How to Use GSK

Dave Benson

August 24, 2005

Chapter 1

Preface

1.1 Who Should Read This Book

This book is geared toward the programmers of servers; it covers both design and implementation.

It assumes familiarity of a number of subjects:

1. C programming. I recommend the revised edition of Kernighan and Richie's (sp?) classic text, *The C Programming Language*.
2. object-oriented programming, either in C (preferable) or with C++ or a similar language. Stoustrap's (sp?) text *The C++ Programming Language* is suitable.
3. socket programming. I recommend Richard Steven's *Advanced Programming in the Unix Environment*.
4. design patterns. We will generally describe ideas in high-level terms. As appropriate we will borrow nomenclature from the book *Design Patterns* by Gamma,...; in other cases, we will use the idea of a *pattern language*: for example, later we will describe and name general attributes of event-handlers. Though *Design Patterns* is highly recommended, it is probably not really essential for understanding this book.

1.2 Availability

GSK can be downloaded from:

`http://gsk.sourceforge.net/`

Chapter 2

What GSK Is All About

GSK is a set of classes designed to help implement very efficient servers in C.

Writing a server involves handling many clients at once. Many libraries exist to help you write servers using threads; GSK is not like that, with it you will write servers that use *callbacks*.

It is our belief that servers implemented using threads are fundamentally slower that systems which use callbacks, for reasons we will discuss eventually.

However, for many simple applications, threads are easier for a number of reasons. First, they mirror procedural descriptions of the problem: they are straightforward to-do lists for an easy server request. This works great when all the threads are running independently, for example, for a static webserver.

However, if there are interactions between clients (and even something as simple as caching can be turn out to be quite subtle in a threaded environment), then you will have to design objects that are thread-safe to manage interactions between threads. Often, specifying a good interace for these objects is difficult. And their implementation depressingly often have bugs of the worst type: bugs that only manifest rarely.

Ignoring threads, using callbacks, is many ways much easier. There is no other thread that can interfere with your data structures, and yet, there is no chance of deadlock.

In essences, programs using callbacks are implicitly defining a *state machine*: you can define a state as the set of callbacks that might be invoked next, and the callbacks are transitions between such states. Sometimes state diagrams are the easiest pattern for describing a callback-driven design.

Another way to think about callbacks is that they are *event-oriented*. The callbacks are event-notification. The callback was registered earlier to the object which is responsible for dispatching the event. For example, if you make a request with the HTTP client object, you provide a callback that will notify you in the event that the HTTP response header has been received.

GSK inherits its object-oriented programming system from glib, a library which makes programming in C much nicer. We will describe glib and its object system shortly in some detail, since it underpins the rest, and is fairly easy to understand.

Chapter 3

Server Design Basics

This section provides an almost non-programmer view of servers, clients and all that.

Before involving yourself writing a performance server or client you will probably need to define the protocol that's being spoken. This brief chapter will roughly define the space by describing the facilities provided by unix-like sytems, as well as describing some general terminology.

3.1 Portable Unix I/O

In general, GSK tries to be portable– but the role of a server inevitably boils down to the support of the operating system.

We don't support all POSIX i/o systems: message-queues, semaphones and shared-memory are all left to the user. And we don't have any reason to support mmap(): its interface is fine.

The system-level primitives that we support are pipes and duplex pipes, "listeners" that accept streams, and datagram connections.

1. TCP.
2. UDP.
3. Unix-Domain Sockets.
4. pipes (and socket-pairs).

3.2 Jargon

TODO: we need to write out descriptions here, but i'm lazy for now, and i'm just collecting vocab here.

1. *transaction request response*

2. *server client*
3. *peer*
4. *proxy*

Chapter 4

`glib` and `gobject`

GSK derives much functionality from glib: we use its lists, hash-tables, file utilities, string utilities, and so on. Most of this stuff is very easy to understand: it's conceptually simple and well-documented. In fact, glib's extreme legibility has been inspirational to GSK development.

Everyone agrees that `glib` is a good idea, but perhaps it would be good to mention some weaknesses:

1. Does not handle out-of-memory. That is, any glib internal function that has malloc() fail will immediately terminate with a error message. In some cases, `g_try_malloc()` can be used to avoid the fatal error messages, but it's only worth it for small allocations.

   Our general attitude is: design your program to avoid OOM conditions. Support a memory-usage limit on arbitrary size structures, like caches, and set them conservatively. Remember that most strategies don't work either: they are seldom tested. At one company I worked for, every C++ allocation was guarded:

   ```
   s = new S;
   if (s == 0)
     return ERROR_OUT_OF_MEMORY;
   ```

   but it turned that the C++ compiler would actually throw an exception on out-of-memory. So all those checks were just useless, untested code, which apparently no-one had every noticed. Which just shows that a system in out-of-memory condition may be best served by terminating instead of handling out-of-memory piecemeal throughout your program.

