The GGZ Game Development Guide

Development of games for the GGZ Gaming Zone 0.99.4+

GGZ Development Team

Table of Contents

1 Foreword

The central element of the GGZ Gaming Zone is, obviously, everything related to games, especially the games themselves. From the very beginning, games of any kind have been a part of GGZ. After a long time of premature development, a lot of experience is present to be shared with the humble reader.

As games are very different from applications, it is not easy to recommend for this or against that. Instead, a general collection of hints is presented, divided into the major differences on the client side (frontends) and the server side (programming languages). This guide covers the following platforms for GGZ game development: KDE/Qt, GNOME/Gtk+, Python (with SDL/Pygame), pure SDL in C, Java and pure C/C++ environments without any graphical toolkits.

The more games provide GGZ support, the more fun it becomes. After reading this guide, all questions about how to provide GGZ support from a game developer's perspective should be answered.

2 The World of GGZ Games

2.1 Games already shipped with GGZ

With more than a dozen games available, which can be played using nearly two dozen game clients, GGZ offers a solid bunch of different games by itself. They're all developed based on a client/server model. So how does that work?

The player, using a GGZ core client to chat with others in a room, launches a game which he wants to play with some others. Technically, a table is created to hold that game, and the player is put onto that table, occupying one seat. The other available seats can be left empty, filled up with bots or used by other players. The quantity of each type is determined by the game server, for example, some games might not support bots (AI players) at all, or it might support an unlimited number of players.

GGZ games are available for basically all standard development platforms. This makes it convenient to take a look at one of the existing games in order to find out programming details. In particular, the Tic-Tac-Toe game is simple enough that it has been awarded the title of being the example game for all platforms.

2.2 Externally developed games which use GGZ

GGZ offers wonderful features to consider for integration into any game. With little work, game developers can foster player communities, integrate flexible protocols and promote their game to a wider audience. The chapters on client and server development will discuss the available options in detail.

External game work no different to internal ones. However, a few assumptions can be made, for example, most external game have been present before they got a GGZ mode, and are thus more complex in design. In the future, some GGZ game servers and clients will most likely also provide a standalone mode, just like some clients already do for local games, namely Muehle, KTicTacTux, Geekgame, Escape/SDL, Xadrez Chines, GGZBoard and Koenig.

2.3 Embedded games

Some games already come with a connection dialog, running in fullscreen and displaying chat messages of other players. In these cases it might be advisable to "embed" the ggzcore library into the game client, letting it perform all the work which traditionally would be done by a GGZ core client. Right now, only one external game runs in embedded mode, namely Widelands. In order to do so, ggzcore must be initialized with the OPT_EMBEDDED flag switched on. The ggzcore API documentation provides more facts and details about the embedded mode.

3 Diving into GGZ

3.1 Preparing a developer setup

Prior to looking into adding GGZ support to a game, it might be useful to understand the GGZ build process. Most likely the game will have a different one but sometimes the situation arises in which one wants to debug GGZ itself. If you're sure you do not ever want this, feel free to skip this part of the guide.

A lot of external software is used in GGZ which should first be installed, along with the developer tools. The `ggzprep` utility, as found in `playground/maintenance`, can do this automatically for some popular Linux distributions. A full GGZ dependency tree weighs in more than 150 MB of binary packages, so the download might take some time.

Building GGZ from source is quite easy but depending on the package a number of tools might be needed. Most parts (C/C++) are autotools-based, meaning they use automake, autoconf and libtool to produce makefiles. The Python package uses autoconf only and the Java package uses the ant system. GGZ Community comes with its own hand-rolled system. The KDE package has switched to CMake recently. Aligning with the build system of the respective platform has proven to be the best and easiest for developers of the platform in question, but of course where they all come together (in GGZ) some care is needed.

Thankfully, the `ggzbuild` tool has been invented. It can be found in `playground/maintenance` as well, together with a ready-to-go configuration file named `ggzbuildrc.sample`. Copy this file to `~/.ggzbuildrc` and modify as needed (changing the installation directory is recommended). Afterwards, calling `ggzbuild` either in GGZ's top-level trunk/branch directory or in each individual module directory will do the right thing. The first way of invocation is recommended for the initial installation as it will always get the dependencies and order of compilation right.

3.2 Distribution

While the following chapters deal with programming the game clients and servers and explain the necessary configuration files which make them known to GGZ, a few words should be said in general about distributing games which provide GGZ support.

First of all, all GGZ packages come with a file README.GGZ, which can be placed alongside of README of the game. It will explain to the user how to run the game in GGZ. Simply refer to it from README or any other documentation and be done with it.

Second, in order to enable GGZ support, the best way is to enable it conditionally. Thus, the game will compile and run even without GGZ, but if GGZ is available it will offer more features. To detect GGZ libraries, the file ggz.m4 can be added to acinclude.m4 in case of autoconf-based C/C++ games. This file should be updated from time to time to reflect changes in GGZ. For CMake-based games, there's GGZ.cmake to detect a base GGZ setup and then there's FindGGZDMod.cmake and FindGGZCore.cmake for the respective base libraries.

All of these files can be found at `http://dev.ggzgamingzone.org/external.php`, a web page which also contains general information about external games and development thereof.

