MagiCServer++

Internet Programming Framework for C++

Version 0.1

Developer's Guide

Marko Grönroos (magi@iki.fi)

June 13th 2003

About this document

This document provides information about the MagiCServer++ framework.

Copyright and License

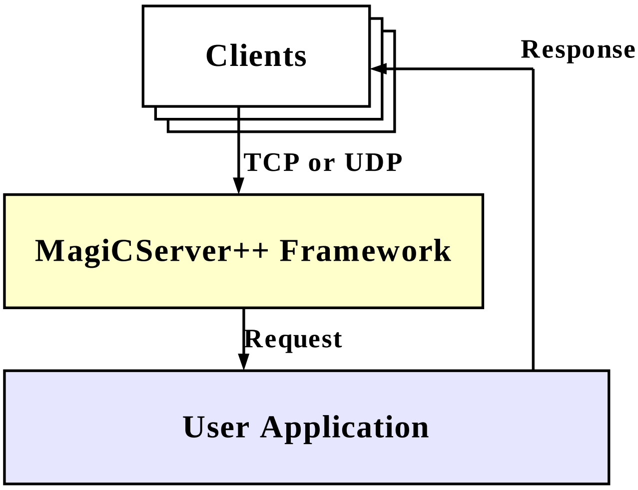
Developer's Guide

MagiCServer++ version 0.1

Table of Contents

Chapter 1 Introduction

MagiCServer++ is a framework for implementing efficient and flexible Internet server applications. It supports both connection-based TCP and connectionless UDP datagram protocols in a transparent fashion.



The main task of the framework is to listen to a server socket and a number of connected TCP client sockets. When a client connects to the server socket, a new connection socket is created and added to the list of established client sockets. When data arrives from a client to a connection socket, it is relayed to the user application as a request. The application can respond, if necessary. The user application is notified also about other important events, such a establishment of a new connection, losing an old one, and initiation of server shutdown.

The framework is implemented as a C++ library, which has been kept as independent from other libraries as possible, to make reuse easier. Error handling is done with error codes; exceptions are not used except for constructors. For data structures, low-level C data structures are used for most tasks. Simple support tools are provided for logging, threading, and queues.

Features

- TCP and UDP
- Threading and thread locking (***Thread*** and ***ThreadLock***)

- Easy-to-use logging facility (***Log***)
- Transparent distribution interfaces

Limitations

- Only one server socket is currently supported in ***ServerListener***.
- No support for multi-process servers
- No C++ wrapper for sockets

For a more complete list of limitations, see the chapter *Known bugs and limitations*.

1.1. System requirements

MagiCServer++ has the following system requirements:

- GNU/Linux operating system
- g++ (GCC) compiler, version 2.96, 3.0, or higher
- pthread library
- GNU Make

Platforms

The following Linux distributions have been tested:

Distribution	***Notes***
Red Hat Linux 9	g++ 3.2
Red Hat Linux 7.3	g++ 2.96 (some pthread functions not enabled by default)
Mandrake 7.0	g++ 2.96
Debian 2.2 + upgrades	g++ 3.3

1.2. Licensing

The library part of MagiCServer++ is licensed under the *GNU Lesser General Public License* (LGPL), also called as *GNU Library General Public License.*

The sample programs are licensed under the *GNU General Public License* (GPL).

The GNU General Public License and GNU Lesser General Public License are given in the source package in files `docs/COPYING` and `docs/COPYING.LIB`.

This documentation is licensed under the *GNU Free Documentation License*, as presented in the end of this document.

Chapter 2 Installing

This chapter describes the installation procedure of MagiCServer++ library, including configuration, compilation, actually installing, and uninstalling.

2.1. Opening the source package

The source code is provided as a `tar` package compressed with `bz2`. You can unpack it with the following shell command:

```
tar jxf magicserver++-0.1beta1.tar.bz2
```

This will unpack the source code in an appropriate subdirectory under the current directory.

2.2. Configuring

To configure the source code for compilation, change to the source directory and run the `configure` script as follows:

```
cd magicserver++-0.1beta1
./configure
```

Optionally, if you wish to later install the package (headers and library) to some other than the default directory, you need to set the installation path with the `--prefix` attribute:

```
./configure --prefix=/usr/local
```

The default path for *root* user is `/usr`, and for other users their home directory.

2.3. Compiling

Include dependencies have to be determined before actual compiling, with the following command:

```
make deps
```

This may produce some errors, which are usually not relevant. Making dependencies is important if you intend to recompile the sources after making changes to them.

The package is compiled with the following simple command:

```
make
```

To build the reference manual (not usually needed), issue command "`make dox`".

2.3.1. Compilation output

The output binaries, documentation, and any intermediate files of the compilation will be written to an output directory tree that is separate from the source tree.

The default output directory is located in:

`/tmp/$USER/build/<architecture>/release`

where `$USER` is the user name and *architecture* is the operating system and processor architecture, for example, `Linux-i686`.

For example, binaries are found under the `bin` subdirectory:

```
cd /tmp/$USER/build/Linux-i686/release/bin
./msrvsample_listener
```

2.4. Installing

After compiling, you can be install the package under the configured installation directory (see above) by issuing the following command in the source directory:

```
make install
```

This will copy the output library binaries and header files to appropriate subdirectories under the installation directory.

Directory	***Description***
`<instdir>/lib`	Libraries
`<instdir>/bin`	Binaries
`<instdir>/include/magicserver`	Headers

Notice that any documentation that comes with the source package is currently not installed anywhere.

After installation, you can clean the output directory tree with "`make clean`" command in the top-level source directory to remove all the temporary files. You do not need to clean the output, but you might want to free the disk space.

2.5. Uninstalling

You can remove the installation by giving the following command in the source directory:

```
make uninstall
```

This removes the installed files and directories only if the installation path has not been changed with `configure` script after installing.

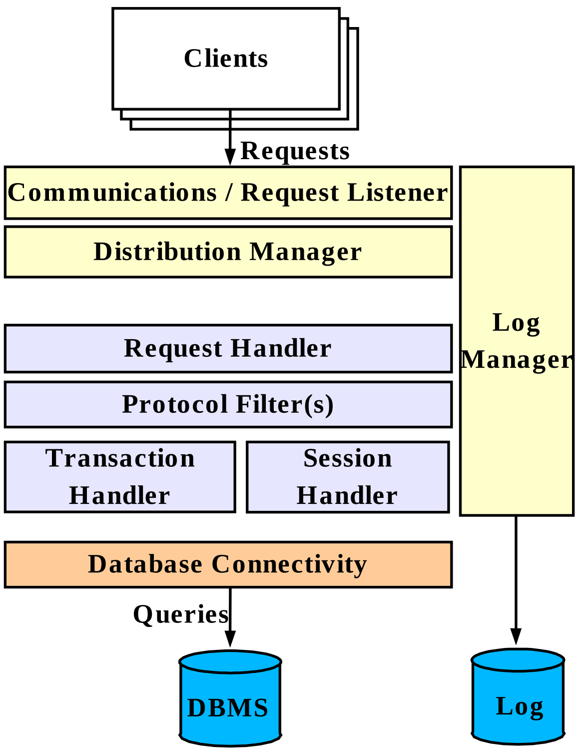
Chapter 3 Server architectures

This chapter describes a general overview to server architectures made possible by MagiCServer++ framework.

Instructions for actually using MagiCServer++ are provided in the subsequent chapters.

3.1. Overview

Figure below presents a generic server architecture, with communications framework tasks shown in yellow, application-specific modules shown in pale blue, database connectivity interface in orange, and databases in blue.



The "**clients"** represent any communicating entities, such as client application or servers in a multi-tier model.

The **communications** or **request listening layer** is the most low-level layer that listens for new connections or data from clients.

Distribution manager is an application-non-specific layer that allocates system resources to request handling by distributing requests to different threads or processes. The threads and processes can execute in a single or multiple CPUs in the same computer, and in some distributed processing architectures processes can exist also in separate computers. Choices of distribution management architectures for different tasks are described in detail in subsequent sections.

The distribution manager communicates requests to the application-specific **request handler**. The request handler is not a functional part as such, but merely an interface through which the requests are communicated.

All communication uses many different **communication protocols** on different levels. Application-level communication protocol is the highest-level protocol, handling application-specific requests. Examples of such protocols are HTTP in a web server, FTP in a file server, and so on.

Transaction handler contains the core application-specific logic for handling requests. Often they too have a hierarchy, typically with a transaction manager that classifies and distributes requests to specific handlers.

The transaction handler often incorporates a **session handler**, which handles tasks and data concerning client sessions. In some applications, this enables dynamic interplay between clients. Many applications do not need to track sessions.

Most servers require access to data storage, either a local file system or a remote database. For databases, **database connectivity** interface layer is required for access to databases. Transparent database connectivity adapter interfaces that can be used to access different types of databases include systems such as ODBC, JDBC, etc.

Logging described in the diagram above is a *vertical log*, that is, all levels of the application write to the same log. In small applications, logs are typically written to a local file system, but in large systems they are often written to a database.