4.1 `GObject` and the type-system

An extension to `glib` available since version 2.0, is the `gobject` library, which provides a framework for object-oriented programming in C. In addition to providing a abstract-base-class `GObject`, it provides a general system for specifying types, values, and parameter-specifications.

Now, 90% of people who hear this ask: why? In the end, no single reason seems to suffice:

1. `gobject` provides a lot more functionality that the standard OO system: in particular its support for language bindings, parameters, introspection, and event-notification is *much* richer that C++'s object.

2. Some people like C aesthetically. Personally, I like that it is a language which is designed to model the machine. That makes it very easy to analyze, and in many cases it actually makes debugging easier to. (The lack of safety can be a challenge, though)

3. C++, which is the other performance and portability leader, is a relatively mixed up language, because it tries to meld C with the abstraction of OO programming. Or perhaps for some other reason. In any event, plenty of anecdotal and personal evidence suggests that C++ leads to much subtler bugs than C. It's an awfully complex language– my bug counts reduced dramatically when switching back to plain C, several years ago.

4. `gobject` provides a debugging mode to allow for quick debugging of run-time casts and reference-counting. That helps compensate for lack of safety in C.

But there's no reason to debate now. GSK 1.0 requires and uses GObject. For now, we believe that all future versions of GSK will use GObject.

Now it would be useful to give an exact list of the functionalities of a `GObject`.

1. A `GObject` is *ref-counted.* That means that it maintains an integer which is initially 1, and can be incremented or decremented. When the "ref-count" becomes 0, the `GObject` is freed.

2. Weak-references. Sometimes you want a reference to an object that does not lock the object in memory. Instead a callback is invoked when the object is freed. Because locking an object is memory is usually desirable when you're trying to do something with it, and because supplying a callback function is annoying, weak-references are far rarer than the strong ref-count.

3. Named, typed parameters. Each parameter comes with a `GParamSpec`, which specifies the type of the parameter, the default value, and other constraints on the parameter: for numeric types, a range of allowed values can also be specified.

Object-oriented programming is jargon-filled. Here we'll define some common programming jargon, which you can try to understand what the OO-theory types are talking about,[1] should you find yourself stranded with one of them, at a cocktail party or desert island or something... [STUPID or FUNNY?]

1. **instance**, **object**. A thing living in a running copy of your program. Typically it is known by its memory address. It has some sort of type, and some sort of methods that can be run on it, and probably a class: if it doesn't have that, these terms connote too much utility, Perhaps you have a structure or record on your hands.

2. **class**. Crawling out of some philosophical corner, comes the notion of an object's class. A class keeps track of data which is common for all instances of an object in a certain program. At times, class' appear to exist outside the program. But they don't. Or maybe they do, who knows?

   Classes may keep track of data such as: methods, member names and types, parameters, virtual functions, parentage, interface information, etc.

3. To **derive** or to **subclass** mean to define a new type of object that has all the properties of another class or classes. The new type is called the **subclass** and the old class is the **superclass**. We also say that the subclass **inherits** from its superclass.

4. **member**. A piece of data which lives in the memory allocated to an object or structure.

5. **method**. A function that can be invoked "on an object", which really just means that an object instance is passed in.

6. **virtual method**. A function that can redefined in a subclass.

7. **type**. A description of the format of a value. In `glib`, types have names. There are several types of type, described next.

8. **object type**. The type of an object. Usually, the value of an object type is a reference to the object.

9. **recursive type**. A type which defined in terms of another type: an array of X, a hash-table of Y, or a W of Z. (gobject doesn't really support this.)

[1]Good luck.

Chapter 5

The `GskStream` Class

This chapter will cover the most important class in GSK, `GskStream`.

GskStream is modelled after a unix process: it can be read from and written to, has an error reporting mechanism, and can be suspended due to its inputs or outputs.

It gets suspended if it cannot do anything without more data, and there is no more data on its standard input. Likewise, it gets suspended if it has written more data than its standard output can consume (or more precisely, if the buffer in between has grown too large).

However, the difference is that unix processes essentially use the threading model. Each process has its own execution context and instruction pointer. The operating system is required to suspend and resume them.

GskStreams are callback-driven instead. A user of a GskStream can trap when the streams are ready for input or output: the `gsk_stream_read` and `gsk_stream_write` calls never block.