4 Game Clients

4.1 Introduction

Game clients in a client/server relation should be regarded as the visualization of the game itself. This is not the traditional view where the game client was the game itself, but in times of connected entertainment not every player can be trusted, so it is a general agreement that the server is the trusted element and the clients are just there to convert the game data to a more human-friendly format. However, this shouldn't stop anybody of developing great game clients. Furthermore, there are some GGZ games which run without a server, or with a minimal server which leaves all checks to the game clients.

From GGZ version 0.0.6 on, game clients request a separate communication channel from the main client, which is connected to the game server directly. This leads to a slightly more difficult setup, but allows for faster connections between the game server and the game clients. The next section has been added to document this.

4.2 The client base library: libggzmod

This is a short description of ggzmod. Please see the API documentation for details.

The ggzmod library implements the GGZ client-client protocol specification, that is, the communication between a game client and a GGZ core client. The specification can be found at http://www.ggzgamingzone.org/docs/design/clientspec/. The ggzmod library is available for game development in C, and comparable counterparts for C++ and Python. Its homepage is http://dev.ggzgamingzone.org/libraries/ggzmod/.

Both on the server and on the client the concept of GGZ libraries is to adapt to the target platform as much as possible. Therefore, the plain C library can always be used, but C++ developers in KDE will prefer the Qt-only reimplementation called kggzmod, and Python developers making use of the Pygame toolkit will certainly look into the Pygame integration layer that ggz-python's pyggzmod implementation provides. If you do not like any ggzmod implementation because none of them fits your programming style, feel free to suggest another integration layer or reimplementation on our mailing list.

Now that the introduction to ggzmod has been made, let us look into how the library actually works!

A launched game client gets a control channel on its way, which it connects to a callback to receive control messages from the GGZ core client which launched the game. Then it requests a direct game connection, which results in the creation of a game channel, which gets connected to a callback too. This game channel is then used to communicate with the game server. Therefore, game clients usually run with two active connections all the time, a local one and a remote one.

The receipt of the channel is one message from ggzmod which can be delivered via a callback, other such messages include seat changes, table chat or detailed player information. Please refer to the client-client specification for details.

Note that using the game channel is entirely up to the game. GGZ provides some support for common protocols, however nobody is forced to use these. The choice of protocols will be covered in a later chapter.

For three programming languages, the basic usage of ggzmod is now shown in the next subsections. This should cover the majority of game client development scenarios.

4.2.1 Example in C, using glib

The native C interface to ggzmod might be used in plain C, but this will require some mainloop interaction with select() calls. More often, a client application toolkit will be used which provides this facility. A basic ggzmod usage in the C programming language using the `glib` library, e.g. for Gtk+ games, may thus look like this:

```
/* The ggzmod object */
static GGZMod *ggzmod;

/* The game server sent some data */
static void handle_game(gpointer data, gint source, GdkInputCondition cond)
{
        int opcode;

        /* Blocking read, for the sake of simplicity */
        int ret = ggz_read_fd(ggzmod_get_fd(ggzmod), &opcode);
}

/* Whenever the core client sends data from ggzd, feed it into ggzmod */
static void handle_ggz(gpointer data, gint source, GdkInputCondition cond)
{
        ggzmod_dispatch(ggzmod);
}

/* Most important callback: Connection to game server was made! */
static void handle_ggzmod_server(GGZMod * ggzmod, GGZModEvent e, void *data)
{
        int fd = *(int*)data;
        ggzmod_set_state(ggzmod, GGZMOD_STATE_PLAYING);
        gdk_input_add(fd, GDK_INPUT_READ, handle_game, NULL);
}

static int initggz()
{
        ggzmod = ggzmod_new(GGZMOD_GAME);
        ggzmod_set_handler(ggzmod, GGZMOD_EVENT_SERVER, &handle_ggzmod_server);
        /* Other callbacks are available, see ggzmod API documentation */

        ggzmod_connect(ggzmod);
        gdk_input_add(ggzmod_get_fd(ggzmod), GDK_INPUT_READ, handle_ggz, NULL);
}

static void deinitggz()
{
```

```
        ggzmod_disconnect(ggzmod)
        ggzmod_free(ggzmod);
}

void main()
{
        initggz();
        /* game runs here */
        deinitggz();
}
```

Note again that the example above is for a Gtk+ game client written in C. Things will differ for other toolkits like Qt or Pygame, or other programming languages like C++ or Python. Also, return values should be checked and such. The API documentation is the authorative source of information on that matter.

4.2.2 Example in C, using GGZPassive

GGZPassive is a minimal main loop implementation which handles both quantized and non-quantized game protocols. It doesn't support any advanced ggzmod features and is therefore only intended to be used by non-interactive game clients. Right now, GGZPassive is used by Grubby, specifically by the games Guru-TTT (Tic-Tac-Toe) and Guru-Chess (Chess).

```
#include <ggzpassive.h>

void handle_game()
{
        int opcode;
        ggz_dio_get_int(ggz_dio, &opcode);
        if(opcode == MSG_GAME_OVER) ggzpassive_end();
}

int initggz()
{
        /* whenever network data arrives, notify us... */
        ggzpassive_sethandler(handle_game);
        /* ...but only when full packets have been read */
        ggzpassive_enabledio();
        /* now connect to GGZ and enter main loop */
        ggzpassive_start();
}
```

4.2.3 Example in Python

The Python wrapper for ggzmod is more high-level than the underlying C library. For example, messages from the GGZ control channel are processed automatically. The game client doesn't have to dispatch those and can concentrate on the actual game messages.