3.2. Distribution architecture

Different server architectures fit in different computing tasks and hardware environments. We can observe at least the following case-specific factors for choosing between distribution architectures:

- Frequency and CPU burden of client requests
- Number and frequency of new client sessions

- Persistence of client sessions
- Response time requirements
- Number of processors
- Back-end database requirements
- Request protocol used: TCP or UDP

All these differences affect the choice of the server architecture for a particular task. Number of processors is certainly an important factor, as taking advantage of multiple CPUs is not possible within a single thread. However, threading is useful also for most single processor situations. For example, if some requests are heavy to process, but short response times would be preferred for light requests. In such cases, it is best to use threading to allow the scheduler to give time slices to light requests.

On the other hand, if client connections are very short -- and especially if requests are done with connectionless UDP datagrams -- and if they are very light and very frequent, it may be best not to use an advanced listening framework at all. In such a case, simple `accept()` and `read()` loops would suffice for a TCP server and `recvfrom()` loop for a UDP server.

MagiCServer++ has currently no support for multiple-process distribution architectures, so they are not considered in this treatment.

3.2.1. Single-thread listener architecture

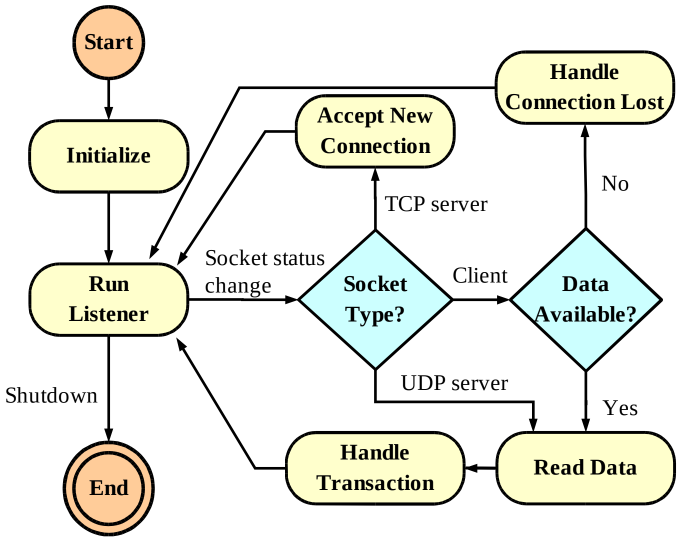
Many servers run in a single thread of execution. This is often a good approach in single-processor machines if client requests are light to process. No time is wasted in distributing requests or context switches. This architecture can be used for both TCP and UDP servers.

UNIX-based systems allow two low-level mechanisms for implementing efficient single-thread servers, one based on the `select()` function and another based on `poll()`. The latter mechanism is not discussed here. The `select()` function allows to wait, that is, to "listen" for status changes in a set of descriptors, for example file or socket descriptors. The exact meaning of status change depends on the socket type.

The listener needs to maintain a list of different types of descriptors (sockets) it is listening. When a status change occurs for a listening TCP server socket, it can only mean that a the listener must use the `accept()` function to establish the connection. If the socket is an UDP server socket, the status change indicates arrival of new data in the socket. If it is a connected client socket, it can also indicate arrival of data, but also breaking of connection.

When request data arrives on a socket, the server can read it, process it, and send a response back to the client. After the socket events are handled, the server returns back to the `select()` loop.

The activities during the execution of a single-thread server are illustrated below:



This architecture type is implemented in MagiCServer++. For more details, please see the following chapters.

3.2.2. Per-client thread architecture

In a "per-client thread architecture", each client connection has its own thread. The task of the main thread is simply to accept new connections, which can be done in a trivial `accept()` loop. When a new connection is accepted, a new thread is created and the main thread can immediately return to the `accept()` loop. The threads, each handling one client, execute in a simple `read()` loop that receives data from the client and send responses. A client thread exits when the connection to the client is terminated, either intentionally with `close()` or because the connection was lost.

This architecture is mostly suitable for cases where client sessions are rather long, as creation of the client threads can take considerably long, even hundreds of milliseconds. Context switches between threads take some overhead, so many threads a lot of time wasted in context switching. Operating systems also have limits on the number of threads, so this architecture can't be used if the maximum number of simultaneous connections can exceed those limits. In addition, threads also take some memory overhead, typically about 16 kilobytes in Linux.

A `read()` loop can pull data in very fast and make quick responses with little latency. The `read()` function has no timeout mechanism, which can lead to problems. Using `select()` for just one socket is one solution.

This architecture is only applicable to TCP servers.

3.2.3. Listener thread architecture

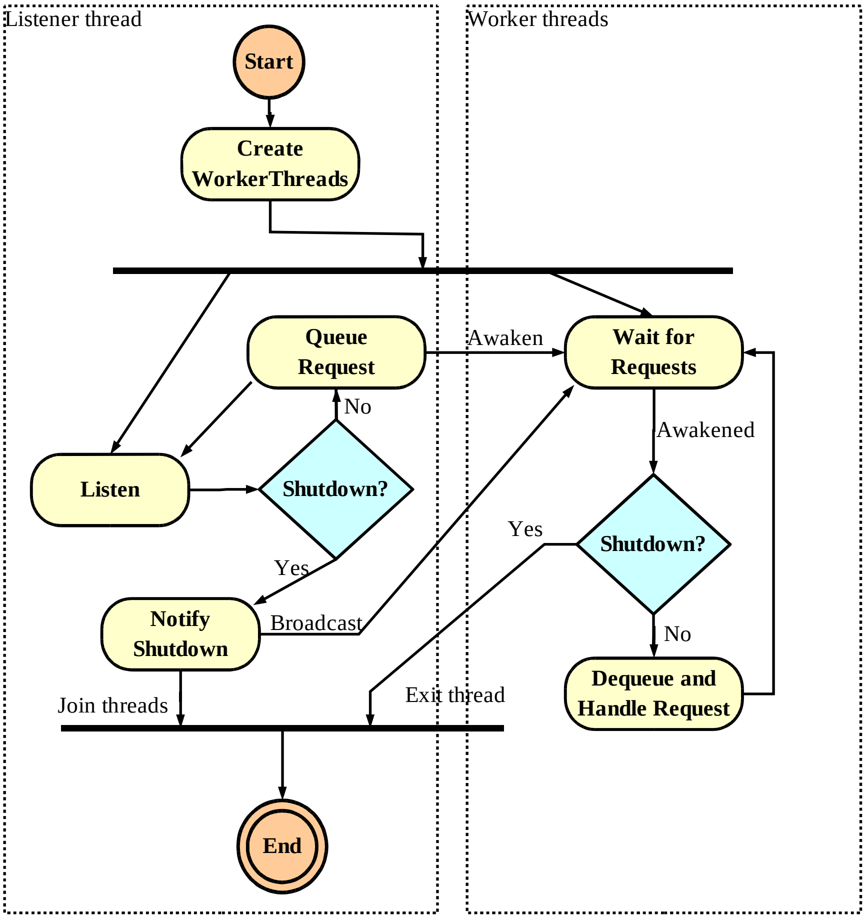
In a listener thread architecture, the main thread listens the server socket for new connections. When one arrives, it accepts the connection and passes the TCP connection socket to a listener in another thread (or a pool of threads). There can be one connection listener for each connection (a trivial case) or a single thread can handle requests from multiple client connections.

This architecture is only applicable to TCP servers.

3.2.4. Worker thread architecture

The worker thread architecture is much like the single-thread listener architecture. The server socket and the client sockets are listened and otherwise managed in the listener thread. The difference is that the requests are passed to other, "worker", threads to process.

The activities of a worker thread server are illustrated in the diagram below:



The requests are queued in the listener thread. The threads normally sleep in wait state, waiting for a signal. When the listener thread inserts a request in the queue, it awakens one of the sleeping threads to process the request. Also, when the main thread has started shutdown, it broadcast a signal to awaken all the threads to exit them.

Worker thread architecture is especially useful when most of the workload is in processing requests. There is some overhead in creating request objects and queuing them for the worker threads to process, so this architecture may not be best if all the requests are very light and very frequent.

While the worker thread architecture excellent for TCP servers, it is the only sensible threaded architecture for UDP servers.

3.2.5. Multi-process architectures

Threads are a relatively new concept in Linux and UNIX server programming. Most existing servers do not use threads for distribution and concurrency, but multiple processes. Using processes instead of threads has both very big advantages and disadvantages. A very important advantage is that each child process runs in its own protected data memory segment so memory corruption in one process does not crash the entire server. Even more advantageously, this model also protects the server effectively from segmentation faults and other crashes.

The most traditional way to implement a multi-process server is an "*ad hoc fork*" architecture where we accept new connections in `accept()` loop in the main process. When a new connection is established, the main process forks (with `fork()`) to create a child process to handle the client session. The child exits after the connection is closed. This model is very heavy, as the creation of a new process with fork can take considerably long time.

A very similar way, "*pre-fork*" architecture, is to first create the server listener socket in the main process and then fork a number of child processes that all share the same server socket. They all enter an `accept()` loop that accepts connections. After accepting a client, the child process handles the entire client session and then returns to the `accept()` loop. This architecture have some obvious variants. Each child process can have a listener loop as we had above in the single-thread model. Each of such child processes can also have an internal distribution mechanism using threads. The main process can accept new connections and process requests just like the children do. This is, however, not recommended because if the main process crashes, the server crashes. It is therefore best that the main process only takes care of watching over the child processes, re-spawning them if they crash, and other administrative tasks.