Occasionally you will want to trap these events yourself, using `gsk_stream_trap_readable()` and `gsk_stream_trap_writable()`. More likely, you will `gsk_stream_attach()` streams together, so that the output of one stream feeds into the input of another.

The remainder of this chapter will describe how to use `GskStream`s in more detail, as well as how to implement your own `GskStream`.

5.1 Using `GskStream`

The most low-level way to use a `GskStream` is the following four functions:

1. `gsk_stream_read`. Read data from the stream object.
2. `gsk_stream_write`. Write data to the stream object.
3. `gsk_stream_trap_readable`. Provide functions that will be called when the stream has data available for reading, and when the stream is shutdown-for-reading (for example on end-of-file).

4. `gsk_stream_trap_writable`. Provide functions that will be called when the stream has data available for reading, and when the stream is shutdown-for-reading (for example on end-of-file).

Here is an example that will print the number of characters in a stream once it has shut down:

```
gboolean
handle_readable (GskStream *stream,
                 gpointer   data)
{
  gsize *stream_length = data;
  gchar buf[8192];
  GError *error = NULL;
  *stream_length += gsk_stream_read (stream, buf, sizeof (buf), &error);
  return TRUE;
}
gboolean
handle_read_shutdown (GskStream *stream,
                      gpointer   data)
{
  return FALSE; /* cause the hook to be untrapped */
}
void
handle_hook_destroyed (gpointer data)
{
  gsize *stream_length = data;
  g_message ("length of stream was %zu bytes", * stream_length);
  g_free (stream_length);
}
void
start_computing_length (GskStream *stream)
{
  gsk_stream_trap_readable (stream,
                            handle_readable,
                            handle_read_shutdown,
                            g_new0 (gsize, 1),
                            handle_hook_destroyed)
}
```

Now, at first this code seems pretty excessively complicated compared with the equivalent threaded code; written using stdio like:

```
gsize compute_length (FILE *stream)
{
  gchar buf[8192];
  gsize rv = 0;
  gsize read_rv;
  while ((read_rv = fread (buf, 1, sizeof (buf), stream)) > 0)
    rv += read_rv;
  return rv;
}
```

But the advantage to the GSK mechanism is the extremely low resource consumption: to compute the lengths of 1000 streams at the same time you would have to use 1000 threads in the latter code. On most operating systems this is *much* more expensive than merely multiplexing on 1000 file-descriptors. Furthermore, it puts less burden on the scheduler, etc.

By far the most common way of using streams is to `gsk_stream_attach` them together. This is much easier than directly trapping the ends of the streams, and implements contention-handling for you.

5.2 Exercises

1. Write a function that takes two streams, attaches to their read-end, and writes out the XOR of the data-streams.

Chapter 6

Brief Tour of Available Stream Types

Streams come in a few different classes. There are those that model resources, like file-descriptor-based and memory-based streams. There are those that filter content, like the zlib-compression and ssl-encryption streams. And there are those that "speak" protocols, like the HTTP client and server streams. Another important stream is the queue-stream, that reads or writes from a list of streams in order.

6.1 Resource-Based Streams

Almost every server eventually talks to streams that represent OS resources: the typical examples are file-descriptors and memory.

The file-descriptor stream is `GskStreamFd`. It breaks into a few categories. Some file-descriptors are readable and some are not. Some file-descriptors are writable and some are not. Some file-descriptors are currently forging a connection to a remote host. Some file-descriptors can be polled using `poll()` or `select()` or equivalent, some, like ordinary files, cannot. Some file-descriptors can be partially shutdown using the `shutdown()` system-call, like others cannot.

`GskStreamFd` can represent all these situations. To make streams that represent files use

```
gsk_stream_fd_new_read_file
gsk_stream_fd_new_write_file
gsk_stream_fd_new_create_file
```

which return a `GskStreamFd` and take a filename as a parameter.

If you already have a file-descriptor, use:

```
gsk_stream_fd_new
gsk_stream_fd_new_connecting
```

These methods assume that you know if the stream is readable, writable, etc. If you don't, you may wish to call

```
gsk_stream_fd_new_auto
```

which uses extra system-calls to figure out these things.

Another important class of stream are the various memory-based streams. The functions which create readable streams are:

```
gsk_memory_slab_source_new
gsk_memory_source_new_printf
gsk_memory_source_new_vprintf
gsk_memory_source_static_string
gsk_memory_source_static_string_n
```

The `buffer_source` drains the data out of a `GskBuffer` without copying it. The `slab_source` will not copy the data, instead it will call a hook with it is done with the data: this is useful for serving cached data. The `printf` source is for quick-and-dirty sources that are typically fairly short (`vprintf` is the same, except it takes a `va_list`, which can be used for chaining). The static-string sources also avoid copying the data: they are based on the premise that this data is permanently available.