Hence, a simple setup might look like the following:

```
import ggzmod
```

```
import socket

# Callback for initial setup of channel to game server
def handle_server():
        ggzmod.setState(ggzmod.STATE_PLAYING)
        sock = socket.fromfd(fd, socket.AF_INET, socket.SOCK_STREAM)

def main(with_networking):
        # Conditionally turn on networking
        if with_networking:
                # Notify us when game server sends something
                ggzmod.setHandler(ggzmod.EVENT_SERVER, handle_server)
                ret = ggzmod.connect()
                if not ret:
                        # Disable networking in case of failure
                        with_networking = False
        # ...
        # Here's the mainloop which either polls the GGZ control channel (bad)
        # or waits for events on it (good), depending on the toolkit
        if network_active:
                if ggzmod.autonetwork():
                        # do something with socket here
                        pass
```

Several GGZ games have some sort of Net class which encapsulates the socket handling. In a future version of GGZ such a class might be part of the default installation of pyggzmod. Then, three different ways of using the library are offered: pyggzmod pure as shown above, pyggzmod + pyggznet abstraction and pyggzmod + pygame integration layer.

4.2.4 Example in C++ for Qt-based games

For Qt and KDE games, the kggzmod library fits perfectly into the native programming style for these toolkits. The kggzmod library is a complete reimplementation of ggzmod, not a wrapper for it. It is highly object-oriented, providing a central module class and representations for players, their statistics, requests to GGZ and events from GGZ.

In addition to kggzmod, the kggzgames library provides more comfort for game developers. It contains a ready-to-go dialog for player management and other dialogs. There is also the kggznet library which contains network classes to handle raw and quantized binary protocols seamlessly. The networking classes are called KGGZPacket and KGGZRaw and will be presented later on in the part about networking.

Starting from KDE 4.0, the libraries kggzmod, kggzgames and kggznet are part of libkdegames. As such, they will be available everywhere for KDE game development for the GGZ Gaming Zone.

The following few lines already suffice to get GGZ support going with pure kggzmod:

```
// Let those be global variables here, although they better be class members
KGGZMod::Module *mod;
KGGZRaw *m_net = 0;
```

```
// Setup
void Proto::init()
{
        mod = new KGGZMod::Module("examplegame");
        connect(mod, SIGNAL(signalNetwork(int)), SLOT(slotNetwork(int)));
        connect(mod, SIGNAL(signalError()), SLOT(slotError()));
}

// Callback for messages from the game server
void Proto::slotNetwork(int fd)
{
        // set up network stream if not done yet
        if(!m_net)
        {
                m_net = new KGGZRaw();
                m_net->setNetwork(fd);
                connect(m_net, SIGNAL(signalError()), SLOT(slotError()));
        }

        // read message from game server
        int message;
        *m_net >> message;

        // send back an answer
        if(message = 12)
        {
                int answer = 42;
                *net << answer;
        }
}

// Callback for errors
void Proto::slotError()
{
        delete m_net;
        mod->disconnect();
        // report error and do something
}
```

In the example above, the KGGZMod::Module object will automatically connect to the GGZ core client (or emit an error signal). All the details will be taken care of. For those game developers who need some extra functionality, the kggzmod-internal events can be observed, similar to what plain ggzmod provides. All events piggyback some data, and by casting them to specific event objects, this data can be retrieved in a type-safe and convenient way. Similarly, specific requests can be sent by the game client, e.g. to put an AI player into an open seat.

...

```
        // show interest in kggzmod events
        connect(mod, SIGNAL(signalEvent(KGGZMod::Event)),
                SLOT(slotEvent(KGGZMod::Event)));
...
void Proto::slotEvent(KGGZMod::Event event)
{
        if(event.type() == KGGZMod::Event::seat)
        {
                KGGZMod::SeatEvent se = (KGGZMod::SeatEvent)event;
                // now event.player() can be accessed
        }

        // Just for fun, try to put AI player into seat 5
        KGGZMod::BotRequest br(5);
        mod->sendRequest(br);
}
```

4.3 Game Registry

Every game client must be registered in a central configuration file in order to be found by the GGZ core clients like KGGZ or GGZ-Gtk. This is done via the ggz-config utility during the installation procedure. Basically, every game client has a file called module.dsc, which is used for this purpose: `ggz-config --install --force --modfile=module.dsc`

It is recommended to let the build system create module.dsc based on a template like module.dsc.in, so that system specific values, e.g. those including path information, can be customized.

The central file, in most cases available under /etc/ggz.modules, is appended with the game configuration data, grouping the data by game type, so that a player can select one out of various clients per game, if possible.