The disadvantage of process-based distribution is that the processes do not normally use a common data memory segment, which is a necessary requirement for many kinds of servers. This problem is more or less easy to solve by using shared memory or other inter-process communication, but these solutions easily bring back the initial problem -- data corruption.

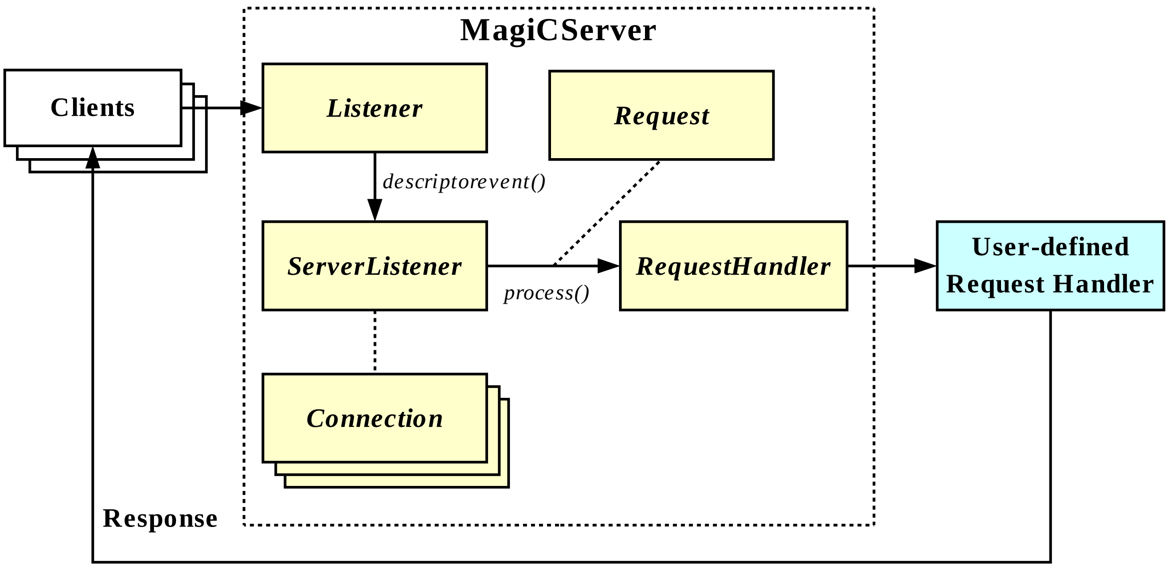
MagiCServer++ does not implement any multi-process architectures, so this topic is not discussed any further. One reason for this omission is that, because the processes have separate data segments, a process-based distribution architecture can not be done as transparently as the thread-based distribution can be done.

Chapter 4 MagiCServer++ architectural overview

This chapter provides an overview of the general architecture of MagiCServer++.

4.1. Introduction

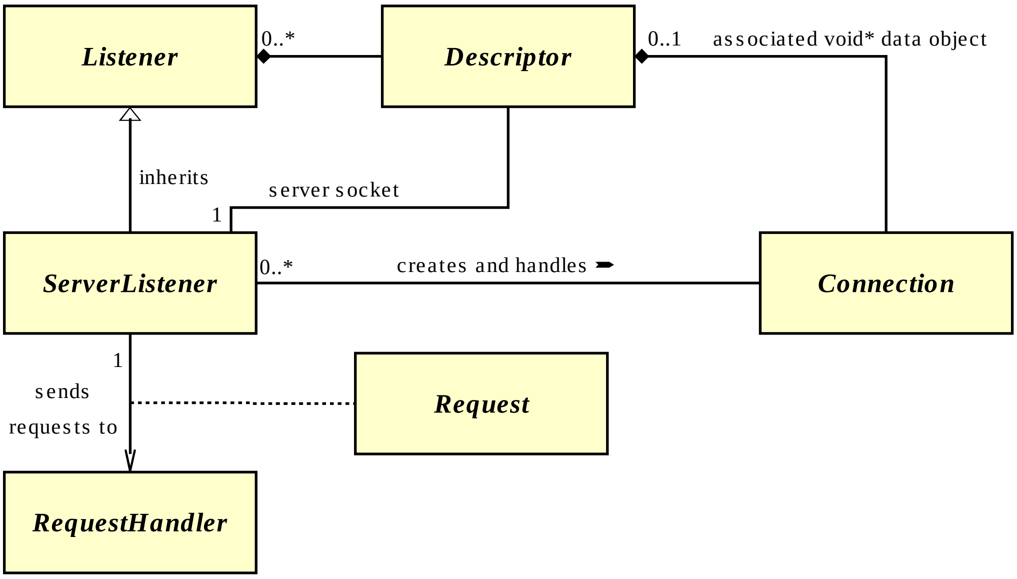
The centerpiece of the framework is the ***Listener*** object, which listens to a number of server or client sockets. ***ServerListener*** is a higher-level class that distinguishes between server and TCP client sockets and manages TCP connections. When a socket event occurs, it calls a user-defined ***RequestHandler*** with the request data provided in a ***Request*** object. Each TCP client connection has an associated ***Connection*** object, which the user application can inherit to store application-specific connection data. This is illustrated in the figure below.



In addition, the framework provides an advanced distribution facility for threaded servers, implemented with ***WorkerPool*** and ***Worker*** classes. This facility has transparent interfaces that enable effortless transition from one server architecture to another.

4.2. Listener framework

The ***Listener*** object manages a set of *descriptors* that can have data associated with them. It listens for descriptor events and, when one occurs, calls `descriptorEvent()`, This method is an *event handler* that needs to be implemented by an inheritor.

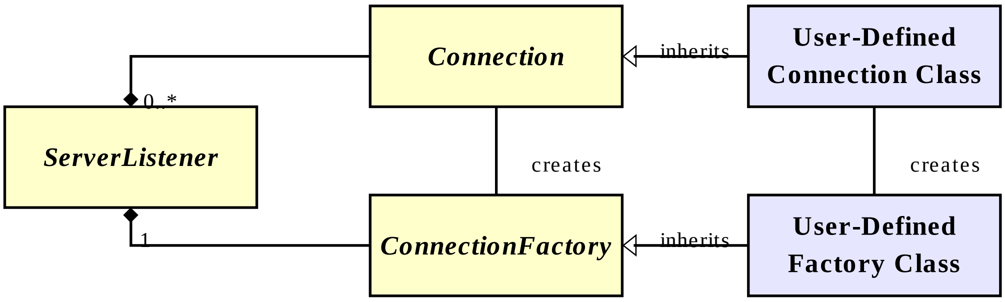


The descriptors handled by the ***Listener*** can actually be any file descriptors such as files, pipes, or sockets. The ***Listener*** class doesn't know anything about sockets or socket types (server or client) or ***Connection*** objects. These are semantics attached to the descriptors and their associated data by the ***ServerListener*** class.

ServerListener inherits ***Listener*** to give the descriptors a specific meaning: they are sockets. One of the sockets is the server socket, which is can be either a TCP or UDP socket. TCP (stream) server socket can accept new client connections and UDP server socket can receive datagrams. A TCP server has also a number of client sockets. The ***Listener*** as just as descriptors, but ***ServerListener*** identifies their associated data object as ***Connection*** objects.

4.3. Connections

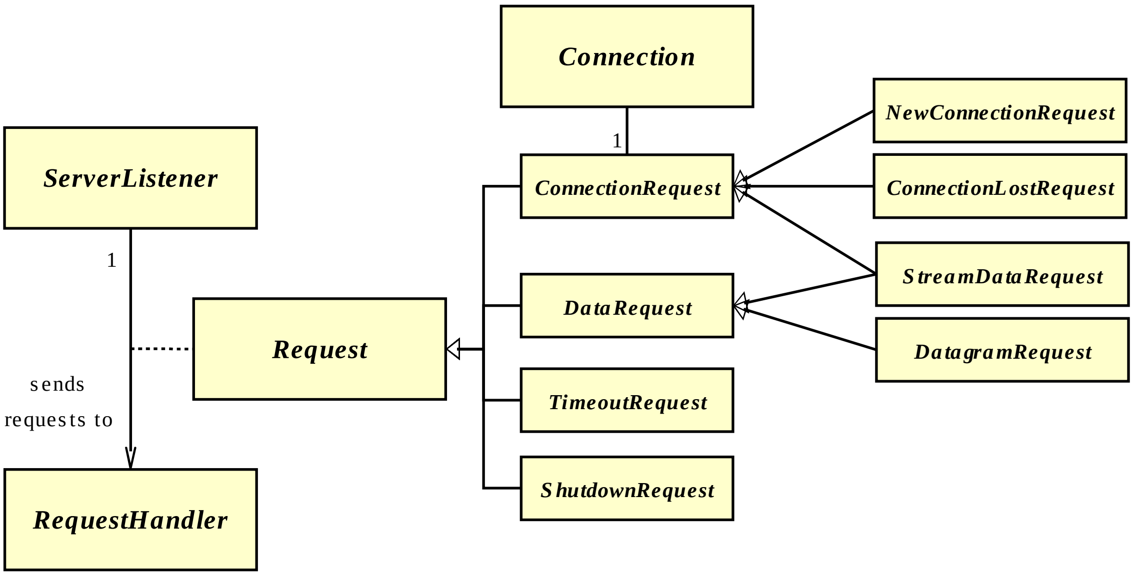
The connection objects are used for accessing connection-related data associated with a socket and session handling. A user application can inherit the ***Connection*** class to store application-specific data to connection objects. In such a case, the user application also has to inherit ***ConnectionFactory***, re-implement the `create()` method, and give a reference to the factory to ***ServerListener*** with `setConnectionFactory()`. The ***ServerListener*** then uses the factory to create user-defined connection objects.



