GGZ Gaming Zone Design Specification

The GGZ Gaming Zone developers

ggz-dev@mail.ggzgamingzone.org

GGZ Gaming Zone Design Specification
by The GGZ Gaming Zone developers

Design specification for the GGZ Gaming Zone. This document covers the internal server architecture and the client/server communication protocols.

Revision History

Revision $Revision: 8345 $ $Date: 2006-07-10 10:00:10 +0200 (Mo, 10 Jul 2006) $

Table of Contents

Introduction by Brent M. Hendricks

At some point during the development of `NetSpades`, I realized that a major rewrite was in order. I wasn't happy with the client/server interactions, and I wanted to add some features which just wouldn't work within the current infrastructure. So I began to formulate grandiose plans for `NetSpades v5`, but continued to work on the 4.X series. A little later, I began to contemplate adding the ability to play Hearts as well as Spades. But why stop there? Why not develop a general framework for playing networked games? And so my plans have developed into what I called `NetGames`.

Just before I was going to announce my new project on SourceForge[1], I happened across Rich's announcement and plans for an online gaming system. It was to be in the same vein as Microsoft's Internet Gaming Zone, so he named it the `Gnu Gaming Zone`(GGZ). His goals and plans were so similar to mine that I suggested we merge projects and collaborate. He readily agreed, and the GGZ was born.

Notes

1. http://www.sourceforge.net

Chapter 1. Design Requirements

There are many planned features for GGZ. Some will take longer than others to implement. Some may be discarded at a later date. The following ones are available already (not ordered yet):

- Password-based user accounts
- Ability to leave and then rejoin game
- Ability to reserve games for specific users
- Capability for users to chat without resorting to second port/socket
- Basic infrastructure from game developers to write network games without having to worry about connections/login/user accounts/etc.
- Multiple "rooms" in which to play games. These rooms may reside on different servers, transparent to the user.

Planned features include (but are not limited to):

- Persistent user statistics
- Dynamic run-time configuration of server
- Ability to add/remove game types from server without having to recompile

Chapter 2. Design Overview

> **Warning**
>
> This chapter is horribly out of date. We will be fixing it soon

There are four parts which comprise the server side:

- Main GGZ server (called ggzd)
- Individual game servers
- Run-time config utility programs
- Meta server

We will discuss each of these in turn, but first we'll look at the overall architecture.

Figure 2-1. Server Architecture

The GGZ server will handle incoming connections, manage the user database, and keep track of all of the games being played (referred to as game tables). Clients are always in direct communication with control.

Note that ggzd will not handle the specifics of how to play any particular game. That logic is contained in the individual game servers. It is expected (and hoped!) that game developers will write their own games servers for use with GGZ. GGZ will attempt to provide a simple framework for writing network games in which developers need not worry about connections or user logins or maintaining statistics. All of that will be done by GGZ. Game developers should only have to concern themselves with gameplay.

The third item is more loosely connected. We will provide some sort of run-time configuration utilities for GGZ, so the main server will not have to be restarted (or worse.. recompiled!) in order for various options to be changed. The first such utility is ggzduedit, with which the user database can be edited.

Some options may include:

- Location of game servers
- Set auto removal of inactive users
  - Set inactivity threshold
- Set auto clearing of statistics
  - Set clearing interval
- Log level (level of detail in server logs)

The GGZ server (ggzd)

This is the main brain for the server side of GGZ. It handles client logins and new user registrations. It manages option negotiation with the clients, and launches new game sessions. It maintains a list of running game sessions and keeps a database of

win/lose statistics for each user. It coordinates games, users, and databases, and is responsible for interacting with the client, the running games, and the config utility.

Several possible designs were being considered before settling on the current one. It is possible that as GGZ develops this design will change as well. Since ggzd must communicate with multiple parties (game tables, users, etc.) it was decided to use a multi-threaded concurrent server where each connection (be it user or game table) gets its own thread. This avoids the situation where ggzd is servicing a request and therefore cannot handle any incoming connections or other requests. Brent chose threads rather than forking child processes because threads have a smaller overhead and it is easier to share memory between threads than between processes.

Every time a new user connects, ggzd creates a new thread to handle all requests for that user. This thread is known as the player handler. If the user decides to launch a new game table, the player handler creates a thread to handle all requests from the game table. This new thread is known as the table handler. The table handler waits until enough players have joined the table and then forks a process, known as the game table process, in which to run the game server. (The reason game servers are not run within a thread is so that game developers not be required to worry about writing thread-safe code.)

During the course of the game, the player thread for each player acts as a liaison between the player and the game table, passing requests back and forth transparent to the player and the game server. When a player logs out, the player handler thread is destroyed.

Interactions between ggzd and running games

Quite a few game servers will be provided with GGZ, but it is hoped that others will write game modules, and either submit them for inclusion in the GGZ package, or maintain and distribute them separately. The following API describes how ggzd will interact with the game processes.

> **Note:** All interaction is enclosed within the ggzdmod library. Game developers should use it (or any of its wrappers) so they don't have to deal with the protocol in use directly.