The format of the game configuration file module.dsc is defined as follows:

```
[ModuleInfo]
Author = <author>
CommandLine = <commandline>
Frontend = <frontend>
Environment = <environment>
Homepage = <url>
Name = <name of the game>
ProtocolEngine = <engine name>
ProtocolVersion = <protoversion>
Version = <version>
IconPath = <iconpath>
HelpPath = <helppath>
```

The entries don't have to have any special values, except that the combination of protocol engine and protocol version is used to ensure the compatibility with any game server of this name.

The icon path is rewritten if present during installation and referring to a local file, making it possible to install a game icon in one go. Otherwise, the icon and help paths should be absolue file names. Likewise, the command line should point to a game binary, and in case no path information is available, the game client is assumed to exist in the common GGZ game client directory, which normally is under /usr/lib/ggz.

The frontend should be a common abbreviation like kde, gtk, qt or sdl, but this is merely a convention, and new toolkits might be added over time. The environment can be xfullscreen, xwindow, framebuffer, console or passive. The default (if omitted) is xwindow.

The following table describes the toolkit choices.

- sdl: Simple Direct Media Layer, http://www.libsdl.org/
- kde: K Desktop Environment, http://www.kde.org/
- qt: Qt Toolkit, http://www.trolltech.com/
- gnome: GNOME Desktop, http://www.gnome.org/
- gtk: Gtk+ (the GIMP Toolkit), http://www.gtk.org/
- java: Java Swing or AWT game, http://dev.java.net/
- x11: bare X Window System, http://www.x.org/
- console: text console, Linux/BSD terminal or similar
- guru: special value; denotes a chat bot game plugin

The game environment setting can be used to exclude games from systems where they wouldn't run, or poorly so. Core clients decide by themselves which game environments they support.

- xwindow: game runs as a window in the X Window System
- xfullscreen: game runs in fullscreen mode in the X Window System
- framebuffer: game uses framebuffer graphics display, e.g. on small devices
- console: game runs in the text console
- passive: special code to denote that this game client shouldn't be offered to users; this is used for chat bot game plugins, for example

The file readme.ggzconfig, which is part of the ggz-client-libs package, contains more information about ggz-config invocation and the handling of game client configuration files.

4.4 Client Toolkits

This section highlights existing experience with development toolkits for game clients. It is neither complete nor very exhaustive - ask on the ggz-dev list if a particular toolkit is of interest for you!

4.4.1 KDE or Qt (C++, Python)

See http://games.kde.org for information about KDE game development. There is also a mailing list for this topic with information available at http://mail.kde.org/mailman/listinfo/kde-games-devel.

The language bindings for KDE are also maturing, so there shouldn't be major obstacles coding a KDE game client in Ruby, Python, Perl or Java, as long as a ggzmod wrapper library exists for this language.

You will most likely be interested in using Qt classes for OpenGL or sprite animations, so get yourself a copy of the Qt documentation.

KDE/Qt games can use the native classes to monitor the control connection and the game channel, e.g. using a QSocketNotifier. Native networking classes are also part of the toolkits. These will be described in a later chapter.

The GGZ games KReversi, KDots, KTicTacTux, Krosswater, Muehle, Keepalive, Koenig, Kamikaze, Copenhagen and Fyrdman have been written using KDE/Qt as development framework, in the C++ programming language. The GGZBoard variant Kuobe, which currently lives in GGZ Playground, is a KDE game written in Python.

As of version 0.0.14, the GGZ KDE Games library (libkggzgames) is used in GGZ. It contains a dialog for player management, system integration code and at its core the kggzmod library which was already introduced earlier.

After GGZ version 0.0.14, the GGZ KDE clients and games packages underwent porting to KDE 4 and were merged into the new ggz-kde-center package.

4.4.2 Gtk+, GNOME (C)

There are several useful game libraries for Gtk+, including GtkGLArea. SVG support is usually pretty good with Gtk+, too.

Gtk+ games use the underlying Gdk library to monitor the control connection and the game channel. A choice for networking would be libggz, as described in a later chapter.

Examples of Gtk+ GGZ games are Tic-Tac-Toe, NetSpades, Connect the Dots, Hastings1066, Reversi, Combat, Chinese Checkers, Chess and GGZCards. T.E.G. is an example for a GNOME game which provides GGZ support.

Starting from version 2.18, the GNOME-Games package itself supports GGZ in several games, including the use of embedded core clients. Therefore, GGZ game development for GNOME is possible whereever GNOME runs!

4.4.3 SDL (C, C++, Python)

SDL is a proven toolkit for game development, with tons of games already available. See http://www.libsdl.org for a list of them.

Using SDL brings in several additional libraries for sound, networking and graphics formats. Integrating ggzmod in C/C++ games using SDL usually requires the use of a select() loop.

The GGZ games Geekgame and TicTacToe 3D have been written using SDL.

The games Escape/SDL, PyKamikaze, Xadrez Chines and all GGZBoard games are using the Python wrapper for SDL called Pygame, available at http://www.pygame.org. Python games also need to use a wrapper for the ggzmod and ggzdmod libraries, which exist as pyggzmod and pyggzdmod in the ggz-python package. For Pygame developers, a special pygame integration layer has been developed, which makes event processing of both user input and GGZ input very easy.