(Note that the diagram above illustrates logically relevant relationships of MagiCServer++ classes; the actual implementation is slightly different.)

4.4. Requests

Requests are generated by the ***ServerListener*** and passed to the ***RequestHandler*** to process. The request related classes have the following class relationships:



The ***ConnectionRequest*** class is associated with an established TCP client connection object, to provide its inheritors access to that object. The abstract ***DataRequest*** class contains a data buffer containing request data read from the socket by ***ServerListener***.

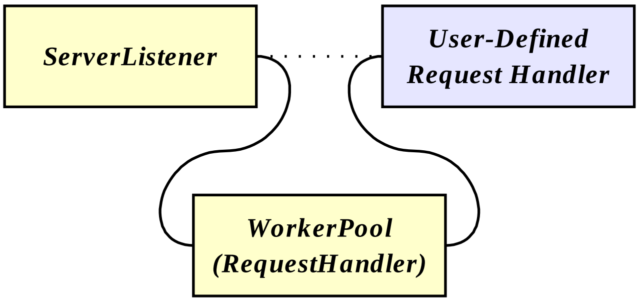
After being processed, the requests are destroyed by the request handler.

The semantics of the various requests are detailed in next chapter.

4.5. Distribution with workers

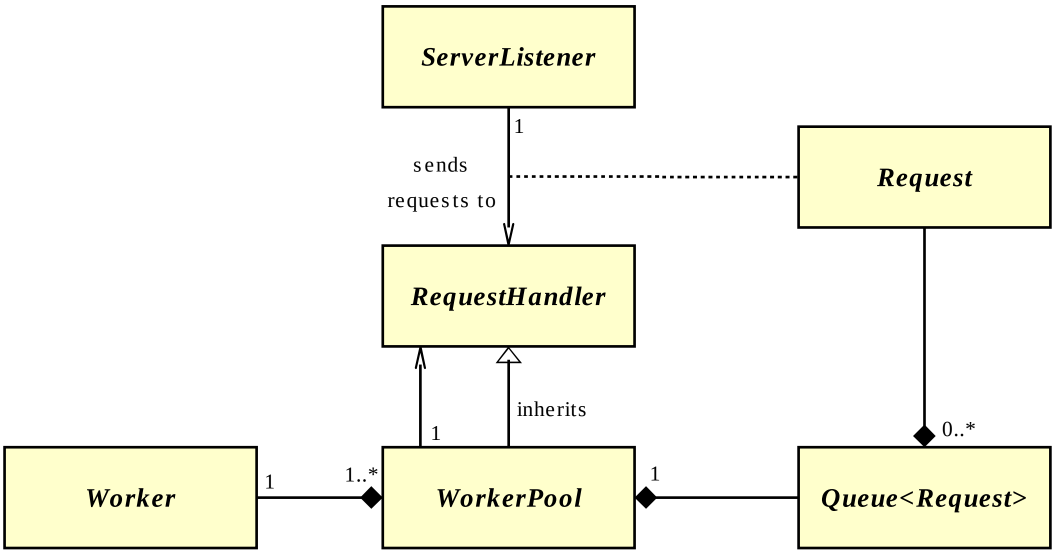
MagiCServer++ provides one distribution mechanism, which is an implementation of the *worker thread architecture* discussed in the previous chapter. For a more general description of the distribution mechanism, please refer to the previous chapter.

Distribution is done transparently using the ***WorkerPool*** class, which inherits and therefore acts like a ***RequestHandler*** that receives requests from ***ServerListener***. This transparent chaining is illustrated in the following schematic diagram:



WorkerPool distributes the requests to workers that process them with a user-defined request handler using exactly the same `process()` interfaces of ***RequestHandler*** class as without the distribution.

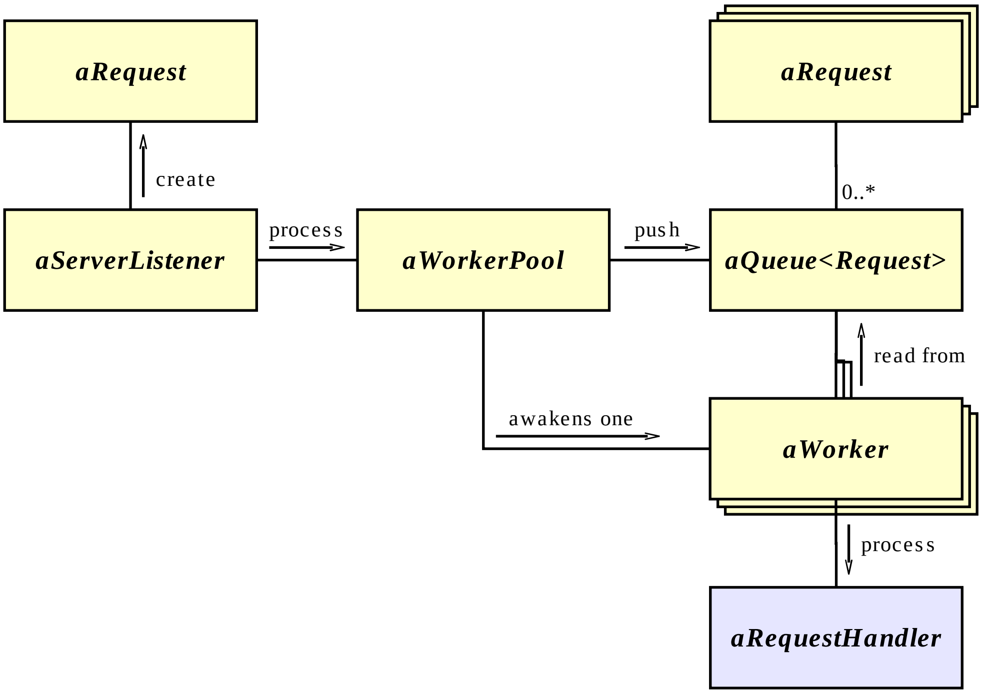
The relevant class relationships are illustrated in the class diagram below:



The ***RequestHandler*** referred by the ***ServerListener*** is a ***WorkerPool***, which in turn refers to a user-defined sub-class of ***RequestHandler***. Each of the ***Worker*** objects have an execution method that executes in its own thread, reading requests from the queue and passing the to the user-defined request handler.

Note that the ownership of the ***Request*** objects changes several times during their lifetime. ***ServerListener*** passes the ownership to the ***WorkerPool***, which passes it to the request queue, which passes it to the Worker, which passes it to the request handler.

The following object collaboration diagram illustrates the handling of requests:



The user of the worker pool distribution mechanism can chain the request handlers as follows:

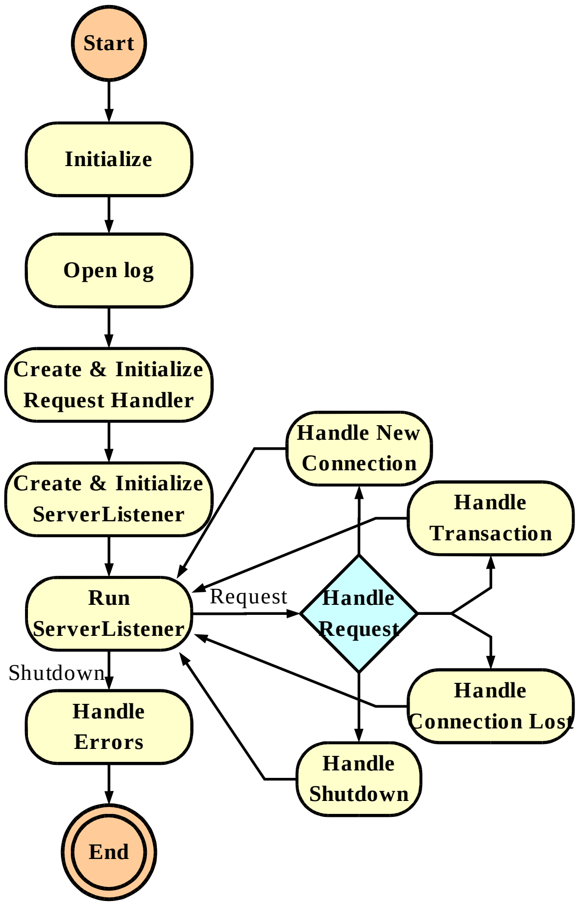
```
/* Create transaction handler. */
MyHandler myHandler;

/* Create a worker thread pool. */
WorkerPool workers (myHandler, log, 10);

/* Create and configure server object. */
ServerListener myServer (workers, &log);
```

Chapter 5 Using the server framework

This chapter gives an introduction to programming server applications with MagiCServer++ framework.