There are also methods that create streams that read from or write to `GskBuffer`s, which are just in-memory buffers for binary data.

```
gsk_memory_buffer_source_new
gsk_memory_buffer_sink_new
```

Both of these methods take can advantage of `GskBuffer` features to avoid copying data.

6.2 Filter Streams

Filter streams are streams which convert one binary format, their input, to another binary format, their output.

In most cases, filter streams are primarily algorithmic.

`GskZlibDeflator` compresses any data in zlib-deflated format. `GskZlibInflator` decompression zlib-deflated files.

The SSL stream is a bit different, but still essentially a filter-based stream. It decrypts its input and encrypts its output, using a given stream as the presumably insecure transport.

Of course, the external-process stream, `GskStreamExternal` can also act as a filter, but you'll have to be wary of the subprocess buffering its output– instead of line-by-line output as you might want for a process like `grep`, you'll probably get data in blocks the size of your system's pipes' buffers.

6.3 Protocol-Handling Streams

A very important type of streams “speaks” a protocol and translates it into higher-level events. The exact interfaces defined by the stream tend to depend on what the protocol offers.

Some protocols are *client-server*-oriented. That means that there is a many-to-1 relationship between certain entities, and there may exist a fundamental asymmetry in the protocol. For example, in HTTP, the client always initiated the transaction. In chat protocols, the client ... In our terminology, a client functions as a stream that can be attached to the raw transport (or an SSL transport perhaps) can speaks the protocol, a “user-agent” if you will. Likewise a server is connected to the transport and allows the user to flexibly and programmatically handle the client connections.

Most other protocols are *peer-to-peer*. Typically there is a 1-1 or many-many relationship between the peers. These are protocols for talking directly between users, or backend machine talk. One-to-one communication can be done using TCP. Many-to-many communication is not streaming: you must use UDP or many TCP connections.

6.4 Container Streams

...

6.5 Exercises

1. Write a function that takes printf-like arguments and returns a compressed stream:

```
GskStream *my_compressed_stream_printf (const char *str,
                                        ...) G_GNUC_PRINTF(1.2);
```

Chapter 7

Implementing `GskStreams`

Before considering whether to implement a stream, you should probably make sure that an existing stream cannot meet your needs:

1. read-only and write-only streams with relatively small content can be handled by the memory-based streams.

2. more complicated streams that have no methods other than the `GskStream` methods, can often be implemented with the `GskBufferStream`.

In general, see the chapter about specific streams to learn more available classes.

But if your streams has methods specific to it, then you probably want to define a new `GskStream` class.

7.1 Designing Your Stream's Interface

It would be a tragic mistake to tell you how to implement a new `GskStream` without telling you how to design its interface.

The design of the stream should be centered around its interface (the methods and members of the stream). The interface of the class should be based on how you plan on using it (or, if you aren't using it, you should pretend you are, or talk to someone who will.).

There are many choices to be made designing your stream:

1. is it writable? if so, what type data does it expect as input?

2. is it readable? if so, what type data does it produce as output?

3. what aspects of the stream are configurable? Sometimes GObject-parameters are appropriate. At other times, set/get methods are more convenient. Still at other times multimodal methods exist, for example, a parse class may have multiple, mutually exclusive modes. The easiest way to configure that is by having a method per mode.

4. does the stream have "events" happen that other people will be interested in? If there is a 1-to-1 correspondence between those who emit events and those who catch events, then `GskHook` may be suitable for your event handling needs. If there is a 1-to-many correspondence then `GObject`'s signals may be apprорriate. See the chapter on Event Handling methods for more details.

5. what kinds of information can be queried about the stream's status?

6. is there a standard dictating what features are expected?[1]

For error-handling, you probably should go through the error methods provided by `GskStream`.

In fact, when I write new `GskStream`s, I usually write a program that uses that stream first, in order to flush out all the interface I will need. Never forget that you are not allowed to block on I/O, that is a very important part of the design.

In a similar vein, the design should be **minimal**– you should not include a feature unless you can imagine a situation in which it would be used. (and it should be a clean solution to the imaged situation to) It is *much* easier from a maintainance viewpoint to extend an existing interface that to change it, so, if in doubt, don't include it.

7.2 Designing Your Stream's Implementation

After you have designed your stream's interface, you can usually easily **design its implementation**.