4.4.4 Swing and AWT (Java)

Java games for GGZ are a very new but popular addition to the project. Right now, they are tightly integrated with the GGZ-Java core client, and not yet included into the GGZ

system integration mechanism. However, GNU Classpath http://www.classpath.org/ provides a free Java API implementation which make games using AWT and Swing GUIs a perfect choice.

Networking classes are part of every Java installation. The port of ggzmod to Java is quite special since the game clients run in the core client's process, not separately, so only one connection (the game channel) needs to be handled by the game clients. This might change or become an option once GGZ-Java is modularized.

All GGZCards games have been implemented in Java, to provide a reference.

4.4.5 Others

It does really not matter which toolkit one chooses. Therefore, Mesa3D (an OpenGL implementation), PLib, GLUT, ClanLib and others are worth to have a look at. Actually TicTacToe 3D uses SDL in combination with OpenGL.

What is important is that the use of ggzmod is convenient. This depends on the toolkit just as on the programming language. If any combination thereof is missing, don't hesitate to contact the GGZ developers.

5 Game Servers

5.1 Introduction

Game servers are spawned by the GGZ main server (ggzd) and are the central game authority, determining the game state (from pregame to game over), and controlling the players, the bots and all game spectators. The Tic-Tac-Toe game server has been written to demonstrate all GGZ features available to game authors who do programming in C; refer to its well-commented source code to get more information.

Game servers are launched by the main GGZ server and receive events on one dedicated file descriptor. There are different event types:

- STATE: indicates that the state has changed (see below)
- JOIN: a player has joined the game
- LEAVE: player has left the game
- SEAT: seat occupation has changed, this encompasses JOIN and LEAVE
- SPECTATOR_JOIN: a spectator came to watch the game
- SPECTATOR_LEAVE: spectator went away
- SPECTATOR_SEAT: spectator has joined or left, this encompasses SPECTATOR_JOIN and SPECTATOR_LEAVE
- PLAYER_DATA: one of the players (clients) sent some data
- SPECTATOR_DATA: data sent from random spectators
- ERROR: an error has occured

Have a look at the API documentation of the ggzdmod library or one of its wrappers, or at the ggz server<->game server protocol documentation to know all the details.

The states in which a game server can be are:

- STATE_CREATED
- STATE_WAITING
- STATE_PLAYING
- STATE_DONE

In order to ease game development, the ggzdmod library has been written, which comes with extensive documentation. See http://www.ggzgamingzone.org/docs/api/ggzdmod/html/ggzdmod_8h.html for the online documentation, which is also available as man page (man 3 ggzdmod.h). Wrappers are available for C++ (ggzdmod++) and Python (pyggzdmod).

5.2 The server base library: libggzdmod

In order to be informed about player joins and leaves, and the status of the game (running, done, ...), the ggzdmod library is essential to use.

This is how a simple C game server embeds the necessary function calls:

```
static void callback_state(GGZdMod *ggz, GGZdModEvent event, void *data)
{
        /* Evaluate state */
```

```
}

int main()
{
...
        ggzdmod_set_handler(ggzdmod, GGZDMOD_EVENT_STATE, callback_state);
...
}
```

5.3 Game statistics: libggzstats

GGZ game servers can use the ggzstats library which offers various statistic types. For example, the ELO ranking and win/loss ratios are supported.

A typical scenario is the following one:

```
GGZStats *stats;

stats = ggzstats_new(ggzmod);
ggzstats_set_game_winner(stats, winner, 1.0);
ggzstats_set_game_winner(stats, (winner + 1) % 2, 0.0);
ggzstats_recalculate_ratings(stats);
ggzstats_free(stats);
```

The statistics are saved into the database (next to the list of registered players), and can be used as data for web-based community sites.

5.4 Configuration Files

Each game server must supply at least one file for each of main server's lists. Currently there is a room list and a game list, so a game can be present in multiple rooms, with either the same or another configuration.

The game file (<name>.dsc) has the following format:

```
[GameInfo]
Author = <author>
Description = <description>
Homepage = <url>
Name = <name of the game>
Version = <version>

[LaunchInfo]
ExecutablePath = <server executable>

[Protocol]
Engine = <protocol name>
Version = <protocol version>

[TableOptions]
AllowLeave = <0/1>
BotsAllowed = <0 [1 2 ...]>
```

```
PlayersAllowed = <1 [2 3 ...]>
KillWhenEmpty = <0/1>
AllowSpectators = <0/1>
AllowRestore = <0/1>
AllowPeers = <0/1>

[Statistics]
Records = <0/1>
Ratings = <0/1>
Highscore = <0/1>
```

The GameInfo and Protocol entries are similar to client-side .dsc files. TableOptions and Statistics, on the other hand, reflect the game server's capabilities. Depending on the scoring model, one of the ranking models can be enabled. The game server will have to report its scores or ratings after the game finishes in this case. The Allow* flags determine if the game can cope with players leaving and rejoining in between, spectators who only watch the game and game restoration based on savegames, respectively. The ggzdmod documentation explains what to do for these cases. KillWhenEmpty is set to 1 by default but can be set to 0 to allow keepalive games even when no players are on the table. Finally, PlayersAllowed and BotsAllowed present a list of valid player and bot configurations. These two entries are lists rather than simple entries. One additional Allow* flag is AllowPeers which if set allows ggzd to reveal the IP addresses of participating players to the game clients of their peers.

