass 2 1 1
< ROK 1
< ESessionCreated 1
```

If the Login command is successful, an integer number is returned in the response. That number is what is called a Session Identifier, or Session-ID. When the application logs-in to the server, a new session is created for that connection. Each created session is identified by a Session-ID that is assigned to it at the time the session is created with the Login command.

One application can have more than one connection to the server. Each connection, after logging-in, creates a new session on the server.

As a side-effect to the creation of a new session, the ESessionCreated event is also generated with the same Session-ID as the parameter.

The first positional parameter of every event is the Session-ID of the session that owns the resource that generated that event (more on resources in the next section). The reason for this is that it is possible for one application to receive events originating from two or more different sessions.

6.2 Resources

After the application has created a session for itself, it can start creating and using various resources. You can think of resources as elements of Gridborg Server's functionality.

The ResourceCreateXXX commands create resources of various types. The integer returned in the ROK response carries the Resource-ID of the newly created resource. Resource-IDs are similar to Session-IDs – they uniquely identify each created resource. All commands that work on a specific resource take the Resource-ID of that resource as the first parameter.

```
> ResourceCreateFrontEnd
< ROK 12
> ResourceCreatePlayer
< ROK 13
> ResourceCreateRecorder
< ROK 14
< EResourceCreated 1 12
< EResourceCreated 1 13
< EResourceCreated 1 14
```

To be precise, a resource should not be considered created until the EResourceCreated with the appropriate Resource-ID is received.

6.3 Calls

When we are talking about calls what we really should be talking about are Front-end resources.

Ah, yes, the Front-end resource type. Front-end resources are your connection to the telephony world, far and wide. Front-end resources enable you to make and receive telephone calls, which is probably the reason why you are using the Gridborg Server in the first place.

To be able to receive or make new calls, the front-end resource must be free of any existing call – only one call per front-end resource is possible.

```
< ECallIncoming 1 12 4ad21e16-6545-da11-891f-0050bac956fc
> CallAnswer 12
< ROK
< ECallConnectionEstablished 1 12
< ECallCleared 1 12 EndedByRemoteUser
```

In this example our front-end resource received an incoming call. We decide to answer it using the CallAnswer command. After we are successfully connected with the remote party, a ECallConnectionEstablished event is sent. After a while the remote user decided to hang-up on us, on which we were notified with the ECallCleared event.

We can also make outgoing calls, as seen in the next example.

```
> CallMake 12 borko@pixie.uniqall-local.net
< ECallOutgoing 1 12 borko@pixie.uniqall-local.net 72cc31f8-0646-da11-8782-0050bac956fc
< ECallConnectionEstablished 1 12
```

Here we initiated a outgoing call on our front-end resource. The other side picked-up the call, and when the connection was established the appropriate event was generated.

6.4 Media Playing and Recording

Player and Recorder resource types are basic media-processing resources in Gridborg Server. Player resources play audio data read from disk files or from audio streams. Recorder resources record audio data to disk files or to audio streams. If it looks simple, it's because it is.

```
> PlayFile 13 hello-world.wav
< ROK
< EPlayerStarted 1 13
< EPlayerStopped 1 13

> RecorderStartToFile 14 recorded_message.wav
< ROK
< ERecorderStarted 1 14
> RecorderStop 14
< ROK
< ERecorderStopped 1 14
```

You may wonder what was recorded in the previous example? The answer is – nothing. Also, where did the sound played on the player resource go? Answer – nowhere. The next section will give the explanation.

6.5 Audio Routing

Individual resources by themselves are of little value. Their power comes into play when you combine them together by connecting their audio streams. Until connected, the Recorder resource can record only silence. Until connected, everything the Player resource plays will fall on proverbial deaf ears.

A resource can be an audio source, audio sink, or both. Player resources are examples of audio sources. Recorder resources are audio sinks. Front-end resources are both.

The `AudioSend` command is used to establish an audio route between resources. Let's see what happens when we connect resources like this:

```
> AudioSend 13 12
< ROK
> AudioSend 12 14
< ROK
```

From now on until the audio routes are disconnected or resources are deleted, everything the Player resource plays from a disk file or from somewhere else will be routed to the Front-end resource, which will transmit it to the remote party. Likewise, everything that the Front-end resource receives from the remote party will be routed to the Recorder resource, so it can record it, if necessary.

There are numerous ways in which you can create a mesh of interconnected resources. The “Gridborg Server - Usage Scenarios” document shows the most common application scenarios.

6.6 Bottom Line

Resource types and commands mentioned in this chapter do not consist the full arsenal of Gridborg Server's features. There are many ways in which you can use and abuse the Gridborg Server, but the underlying idea is the same as the one we presented here.

7 Programming

Now that you are familiar with both how to talk with the Gridborg Server and how its resources are used to obtain the required functionality, you can start writing your own applications.

Besides your own programming style, the type of the service you are creating will probably be the force influencing how you design your application. For IVR-style service (interactive voice response), chances are that events will be what's driving the entire application.

Don't forget that the computer where the Gridborg Server is running could be some computer other than the one that is running your application. This issue comes into play if both Gridborg Server and the application are accessing same files. Example of this can be an application examining the file that was recorded by the server's Recorder resource.

7.1 The Complete Example: simple voice-mail application

The `BasicChannel` class from the downloadable archive builds upon the `SocketClient` class. The idea behind it is to provide a simple input/output channel for a single telephone call, suitable for single-user IVR applications.

After the connection to the server is established, the `Login` method is used. If the login is successful, a `BasicChannel` object will proceed to create one instance of front-end, player and recorder resources, establishing audio routes between them. After all resources are created, and the object is ready for use, the readiness will be signalled by invoking the `OnChannelReady` handler method. The `BasicChannel` class also wraps some common commands into methods. Here `MessageFormatter` class is used for formatting commands and `MessageParser` class is used for parsing responses.

The `AnsweringMachine` class is an example of an almost-complete program for the Gridborg Server. Built on top of the `BasicChannel` class, it creates a simple implementation of the familiar voice mail service.

When the call is picked up by the machine, a greeting message is played. After the beep tone which marks the recording start, a recorder starts to record the message to a file. During playback of the greeting message the user has the opportunity to enter the “administrative mode” by pressing the '#' key and entering the password sequence. Once in the administrative mode, recorded messages can be played back or deleted.

You'll notice, in the AnsweringMachine class constructor, that directory paths are specified separately for the application and for the server. If server and application are running on different computers, each must access files and directories in its own way. For example, the recording directory for the application could be specified as `//networkdisk/rec`, and for the server as `/mnt/networkdisk/rec`.

8 Change History

Change History

Date	Description
2005-11-10	Last-minute changes.
2005-11-11	First published version.
2006-06-30	Added Python code examples.