Four types of data are exchanged between ggzd and game servers:

- chr: a 1-byte signed char
- int: a 4-byte signed integer in network byte order
- str: a multibyte null-terminated string preceded by its length (including null-termination) as an integer.
- fd: a file descriptor passed via sendmsg() along with a single byte of dummy data

The following is a complete list of messages between the game module and the control section of the server. Again, I have chosen to intersperse game module requests with the corresponding control responses.

Please note: the following is not written stone, merely a list of ideas.

```
REQ_GAME_LAUNCH
int: number of seats at table
sequence of
int: seat assignment (-1 for OPEN, -2 for COMP, -3 RESV)
str: name of player (if assignment >=0 or == RESV)
fd:  file dscriptor of player (if assignment >= 0)
RSP_GAME_LAUNCH
```

```
chr: success flag (0 if OK, -1 if error)

REQ_GAME_JOIN
int: seat number
str: name of player
fd:  file dscriptor of player
RSP_GAME_JOIN
chr: success flag (0 if OK, -1 if error)

REQ_GAME_LEAVE
str: name of player
RSP_GAME_LEAVE
chr: success flag (0 if OK, -1 if error)

MSG_GAME_OVER
int: number of statistics
sequence of
int: player index
int: number of games won
int: number of games lost

MSG_LOG
int: log level mask
str: log message

MSG_DBG
int: debug level mask
str: debug message
```

Interactions between ggzd and config utilities

Since the server runs non-interactively in the background, there needs to be some run-time configuration tools so that server options may be changed without restarting. Updates are either executed indirectly (e.g. when removing a player from the database), or explicitely (e.g. when changing some configuration parameters and sending a hangup signal to ggzd). The latter one will be steered by a configuration utility as well at some point.

Messages between ggzd and the config utilities might include:

- Request available game types (loaded modules)
- Add/Remove Game types
- Remove users
- Request list of active games
- Clear Player statistics
- Modify logging

Individual Game Servers

As described above, game servers run in their own processes, and are responsible for handling the gameplay of a particular game.

There are three possibilities for game server design.

- Compiled in. The game table process calls a startup function which begins execution of the game server. This scheme has the benefit that the server has access to ggzd's data structures at the time the process was forked. Communication between the game table and ggzd can be via pipes or a socketpair. The problem with this scheme is that to change which games are offered by a particular server requires a recompile. Not good.
- Dynamically loaded modules. Similar to the above except that game server exist as loadable modules which may be inserted and removed at runtime. This allows for adding new game types without a recompile. Downside is that it requires both game developers and myself to know how to deal with loadable modules.
- Exec() separate program. In this scheme, GGZ acts much like the inet daemon by handling the connections and then doing a fork()/exec() to launch the game server. This scheme also allows for adding new games at runtime and has the bonus effect that game servers can be written in other languages that C.

The current design requires game servers of type three. It is possible however, that in the future GGZ will allow dynamically loaded server modules as well.

Once the game server is running it is necessary that ggzd pass it some required information such as player names and file descriptors. This communication occurs over a pipe or socketpair which is established prior to the forking of the process. The interactions between game modules and the control section are listed above in section 3.1.1.1.

Config Utility

There is currently one such tool, namely ggzduedit. It can be used to access the player database, edit permissions and add and remove registered players.

Key Subsystems

While the architecture of the server is divided into the aforementioned four parts, there are a few "subsystems" which are necessary.

Options Parser

This is a two part system. One to parse the command-line arguments, and one to parse the configuration file. Options specified on the command line should have a higher precedence than those in the config file. Additionally, an alternate config file may be specified on the command line.

At the present, the popt library is used for command-line parsing.

The configuration file parsing is a three-phase process which bootstraps itself from a dark, dreary and empty server to one supporting multiple chat rooms, each of which can host one specific game:

- Phase One: Read the ggzd.conf file from SYSCONFDIR. The file to read can be determined at runtime with the *--file=/path/to/conffile* option. This feature is most notably used when testing a new configuration.

The same line parser is used throughout the configuration process. It is relatively simplistic, much like strtok(), but processes an entire line in a single shot. It then sets two module variables 'varname' and 'varvalue' to the appropriate contexts from the configuration line. The file parsers can then use these strings to configure various run-time variables.

- Phase Two: Scan all the game description files and parse those which have been signaled as in use by the main configuration file.
- Phase Three: Scan all the room description files and parse those which have been signaled as in use by the main configuration file. These must be processed after the game files, as each room will host a game which is already loaded by phase two.

The actual contents of the configuration files is discussed in the server administration documentation.

Login/User Database

The server will need to store a database of user ID, name, password and permission at the very least. This system must allow searching by name or ID, and allow for easy addition/deletion. GGZ can use libdb for this. A SQL database is possible too but probably not necessary for most installation. Both PostgreSQL and MySQL are supported at this point.

MOTD System

The message of the day is read at initialization time from a file pointed to by the MOTD configuration file option. The file can consist of up to 80 lines (this is configurable at build via MAX_MOTD_LINES).