The room file (<name>.room) has the following format:

```
[RoomInfo]
Name = <roomname>
Description = <description>
GameType = <name of the game>
MaxPlayers = <maximum of players>
MaxTables = <maximum of tables>
EntryRestriction = none|registered|admin
```

A room gametype of 'none' means that this room is only used for chatting. This is the case with the standard entry room which ships with ggzd. Have a look at the current game server description files if there are open questions.

5.5 Programming Language

5.5.1 C Development

C programmers can use ggzdmod natively.

5.5.2 C++ Development

For C++ developers there exist some choices in wrapping the ggzdmod library. For example, Krosswater uses the Zone library, and Muehle uses the GGZGameServer class, which are both wrappers for ggzdmod. In the latter case, you only need to inherit your main server class from GGZGameServer, and reimplement all of its protected event methods. This class can be found in the ggzdmod++ library. An experimental Qt-based wrapper called kggzdmod also exists now.

5.5.3 Python Development

There are python wrappers for ggzdmod, ggzmod and ggzcore: pyggzdmod, pyggzmod and pyggzcore. They are available as part of the ggz-python package.

5.5.4 Ruby Development

There are two ruby wrappers for GGZ libraries available in the playground: ruggzdmod for game servers and ruggzmod for game clients. In addition, the RubyToe game serves as an example game with client based on the Ruby bindings to Qt4, server and plugged-in AI module.

5.5.5 Java development

Java game server developers can use jggzdmod, which is a complete reimplementation of the ggzdmod library. The jancillary native module is a wrapper around some libc and libggz functions on and makes it easy to integrate Java-based game servers with the system.

6 Programming Details

6.1 Introduction

This chapter covers all the details which might or might not apply to games running on GGZ. It start with GGZ-specific feature implementations and game wrapper tools, and proceeds to general information about networking protocols, configuration files and data updates. In order to not leave the GGZ track, two game development frameworks named GGZCards and GGZBoard are presented, which might be more inviting to new game developers than a completely new game.

6.2 Game features

GGZ provides a lot of features for the games running on it. An overview on them is available in the separate document "GGZ Features for Game Developers". A few of them will be covered here.

6.2.1 Statistics

Depending on how the game presents itself to the GGZ server (see the description of the .dsc files), it is expected to report some statistics after the game has finished. This is done with a call to ggzdmod_report_game() in the ggzdmod library or an equivalent call in the wrapper libraries.

6.2.2 Savegames

In addition to statistics, a savegame can also be reported to the GGZ server. The ggzdmod_report_savegame() call should however not be performed after the game is over. Instead, calling this function as early as possible makes it possible to create a continuous savegame, which can be used for game continuation in case of a game server crash. Automatic savegame reloading support was implemented in GGZ 0.99.3. On-demand loading code is currently under development.

6.2.3 Named bots

If more than one AI player is available, e.g. for different difficulties or to distinguish an internal AI player from an external dedicated engine, the concept of named bots can be used to offer all of them to the user who launches a game. The bots will be registered in ggzd's database on startup. In order to do so, they must be registered with a game server's .dsc file:

```
[NamedBots]
FirstBot = internal_ai
Secondbot = external_ai
```

The ggzdmod_get_bot_class() function is then called on the name of the bot (e.g. FirstBot) and returns the bot class, which the game server uses internally to activate the correct AI accordingly.

6.3 Development and debugging tools

With `ggz-faketable`, a combination of game server and game client can be launched from the command line without the need to have a GGZ server or core client running. The tool will manage initiate its own connections to the game executables and the fake network connection between the game server and the game clients. Debugging becomes easy since errors will be reported centrally by the tool.

Another launch tool is `ggz-wrapper` which acts as a headless GGZ core client and initiates a game on a GGZ server.

6.4 General purpose wrappers

GGZ from version 0.0.6 on has been providing two wrappers which make life easier for game developers. Either one of these wrappers replaces a game of choice, while the game itself is kept to be launched by the wrapper then. This puts the wrapper between the core client and the game client and can be used to debug network connections. The drawback is that no ggzmod event beside GGZMOD_EVENT_SERVER is handled, therefore requesting player information or performing table chat won't work.

6.4.1 ggzwrap

This one should be used in production code, and it's very suitable for the server, because it is small, has no GUI and supports command line options. Its arguments are the program to be executed (–exec), the inbound and outbound file descriptors to be redirected (–fdin, –fdout), and an option whether the received strings should be converted between 'traditional GGZ' easysock order and normal C string order (–convert). So a sample call may look like this: `ExecutablePath=/usr/local/ggz/lib/ggzwrap --exec=/usr/local/ggz/lib/tictactoe --fdin=3 --fdout=3`

The GGZ Muehle game uses this one.

6.4.2 ShadowBridge

The KDE equivalent of ggzwrap can be used to debug network connections visually. It displays incoming and outgoing traffic between the game and the game server, which is (technically) between the game client and ggzcore for most of our games (those without direct connections, anyway).