5.1. Overview

The following diagram illustrates the basic activities during the execution of a server with a listener architecture. Distribution is not shown in this diagram.



Initialization includes parsing possible command-line arguments, reading configuration files, setting up the application-specific data structures, and performing other application-specific tasks such as initializing database connections.

Other main tasks of the server execution are described in following sections.

5.2. Writing a request handler

A request handler is a class that inherits the ***RequestHandler*** class and reimplements the generic `process(Request*)` method or any of the more specific `process()` methods. The default implementation of `process(Request*)` is a switchboard that casts the *Request** to the proper subclasses and calls the appropriate handler method.

5.2.1. Request specific handler

A request specific handler allows the default implementation of *process(Request*)* to cast the Request pointer to proper subclass and call the appropriate *process()* method. It also takes care of destructing the request object after processing.

Class	***Description***
NewConnectionRequest	A new TCP client connection has been accepted.
StreamDataRequest	Data has arrived to TCP client socket. The data is automatically read from the socket by ***ServerListener*** and can be retrieved from the request object with *getData()* and *dataLen()* methods.
DatagramRequest	Data has arrived to UDP server socket. The data is automatically read from the socket by ***ServerListener*** and can be retrieved from the request object with *getData()* and *dataLen()* methods.
ConnectionLostRequest	Connection to a TCP client has been lost. The socket is closed. The ***Connection*** object associated with the connection will be destroyed after this request.
ShutdownRequest	Shutdown of the server has been initiated. New connections will not be accepted. Old connections are still open and the request handler can send a shutdown message to them and then close them. The ServerListener exits after this request is processed. If ***WorkerPool*** is used, all ***Worker*** threads are exited before sending this request, which is processed in the main thread.
TimeoutRequest	***Listener*** timeout counter has timed out.

Inheriting RequestHandler

Let us first look at the header for a request handler:

```
#include <MagiCServer++/msrvrequest.h>

using namespace MSrv;

/*****************************************************************
 * Sample request handler
 *****************************************************************/
```

```
class MyHandler : public RequestHandler {
  public:
                   MyHandler   () {/* ... */;}
    virtual        ~MyHandler  () {/* ... */;}

    /* Reimplementations */
    MSrvResult request (NewConnectionRequest* request);
    MSrvResult request (ConnectionLostRequest* request);
    MSrvResult request (StreamDataRequest* request);
    MSrvResult request (DatagramRequest* request);
    MSrvResult request (ShutdownRequest* request);
    MSrvResult request (TimeoutRequest* request);
};
```

Notice that the MSrv name space needs to be used for accessing classes in the MagiCServer++ library. You can do it with the "`using`" directive, as above, or by using the `MSrv::` name space specifier for all MagiCServer++ classes.

The constructor should handle any application-specific initialization and the destructor destruction. Inheritor can naturally have any application-specific member variables.

New Connection Request

The following request handler example resolves the host name of the connected client and sends it a welcome message.

```
MSrvResult MyHandler::process (NewConnectionRequest& rRequest)
{
    /* Resolve host name. */
    struct hostent* host = gethostbyaddr (
              &rRequest.connection().address().sin_addr,
              sizeof (in_addr),
              AF_INET);

    /* Send welcome message to the client. */
    if (host) {
        char msg[MYLEN_OUTPUT_BUFFER_SIZE];
        snprintf (msg, MYLEN_OUTPUT_BUFFER_SIZE,
                  "001 Hello, %s!\n",
                  host->h_name);
        write (rRequest.socket(), msg, strlen (msg));
    }

    return 0;
}
```

Connection Lost Request

The connection lost request occurs when a client connection is unexpectedly broken, usually because the client end closed the socket.

The following request handler example simply logs the event.

```
MSrvResult MyHandler::process (ConnectionLostRequest& rRequest)
{
    rRequest.serverListener().log().message (
                                "SAMPLE", Log::Audit, 0,
                                "Connection lost");
    return 0;
}
```

The ***Connection*** object associated with the connection will be destroyed when the ***Request*** object is destroyed.

Stream Data Request

The stream data request occurs when data is received from an established TCP client connection. The data is read automatically to a buffer stored in the request object.

The following request handler example processes the request and sends a response.

```
MSrvResult MyHandler::process (StreamDataRequest& rRequest)
{
    char*      data   = rRequest.getData ();

    /* Cut request data buffer at newline. */
    for (int i=0; i<rRequest.dataLen(); ++i)
        if (data[i] < '\x20') {
            data[i] = 0x00;
            break;
        }

    /* Shutdown command. */
    if (!strcmp (data, "shutdown"))
        rRequest.serverListener().startShutdown ();
    else {
        /* Format and send a response. */
        char msg[1024];
        snprintf (msg, 1024, "Hi, you wrote: '%s'.\n",
                  data);
        write (rRequest.socket(), msg, strlen (msg));
    }
}
```

Shutdown request

Shutdown requests occur when the ***Listener*** (and ***ServerListener***) is shutting down after receiving `startShutdown()` command. Below is a sample handler:

```
MSrvResult MyHandler::process (ShutdownRequest& rRequest)
{
    rRequest.serverListener().log().message (
        "SAMPLE", Log::Info, 0,
        "Server is shutting down.");

    /* Send shutdown message to all connected clients. */
    const char* msg = "Shutting down immediately! (byebye)\n";
```

```
    /* Iterate through connections in the listener. */
    for (ServerListener::ConnIter serv_i (
                                rRequest.serverListener());
         !serv_i.exhausted ();
         serv_i.next()) {
        /* Write shutdown message to the connection. */
        write (serv_i.get().socket(), msg, strlen (msg));
    }

    return 0;
}
```

The request handler does not have to close the client connections, as ***ServerListener*** does that automatically after this request has been processed.

If the worker threading model is used (with ***WorkerPool***), the final shutdown request will be processed in the main thread after other threads have been shut down.

For more details see the section on Shutting down below.

Timeout request

Listener generates timeout events according to the timeout setting set with `setTimeout()` and calls `timeoutEvent()`, which can be reimplemented by inheritors. The implementation in ***ServerListener*** generates ***TimeoutRequest*** requests, presuming that the timeout events have been enabled in ***ServerListener*** by applying the `Request::NewConnection` mask with the `setRequestMask()` method. For example, if a request handler wants to process also the timeout requests, it should implement the following kind of initialization function:

```
MSrvResult MyHandler::init (ServerListener& rListener)
{
    /* Add the timeout request to request mask. */
    rListener.setRequestMask (
        rListener.requestMask() | Request::Timeout);
}
```

Timeouts are the only request type disabled by default. You might not want to generate timeout requests if the ***Listener*** timeout period is very small, as processing them may take considerable time. This is one reason why the timeout should not be very short.

The time requests can be handled as follows:

```
MSrvResult MyHandler::process (TimeoutRequest& rRequest)
{
    rRequest.serverListener().log().message (
        "SAMPLE", Log::Info, 0,
        "Listener notified about a routine timeout.");

    /* We could do something interesting here if we wanted. */
    return 0;
```

```
}
```

5.2.2. Generic request handler

Writing a generic request handler is an alternative way to handle the requests. A generic request handler processes all types of requests in a single method, `process (Request*)`. Reimplementing it is useful mostly in chained handlers, as is done in ***WorkerPool***. Its speed advantage is not significant (a few nanoseconds perhaps).

Inheriting generic handler of RequestHandler

Let us first look at the header for a request handler:

```
#include <MagiCServer++/msrvrequest.h>

using namespace MSrv;

/*******************************************************
 * Sample request handler
 *******************************************************/
class MyHandler : public RequestHandler {
  public:
                      MyHandler   () {/* ... */;}
    virtual           ~MyHandler  () {/* ... */;}
    virtual MSrvResult request (Request* request);
};
```

Generic request handler method

The `RequestHandler::request()` method handles the actual request. The inheritor of ***RequestHandler*** must reimplement it.

The `Request::gettype()` method returns a numeric type identifier for a request, allowing a simple switch-case construction for handling them. The requests can have the following types:

Type	***Class***
`NewConnection`	***NewConnectionRequest***
`StreamData`	***StreamDataRequest***
`Datagram`	***DatagramRequest***
`ConnectionLost`	***ConnectionLostRequest***
`Shutdown`	***ShutdownRequest***
`Timeout`	***TimeoutRequest***

Below is a very simple skeleton for a request handler:

```
MSrvResult MyHandler::process (Request* pRequest)
{
    switch (request->getType ()) {
```

```
        /* Handle new connection notification. */
        case Request::NewConnection: {
            ...
        } break;

        /* Handle data requests for both TCP and UDP. */
        case Request::StreamData:
        case Request::Datagram: {
            ...
        } break;

        /* Handle connection lost notification. */
        case Request::ConnectionLost: {
            ...
        } break;

        /* Handle server shutdown notification. */
        case Request::Shutdown: {
            ...
        } break;

        /* Unhandled request types. */
        default: {
            ...
        }
    }

    delete request; /* Handler has to destroy it. */
    return 0;
}
```

Responses to client are sent using standard low-level I/O routines for sockets. For a more detailed example, see the *Common Sample Request Library.*

Notice: Request handler has the responsibility to destroy the Request object.