The message of the day file can contain % specifiers which are replaced by the server before sending to the client. The following codes are supported:

- %a - Server administrator's name
- %C - Server cputype (according to last kernel build)
- %d - Current date on server
- %e - Server administrator's email address
- %g - Current number of game tables
- %G - Current number of game tables with free seats
- %h - Server's hostname
- %o - Server's OS name
- %p - Port number for this server
- %t - Current local time on server
- %u - Current server uptime statistic
- %U - Current numbers of users on system
- %v - Version of running server

Player Statistics

Since the game modules are dynamic, it makes sense to store the statistics on a per-game type basis, rather than on a per-user basis. Statistics are currently handled by ggzdmod, independent from ggzd. Several types will be supported eventually, some of them are already implemented, e.g. win/loss or ELO.

Module loading

Not written

Data Structures

There will be an array of *game_t* structures on the server. This array will be initialized at startup, and may be changed when dynamic loading of new game modules occurs. The index into this array is referred to as the game type index.

```
game_t {
str: short string for name of game (16 chars?)
str: long string for description (256 chars?)
fnc: pointer to function for launching game
chr: allowable player numbers (2^num)
chr: allow computer players (1 for yes)
int: sizeof options struct in bytes
chr: enabled flag (1 if playing this game is enabled)
}
```

There will be an array of *game_run_t* structures representing running games, This array will be dynamic since as games are started and finished, entries in the array are created and destroyed. The index into this array is referred to as the game index.

```
game_run_t {
int: game type index
int: number of player slots
*int: array of player codes for registered players
chr: play/wait flag (0 if waiting for players, 1 if playing)
int: process or thread ID
*void: pointer to options struct for this game
*int: array of player codes for reservations
int: number of open player slots
int: file descriptor for communication
chr: computer players (2^num)
}
```

There will also be a large array of *user_t* structures, representing connected users. As soon as a user connects, an entry is created and the file descriptor filled in. When the user completes the login process, the user code and name are filled in. When the user launches, or joins a game, the game index is filled in.

```
user_t {
int: user code (unique user id number)
str: user name
int: file descriptor for communication
int: game index
}
```

An array of *reservation_t* structures holds all of the reservations requested. These are created when a game is launched with reservation requests. They may be altered once the game has been launched. They are deleted when a user accepts a reservation or declines it.

```
reservation_t {
int: game index
int: user code
}
```

The server options are stored in a *Options* structure. This holds many run-time configurable options. *Note:* Not all of these options are implemented at this current time.

```
Options {
str: Name of configuration file specified in --file
chr: remove_users
int: User inactivity time
chr: clear_stats
int: stat_clr_time
int: TCP/IP port to use for communications
str: Directory in which game description files are found
str: tmp_dir
str: The base configuration directory
str: The server admin's name
str: The server admin's email address
int: Whether to perform hostname lookups for log files
}
```

A chat room is implemented internally in a *RoomStruct*. These are stored in a run-time allocated array as needed. Hopefully this will allow on-the-fly room creation in the future.

```
RoomStruct {
str: Short room name
str: Long room description
int: Number of players in room
int: Maximum number of players allowed in this room
int: Number of active tables in room
int: Maximum number of active tables in room
int: The game type this room hosts
time_t: A timestamp when the player list last changed
time_t: A timestamp when the table list last changed
*int: An array of player indices (players in room)
*int: An array of table indices
*ChatItemStruct: The tail of a linked list of chat for this room
}
```

Chat messages are implemented as a set of linked lists, one per chat room. The chat room itself points to the tail of the linked list so that chats may easily be tacked onto the end. Each player has a pointer to their head of the chain, which is the next message they expect to receive. A *ChatItem* is stored as follows:

```
ChatItem {
int: Number of players who have not read this chat
str: The name of the sender of this chat
str: The message itself
*ChatItem: The next message in the linked list
}
```

The message of the day is stored internally in a *MOTDInfo* structure. It is read at system initialization and will not change dynamically (at least yet).

```
MOTDInfo {
str: Filename where MOTD is found
chr: Whether to utilize the MOTD (bool)
ulong: Time the server started up (used to calc uptime)
int: Number of lines in MOTD file
*str: An array of MOTD text lines
str: The server's hostname
str: The server's system name (eg: Linux)
str: The server's CPU identifier - this is not strictly accurate
and depends on who compiled the kernel
str: The port number the server is using
}
```

Logfile options are stored separately from the main server options in a *LogInfo* structure:

```
LogInfo {
int: Have log files been initialized?  If no, we emit to stdout/err
int: Which syslog facility to use
uint: A bitmap of options (see err_func.h)
str: Filename for logfile (if not syslog)
*FILE: Stream for logfile
uint: Log types to include in logs (see err_func.h)
** The following are only included if debug is on **
chr: A flag to note that debug level was set on command line
str: Filename for debug file (if not syslog)
*FILE: Stream for debug file
uint: Debug types to include in logs (see err_func.h)
}
```