6.4.3 Go client

The GNU Go integration, which is realized using the CGoBan client, uses a special wrapper (goclient.helper) on the client side, which communicates using FIFOs in both directions: to the GGZ core client, and to the CGoBan client. The wrapper has been written in Python and accomodates those game clients like CGoBan which use FIFOs rather than stdin/stdout to send messages. It should be possible to use it for any similar game without problem.

6.4.4 External AI players

In a broader sense, integration of external AI processes, as in the cases of gnuchess, gnugo and gnushogi, is also of interest in the context of this section. Sometimes, a library to interface with the process is available. This is the case with the GGZ chess server, for example.

6.5 Network connections

The following subsections explain how to add networking capabilities to game clients and servers. They cover the topic on a low level, focusing on nice GGZ integration and ease of use.

Note that GGZComm introduced a XML game protocol format, and the generator ggzcommgen can generate networking source code automatically for several programming languages. Check its page on http://dev.ggzgamingzone.org/protocols/ for more information. Using GGZComm will save a lot of work and lead to less implementation errors.

6.5.1 Easysock (C), now libggz

A very easy and convenious library for those who don't need any fancy features for TCP connections. This library is part of GGZ and therefore always available. It has been merged into libggz, so all functions are prefixed with ggz_ now. See the documentation for libggz for more details. http://dev.ggzgamingzone.org/libraries/libggz/

Usage example of the networking part of libggz:

```
int fd;
char *answer;

fd = ggzdmod_get_seat(mod, 1).fd;
/* or: */
fd = ggz_make_socket(GGZ_SOCK_CLIENT, 9090, "localhost");
...
ggz_write_int(fd, 0);
ggz_write_string(fd, "Hello World.");
ggz_read_string_alloc(fd, &answer);
...
ggz_free(answer);
```

The library supports IPv6, and offers an experimental feature to perform operations such as host name lookups asynchronously. Since easysock itself performs blocking operation, it should not be used in code which requires interactivity (such as GUI code), unless threading is used.

The 'easysock' protocol is also supported by Qt's QDataStream class, and by the KGGZRaw class which is part of the KDE Games library starting from KDE 4.0. It is also supported by the Net class which is part of ggzdmod++ so that C++ game servers can use the same protocol mechanism.

6.5.2 DIO (C), in libggz

As an alternative to the easysock functions in libggz, the Data I/O module (DIO) can be used. Its main difference is that all read and write operations are non-blocking. This is achieved by using a buffer and a two-byte length marker at the beginning of every packet (so-called quantized packets).

```
/* Initialize DIO */
GGZDataIO *dio;
dio = ggz_dio_new(fd);
ggz_dio_set_auto_flush(dio, true);
```

```
ggz_dio_set_read_callback(dio, read_cb, NULL);
...
/* Send a packet */
ggz_dio_packet_start(dio);
ggz_dio_put_int(dio, 0);
ggz_dio_packet_end(dio);
...
/* When data has arrived, start reading */
ggz_dio_read_data();
...
/* This gets called for every packet */
void read_cb(GGZDataIO *dio, void *user)
{
        ggz_dio_get_int(dio, &opcode);
}
```

DIO is not well-documented but not too hard to use. The GGZ game servers Tic-Tac-Toe and GGZCards and all of their respective clients make use of it.

The 'dio' protocol is also implemented by the KGGZPacket class which is part of the KDE Games library starting from KDE 4.0.

6.5.3 Qt (C++)

Those game clients which use the Qt toolkit will most likely use the Qt network code as well. This has mostly to do with QSocket, QSocketNotifier and QDataStream, which are all very easy and intuitive to use.

Documentation: http://doc.trolltech.com/3.3/qdatastream.html

6.5.4 KDE (C++)

KDE does provide very powerful networking mechanisms. There are not only classes like KSocket/KExtSocket, but also the DCOP library, and the KDE games library named libkdegames.

Documentation: http://developer.kde.org/documentation/library/

In libkdegames, the KGGZPacket interface has been added to handle quantized packets ('dio' protocol) for KDE game clients. It integrates easily with kggzmod, but doesn't depend on it in any way. The way the integration works is that instead of letting kggzmod notify the game client for new data from the network, it will notify KGGZPacket about it. KGGZPacket in turn will silently perform its job until all data has been read, and then will notify the game client. This is demonstrated in the following example:

```
// Use KGGZPacket together with kggzmod library
KGGZMod::Module *mod = new KGGZMod::Module("mygame");
KGGZPacket *packet = new KGGZPacket();

// Redirect network notifications from kggzmod into KGGZPacket
connect(mod, SIGNAL(signalNetwork(int)), packet, SLOT(slotNetwork(int)));
// In return, we get notifications from KGGZPacket
connect(packet, SIGNAL(signalPacket()), mygame, SLOT(slotPacket()));
```

```
// Error handling should not be forgotten
connect(mod, SIGNAL(signalError()), mygame, SLOT(slotError()));

// That's it! Game client is up and running and will receive packets
// from the game server
```

Similarly, the KGGZRaw interface can handle raw binary communication, and is also part of libkdegames. KGGZRaw is made available to game servers, but is also used internally in kggzmod. Both classes, KGGZRaw and KGGZPacket, provide full API documentation.