5.3. Main application

The main program of a server application consists of the main tasks illustrated in the diagram above. Let us go through them step-by-step.

5.3.1. Opening log

Server log has to be opened for writing before the ***ServerListener*** can be created (see below), as its creation and initialization may need to write to the log.

```
/* Open log to a file. */
LogFile log ("mylog.txt");

/* Write log entry. */
log.message ("SAMPLE",       /* Module name.           */
             Log::Audit,     /* Error severity.        */
             0,              /* Optional message code. */
             "Log opened.");/* Actual message text.   */
```

This opens a log file and writes an opening entry to it. If the log file already exists, the log is appended to the end. Otherwise the file will be created. The *module name* given to the `message()` method is a free label attached in log entries. The *error severity* can be either `Audit`, `Warning`, or `Critical`. An `Audit` entry means any routine message. `Warning` means an unusual situation, but which is handled cleanly. `Critical` means that a serious error has occurred, which may lead to data corruption or eventual malfunction of the server.

The *optional message code* is a numeric code that is attached to each message. Its purpose is to give a clear identifier for the specific message. Numeric message codes are especially important in internationalized applications where the actual message text can be in a language that is not understood by the developers or support personnel. Numeric codes are also useful for automatically analyzing logs, because the actual message texts are often changed slightly for various reasons (spell checking, etc).

The actual message text is like a format string for `printf()` and you can give it additional arguments. For example:

```
    log.message ("SAMPLE", Log::Critical, 0,
                 "Execution failed with error %d.",
                 -msrvResult);
```

If you wish to open the log to write to standard output for some reason, you can set it in two ways, either:

```
LogFile log (stdout);
```

or:

```
LogFile log ("-");
```

5.3.2. Creating, initializing, and running ServerListener

Creation of the ServerListener is easy, after which it must be initialized. Setting the request handler is mandatory. Setting log is usually desired, though some simple servers do not need logging.

```
/* Create transaction handler. */
MyHandler myHandler;

/* Create and configure server object. */
ServerListener myServer (myHandler, &log);
```

Next step in server initialization is setting up the server socket. This requires a port number and protocol. Choises for the protocol are `ServerListener::TCP` and `ServerListener::UDP`.

```
/* Create a server socket and bind it to an address. */
```

```
msrvResult = myServer.bind (portno,
                            ServerListener::TCP,
                            0);
if (msrvResult < 0) {
    log.message ("TEST", Log::Critical, 0,
           "Server initialization failed with error %d.",
           -msrvResult);
    return EXITVAL_INIT_FAILED;
}
```

Finally, the `listen()` needs to be called to start listening.

```
/* Enter the listener loop. */
msrvResult = myServer.listen ();
if (msrvResult < 0) {
    log.message ("SMPLLIST", Log::Critical, 0,
                 "Server execution failed with error %d.",
                 -msrvResult);
    return MSRVTEST_RETVAL_EXEC_FAILED;
}
```

The `listen()` will return after the server has shut down.

5.3.3. Shutting down

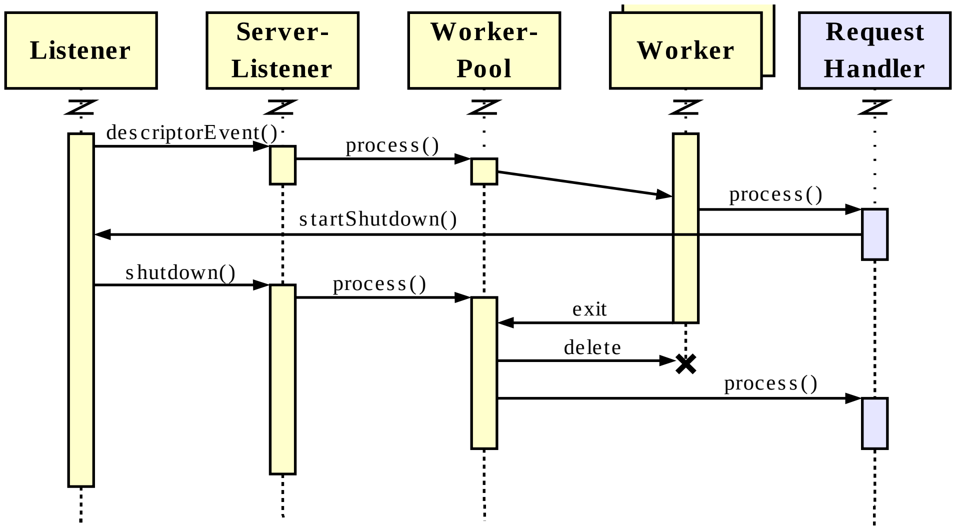
Shutdown of a server is a delicate procedure, especially on threaded servers where threads may be processing requests while the order to shut down occurs.

Shutdown can be initiated by a request handler. The are two very simple alternative ways to initiate shutdown. First, the handler can return a special error code, `MSRVERR_SHUTDOWN_EVENT`, to notify the ***ServerListener*** that it should go down. Second, the handler can call the `startShutdown()` method of ***Listener***.

When the ***Listener*** goes to shutdown state, it immediately stops listening for socket events such as new connections in the server socket or data in the client sockets. It notifies about the shutdown event to ***ServerListener***, which results in a shutdown request being sent to the request handler, which is either the user-defined handler or ***WorkerPool***.

If ***WorkerPool*** distribution manager is being used, it will shut down all the worker threads when it receives the shutdown request. The threads shut down after the request queue has been processed. After that, the ***WorkerPool*** calls the user-defined request handler in the main thread to process the shutdown request.

Sequence diagram below illustrates the initiation and handling of shutdown on different levels of worker thread architecture.



After the shutdown request has been processed, ***ServerListener*** closes any remaining open client connections and then the server socket.

Finally, ***Listener*** returns from the `listen()` method.

5.4. Other programming issues

5.4.1. Listener timeouts

Setting a ***Listener*** timeout is important, because when a ***Listener*** is waiting events on sockets, it cannot handle events from other threads. ***Listener***s normally awaken only when there arrives new connection request or data arrives in a socket. This is typically relevant for shutdown. If a request processor working in some thread processes a shutdown command, it must put the ***Listener***s in shutdown state and then wait for it to awaken and actually start the shutdown.

The default timeout for ***Listener*** is one second.

Chapter 6 Examples

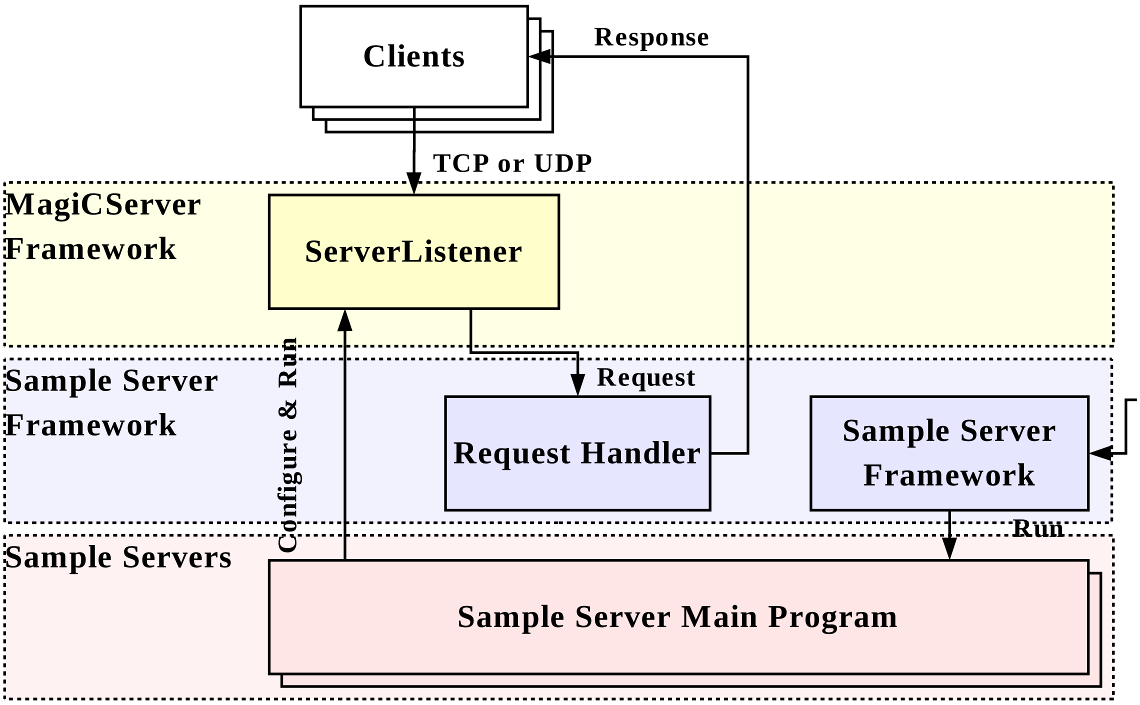
This chapter provides descriptions of some simple server examples implemented with MagiCServer++.

6.1. Overview

Two sample server applications that use MagiCServer++ library are provided:

- Single-thread listener architecture (`smrvsample_listener`)
- Worker thread architecture (`smrvsample_worker`)

The actual application logic of the sample servers is identical and implemented in a common library that acts as a framework for the sample servers. This architecture is illustrated in the figure below:



The sample server framework implements a *main()* program that parses command-line arguments and calls the actual sample server main program. The main program creates and initalizes the request handler, log, and a listener, and then runs the listener.

6.1.1. Directory hierarchy

Below is the directory tree of the examples directory with all the contained files. The three modules, the library module and two sample application modules, are emphasized.

```
<topdir>
└─examples
    ├─examples.mk                                 Examples recursive makefile
    ├─libmsrvsample                               Common sample library
    │   ├─libmsrvsample.mk                        Makefile
    │   ├─include                                 Header files
    │   │   ├─msrvsamplemain.h                    Common definitions
    │   │   └─msrvsamplehandler.h                 Sample request handler class
    │   └─src                                     Source files
    │       ├─samplemain.cc                       Main program
    │       └─samplehandler.cc                    Sample request handler
    ├─msrvsample_listener                         Single-thread listener server
    │   ├─msrvsample_listener.mk                  Makefile
    │   └─src                                     Source files
    │       └─samplelistener.cc                   Server implementation
    └─msrvsample_worker                           Worker thread server
        ├─msrvsample_worker.mk                    Makefile
        └─src                                     Source files
            └─sampleworker.cc                     Server implementation
```

The makefiles use the MagiCBuild build system to recursively compile the modules and their submodules.

6.2. Common sample server library

This tiny library implements all the application logic of a server by subclassing the ***RequestHandler*** class and reimplementing the `process()` method.

This sample server library forms an application framework that is used by the actual sample server applications presented below.

6.2.1. Main program

The library acts as an application framework by implementing the `main()` program internally. When the program starts, it parses the command-line arguments and then calls a `serverMain()` function with those parameters. The `serverMain()` function must be defined by the user of the library or otherwise a linkage error occurs.

```
int main (int argc, char* argv[])
{
    int        exitValue = 0;
    TestArgs   args;

    /* Parse command-line arguments. */
    exitValue = parse_cmd (argc, argv, args);
    if (exitValue)
        return exitValue;

    try {
        /* Initialize and run the server. */
        exitValue = serverMain (args);
    } catch (std::runtime_error& e) {
        fprintf (stderr, "Exception caught "
                         "at main level: %s\n",
                         (const char*) e.what ());
    }

    return exitValue;
}
```

The main program also catches any uncaught exceptions and prints the error message stored in the exception object to standard error.

6.2.2. Command-line parser

The command-line parser parses the command-line arguments given to the application and sets the appropriate configuration parameters in the ***TestArgs*** struct. The struct is passed to the `serverMain()` main program of the actual sample server.

The command-line parser accepts the following options:

Option	***Description***
`-h`	Prints help.
`-d`	Runs the server as a daemon by detaching it from the terminal. The standard output will be redirected to `/dev/null`.
`-udp`	Creates an UDP (User Datagram Protocol) server socket instead of the default TCP socket.
`-p` *`<portno>`*	Sets the port number of the server. The default is 1234. The port number must be greater than 1000 for normal users; only `root` can run the server with a smaller number.
`-l <logfile>`	Sets the log file. If the log file name is not given with this option, the log is written to standard output.

6.2.3. Application logic

The actual application logic or "business logic" is implemented in the ***MyHandler*** event handler, in file `examples/libmsrvsample/src/samplehandler.cc`.

The request handler does the following actions when it receices requests:

Request	***Action***
New connection	Send welcome message to new client
Data (message)	Both TCP and UDP requests are handled in the same way. If the message is a "quit" or "shutdown" command, it is executed. Otherwise, send a response to the client and a notification to all other clients.
Connection lost	Nothing (except write to log)
Shutdown	Send a shutdown notification to all connected clients.

6.3. Single-thread listener server

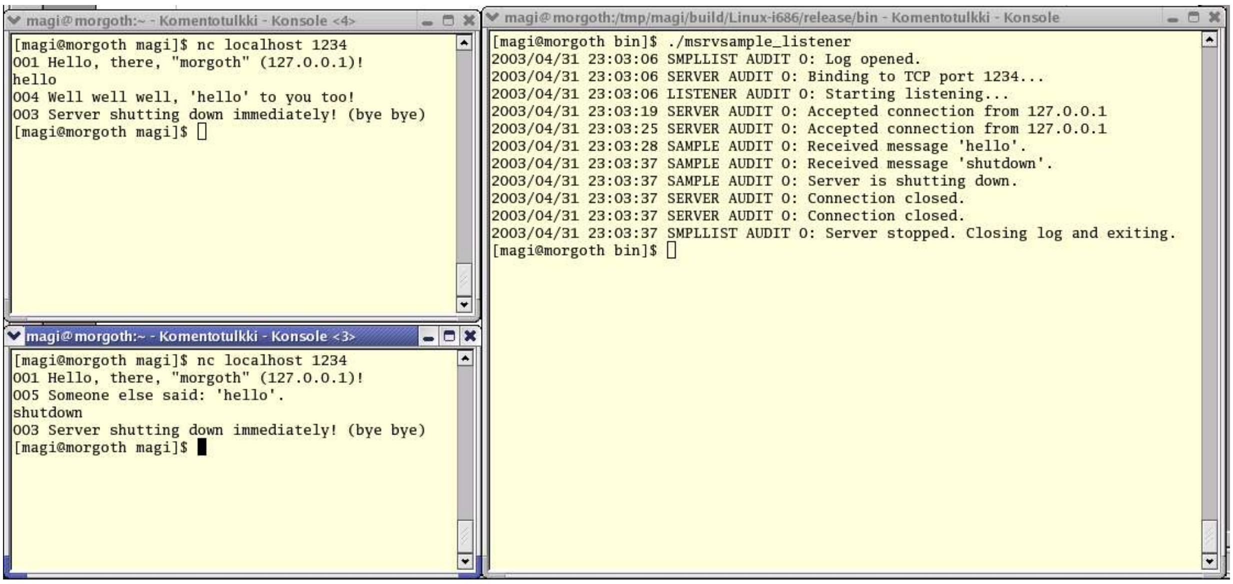
The sample server based on a single-thread listener architecture uses the sample server framework described above and implements the `serverMain()` function that initializes the ***ServerListener*** and then runs it.

```
int serverMain (const TestArgs& args)
{
    MSrvResult msrvResult = 0;
    int        exitValue  = 0;

    /* Open log to file or standard output. */
    LogFile log (args.logfile? args.logfile : "-");
    log.message ("SMPLLIST", Log::Audit, 0, "Log opened.");

    /* Put process in background, if requested. */
    if (args.daemonize)
        if (daemon (0, 0) < 0) {
            log.message ("SMPLLIST", Log::Critical, 0,
                  "Daemonization failed with error %d; %s.",
                  errno, strerror (errno));
    }

    /* Create transaction handler. */
    MyHandler myHandler;

    /* Create and configure server object. */
    ServerListener myServer;
    myServer.setLog (log);
    myServer.setHandler (myHandler);

    /* Create a server socket and bind it to an address. */
    msrvResult = myServer.bind (args.portno,
                         args.udp? ServerListener::UDP :
                         ServerListener::TCP,
                         0);
```

```
    if (msrvResult < 0) {
        log.message ("SMPLLIST", Log::Critical, 0,
                "Server initialization failed with error %d.",
                -msrvResult);
        exitValue = MSRVTEST_RETVAL_INIT_FAILED;
    }

    if (msrvResult >= 0) {
        /* Enter the listener loop. */
        msrvResult = myServer.listen ();
        if (msrvResult < 0) {
            log.message ("SMPLLIST", Log::Critical, 0,
                    "Server execution failed with error %d.",
                    -msrvResult);
            return MSRVTEST_RETVAL_EXEC_FAILED;
        }
    }

    /* Server has stopped. */
    log.message ("SMPLLIST", Log::Audit, 0,
                "Server stopped. Closing log and exiting.");

    return exitValue;
}
```

6.3.1. Sample session

Below is a screenshot of a sample session with the single-thread listener sample server. On left, we have two client sessions, and on right, we have server run with log printed to standard output.



The example uses the 'nc' (netcat) system utility as a client application to connect to the server. The following steps can be observed:

1. The server is started in the right window and is bound to TCP port 1234.

2. Both clients try to connect to the server.
3. Server accepts the connections and greets the clients with a welcome message (001) containing the host name (here `morgoth`) and IP address (here `127.0.0.1`) of the client.
4. The client in upper left window sends a "`hello`" request to the server.
5. Server receives the request, sends a response (004), and also notifies the other client with a message (005).
6. The client in lower left window sends a "`shutdown`" request to the server.
7. The server initializes shutdown, sends a shutdown message (003) to all the clients, and closes the connections.
8. Finally, the server stops and exits.