6.5.5 Net/MNet (C++), now ggzdmod++

The game Kamikaze, which provides support for GGZ, introduced an own set of network classes which are very convenient to use. Their development was backed by experiences with the other toolkits. The classes are now available as part of the ggzdmod++ library.

Net objects provide simple buffered input/output with host/network conversion equal to libggz's DIO. However, the packets are not quantized and Net protocols are not compatible to DIO protocols; instead, they're compatible to easysock protocols. MNet objects are inherited and in addition provide multicasting. A sample networking session looks like this:

```
// The network object
MNet n;

// Send unbuffered and buffered message to player 1
n << Net::channel << fd1;
n << "hello world";
n << Net::begin << arg1 << arg2 << Net::end << Net::flush;

// Setup broadcasting
n << MNet::add << fd1;
n << MNet::add << fd2;

// Broadcast message to both players
n << Mnet::multicast;
n << "hello all";

// Reset to normal behaviour
n << MNet::peer;
```

6.5.6 Network (Python)

The Network class as used in PyKamikaze, Escape/SDL, Xadrez Chines and GGZBoard is sitting on top of both the Python Socket module and the ggzmod wrapper. Plans exist to unify the implementations and provide a simple networking layer on top of the socket class, then dubbed Network.

```
# Build network object
net = Network()

# query events
ret = net.autonetwork()
```

```
if ret:
    gameevent()

# send/receive data
opcode = net.getchar()
net.sendbyte(999)
```

6.6 Configuration

Games often want to read or even write configuration data. Each GGZ game server has its own directory where savegames and other data may be stored into. Game clients usually take advantage of configuration classes in their toolkits (e.g. KDE or GNOME), or write a file into the directory ~/.ggz/. Several libraries exist to handle this task.

6.6.1 GGZConfIO (ini-Style), in libggz

Many GGZ configurations are read and written using GGZConfIO. Formerly located in the ggzcore library, it is now part of libggz and consists of many ggz_conf_* functions. The details are again explained in the libggz documentation. http://dev.ggzgamingzone.org/libraries/libggz/

Usage example:

```
int conf;
char *value;

conf = ggz_conf_parse("/etc/ggz/games/mygame.conf", GGZ_CONF_RDWR);
if(conf != -1)
{
        value = ggz_conf_read_string(conf, "SampleSection", "SampleKey", NULL);
        ggz_conf_write_int(conf, "OtherSection", "OtherKey", 99);
        ggz_conf_commit(conf);
        ggz_conf_remove_section(conf, "OtherSection");
        ggz_conf_close(conf);
}
```

6.6.2 Minidom (XML-Style), now in libggzmeta

If you want to read or write XML data, while avoiding the overhead of a complex XML library, the GGZ project offers the minidom library, which you can find in utils/metaserv/libmeta. Currently, both the GGZ metaserver and the Copenhagen game server make use of this library, as well as TelGGZ and other clients who use libggzmeta to communicate with the metaserver.

Usage example:

```
DOM *conf;

conf = minidom_load("/etc/ggz/games/mygame.xml");
if((conf) && (conf->valid) && (conf->processed))
{
        printf("Root tag is named %s.\n", conf->el->name);
```

```
        printf("Attribute 'version' has value %s.\n", MD_att(conf->el, "version"));
}
minidom_free(conf);
```

Minidom is not suitable for real-world complex XML contents, but XML configuration files shipped with games will be read just fine.

6.7 Game development frameworks

6.7.1 Developing card game servers with GGZCards

The GGZ game servers for Hearts, LaPocha, Whist, Euchre, Bridge, Spades, Suaro and Sueca all share one common property: their games are displayed on a GGZCards client, and their messages are transmitted using the same protocol. Other card games can be added quite easily. There's a separate GGZCards documentation available for consultation.

6.7.2 GGZBoard, the board game client container

Consolidating the different interfaces of the already available games of Chess, Go and Reversi, GGZBoard does now already support 10 games, including Hnefatafl, Checkers, Ludo and Shogi. Each game consists of a game module (including client-side AI, optional), a network module (optional) and a server module, if no server exists yet. All the games and the GGZBoard container can exchange messages such as moves, player names, game characteristics or error codes.

The networking protocols are distinct, however some games use the common board games protocol, dubbed Bogaprot.

6.8 Special libraries

6.8.1 GGZChess, an artificial intelligence for chess

Validating board moves, finding opponent moves, and keeping track about the game status is the task of GGZChess, a small library to be used in chess games. Currently, GGZChess is used by the chess game server, the grubby game client, the GGZBoard chess module and a special client for DROPS (L4-based operating system) dubbed Xadrez.

An interface compatible to GGZChess exists for the inclusion of the GNU Chess AI program into chess games.

6.8.2 Data updates with Get Hot New Stuff

Levels, graphics files and other kind of data can be shared using the Get Hot New Stuff framework. In GGZ, the KDE game clients provide support for KNewStuff2 which comes with the KDE libraries, whereas the Python/SDL games can use the SDLNewStuff library which is shipped in the ggz-python package. For SDL games written in C, a special implementation written for Widelands (but not integrated) exists in the playground.

On http://newstuff.kde.org/ the repository for the example game KTicTacTux can be found.