6.4. Sample server using worker thread architecture

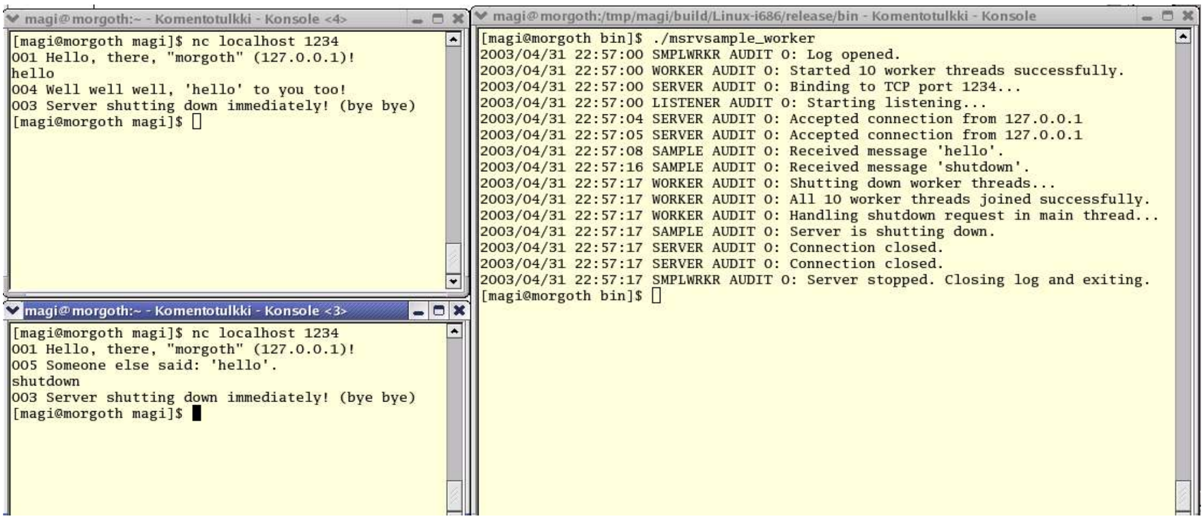
The sample server based on a worker thread architecture uses the sample server framework described above and implements the `serverMain()` function that initializes the ***ServerListener*** and then runs it.

The implementation of this architecture is almost identical to the single-thread model, except for the hooking of ***WorkerPool*** as a distribution manager between the ***ServerListener*** and the actual request handler.

```
...
/* Create transaction handler. */
MyHandler myHandler;

/* Create a worker thread pool with 10 worker threads. */
WorkerPool workers (myHandler, log, 10);

/* Create and configure server object. */
ServerListener myServer;
myServer.setHandler (workers);
...
```

6.4.1. Sample session

Below is a screenshot of a sample session with the worker thread sample server. On left, we have two client sessions, and on right, we have server run with log printed to standard output.



The execution of the server goes exactly as in the single-thread listener case above, except that we can see in the server log that the server starts 10 worker threads in the startup, joins them during the shutdown, and processes the final shutdown request in the main thread.

Chapter 7 MagiCServer++ development

This chapter gives information for developers interested in making changes to the MagiCServer++ framework.

7.1. Build system

The *MagiCBuild* build system is used for compiling MagiCServer++. The build system consists of a framework of makefiles for GNU Make, and a configuration script. MagiCBuild has a user interface very similar to the commonly used configuration scripts generated with GNU Autoconf and GNU Automake.

The build system is currently undocumented.

7.2. Reference documentation

Reference Manual includes class documentation for all the C++ classes in the library. It is generated with *Doxygen* documentation generator. Doxygen reads the source and header files and generates documentation in HTML and PDF formats, as well as man pages.

The Reference Manual is generated with make target "`make dox`".

7.3. Coding conventions

MagiCServer++ source code and headers follow a few conventions, as provided below. Also other coding conventions commonly used in Linux and UNIX C++ and C programming should be used.

7.3.1. Code formatting

Indentation depth of 4 is used. This is usually done with tabs with the tab length set as 4 characters, but it can be done with spaces too. Opening brace is always written in end of the line, except for the first brace in function. For example:

```
void myFunction ()
{
    if (a = 1) {
```

```
        hello();
    }
    else { /* Else statement on separate line. */
        foobar ();
    }

    switch (42) {
      case 42: { /* We use block here for local scope. */
          int i = 42;
          foobar ();
      } break; /* Break statement here. */
    }
}
```

7.3.2. Code comments

Classes, methods, and functions are commented using Doxygen compatible notation. These code structures are preceded with a C-style comment block with 80 characters wide upper and lower border made with asterisks:

```
/*************************************************************
 * Brief description.
 *
 * Longer description, which may take
 * several lines.
 *
 * \return Return value of method or function.
 ************************************************************/
```

All Doxygen directives are available inside these comment blocks, as applicable for each code structure type.

A typical class documentation would be as follows:

```
/*************************************************************
 * Brief description of the class ...
 ************************************************************/
class MyClass : public BaseClass {
  public:
    void someMethod (int anArg, bool otherArg);
  private:
    int mMyVariable; /**< Documentation for the variable. */
};
```

Methods are not documented in headers, but in source files. Doxygen supports this form of documentation seamlessly.

```
/*************************************************************
 * Brief description of the method ...
 *************************************************************/
void MyClass::someMethod (
    int  anArg,    /**< Documentation for anArg.    */
    bool otherArg) /**< Documentation for otherArg. */
{
   ...
}
```

Code comments are made as follows:

```
{
   int  localVariable = 0;     /* Descr. of the variable 1. */
   bool otherVariable = false; /* Descr. of the variable 2. */

   /* Comment for some code part. */
   int result = functioncall (argument1,  /* Comment. */
                              argument2); /* Comment. */
}
```

It is not really relevant whether the source code comments are made with C or C++ style comments. The C comments may appear more clear than C++ comments. All successive comment lines are intended at the same level, that is, their lining should not appear ragged.

7.3.3. Naming conventions

Class and structure names begin with upper case letter. Function and method names begin lower case. Words are indicated with capitalization of the first letter of each word.

Names of constant values, typically defined as macros, are written in all upper case.

Variable names

Variable names begin lower case. They do not have any prefixes indicating type ("Hungarian notation") of the variable, except for the scope, ownership, and reference type. The following prefixes apply:

Prefix	***Example***	***Description***
m	int mMember	Member variable in a class
p	int* pPointer	Pointer to an owned object
r	int& rReference	Reference
rp	int* rpPointer	"Reference" pointer to not object not owned
s	static int sVariable	Static variable

Ownership means basicly responsibility of destruction; the owner of an object has the responsibility to destroy it when it itself is destroyed. References (as in int&) are never owned by the referencing object, and the same meaning of reference applies to pointed objects not owned.

The prefixes can be combined in the following ways:

Prefix	***Example***	***Description***
mr	int& mrReference	Member reference variable
mp	int* pPointer	Member pointer to an owned object

Prefix	***Example***	***Description***
mrp	int* mrpPointer	Member pointer to an object that is not owned
smp	static int* smpVariable	Static member pointer to an owned object

...and so on.

The actual type of variables and constants should be clear from the context. Variables can have a natural type specifier as postfix. For example:

Type	***Example***	***Description***
Socket	socket	Class name as variable name, when no semantics are bound to the variable.
Socket	clientSocket	Semantics of the variable are given in prefix, class name as suffix.
Array<Thread>	threads	Array type indicated with plural suffix (s)
int	threadCount	Quantities indicated with "Count", or if semantics are clear, with plural suffix "-s".
bool	mIsShutdown	"Is" indicates truth value

Chapter 8 Known bugs and limitations

8.1. Bugs

MagiCServer++ has the following known bugs.

- The build system prints some shell execution errors. This is a problem of the build system.
- The `gethostbyaddr()` function used in the sample request handler is not thread safe because the pointer it returns refers to a static data structure. Corruption may occur if two threads use the function within a very short time window.
- There is a small time window between the time a ***Worker*** checks if the ***Listener*** is in shutdown state and going to wait state. If the shutdown broadcast occurs in this window, the worker won't know about it and goes to eternal sleep.
- There probably are a few memory leaks, as the software has not been tested for those.
- *Reference Manual* is rather messy and contains many unwanted entries. This is due to limitations of the Doxygen documentation generator.
- Error checking, especially for out of memory situations, is not complete.

8.2. Limitations

MagiCServer++ has the following limitations.

- ***ServerListener*** can handle only one listening server socket. This limitation can be circumvented by running multiple ***ServerListener***s in multiple threads, while using a common request handler.
- Some relevant signals should be handled. Especially, a signal that would awaken the ***Listener*** from `select()`, for example when a request handler has ordered shutdown. Currently this situation is handled with a `select()` timeout.

- There is no option in ***Log*** to open log to system log (syslog).
- The server port number can not changed and the server socket can not be reconfigured without shutting down and restarting the server.
- Shutdown doesn't allow a forced shutdown that ignores the remaining requests in request queue.
- Changing the distribution architecture run-time is not supported, though it possibly can be done.
- There is no mechanism to terminate a runaway thread.
- Inheriting ***Connection*** and ***ConnectionFactory*** objects is not demonstrated in examples.
- STL is still used for the runtime-error exception class.

Chapter 9 GNU Free Documentation License

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simpleHTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generatedHTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to aTransparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Alphabetical Index