Here are some standard implementation concerns:

1. do you need to do buffering? A few sources operate with absolutely no buffering of data, but most sources will find buffering convenient.

2. how do you wish to notify the user of things that happen without them invoking a function explicitly? Some sort of callback mechanism is required, see the Event-Handling Mechanisms for options in this regard.

3. Make sure you obey the restrictions in "`GskStream` Contracts".

7.3 `GskStream` Shutdown

One subtlety of handling streams is determining how they are terminated. Essentially all streams can be shutdown either by the user of the stream by calling

[1]But we've been known to generalize the standard a bit. For example, `GskXmlrpcStream` supports either side sending any number of requests and getting any number of responses, and over any transport. The XML-RPC specification is more restrictive: it states that an XML-RPC transation is a single request issued as POST-data over HTTP, with the response given in the body of the HTTP response.

```
gsk_io_read_shutdown
gsk_io_write_shutdown
gsk_io_shutdown
```

or by the stream's implementation, which will use methods like:

```
gsk_io_notify_read_shutdown
gsk_io_notify_write_shutdown
gsk_io_notify_shutdown
```

The user can trap the shutdown event in either direction, the second function pointer arguments to `gsk_stream_trap_readable` or `gsk_stream_trap_writable` are functions to be called with the stream is shutdown (either by the user or the implementation).

One subtlety to understand is that it is OK to call the various notify-shutdown functions from your raw-read and raw-write implementations.

For most users, they are calling `gsk_stream_read` or `gsk_stream_write` from within a trap-readable or trap-writable handler. The implementation defers shutdown until after the handler is done. To understand why this is important, consider the code:

```
gboolean handler (GskStream *stream, gpointer data)
{ char buf[1024];
  GError *error = NULL;
  g_message ("read %u bytes", (guint)gsk_stream_read(stream,buf,sizeof(buf),&error));
  return TRUE; }
gboolean shutdown_handler (GskStream *stream, gpointer data)
{ g_message ("shutdown");
  return FALSE; }

stream = gsk_memory_source_static_string ("hello");
gsk_stream_trap_readable(gsk_stream,handler,shutdown_handler,NULL,NULL);
gsk_main_run();
```

as written, this will correctly print the text:

```
Message: read 5 bytes
Message: shutdown
```

However, the memory source actually called `gsk_stream_notify_read_shutdown` from within the `gsk_stream_read`! The reason the printouts came in the correct order is that the shutdown handler is never run until the running handler is done.

QUESTION: If you change the

```
gsk_main_run();
```

to

```
handler(stream, NULL);
```

which order will the printouts come in?

7.4 GskStream Contracts

Your stream should obey the following requirements:

1. none yet

7.5 Exercises

1. Make a stream class that takes an array of streams and a function to combine equal lengths of raw binary data. Don't forget to manage the cases where one stream is much faster than another. Here is the header file for this project:

```
typedef void (*MyByteSlabCombiner)(guint          n_slabs,
                                   guint          slab_length,
                                   const guint8 *const *slabs,
                                   guint8        *output,
                                   gpointer       combiner_data);
GskStream *my_byte_slab_stream_new (guint n_streams,
                                    GskStream **streams,
                                    MyByteSlabCombiner combiner,
                                    gpointer       combiner_data);
```

Chapter 8

Event-Handling Mechanisms

There are basically three event-handling mechanisms: `GskHook`, `GObject`'s signal mechanism, and custom event notification mechanisms.

`GskHook` is convenient for mechanisms where efficiency is important and notifications are on a 1-to-1 basis. An important aspect of hooks is that they only notify when they are ready, but they can also be turned on and off by the user who traps the hook.

`GObject`'s signal mechanism is good for some 1-to-many notifications. Any number of users can trap a signal, even from language-bindings. Signal emission can include parameters. But signal emission is quite a bit slower than `GskHook` notification. Also, implementors of classes that provide signals do not have a callback for when the first user is interested in the event, or when the last user loses interest.

Please see the `GObject` documentation for how to use signals.

custom notification methods are the only method remaining for other types of problems. We do not have any code to help, really, but we do mention a few common design patterns the section "Designing Custom Notification Methods".

8.1 GskHook

GskHook deals with a particular type of event notification. It is designed to permit i/o-contention by allowing itself to be blocked by the user. It is designed for events where only one owner makes sense.

A GskHook is also a member of a GObject; it cannot be used outside of a GObject. It should be part of the design of your GObject type.

From the user's perspective, here's what you do with a hook.

1. You *trap* a hook: you provide callbacks that will be called when the event triggers, when the event source shuts down, and when the hook is untrapped (possibly due to a shutdown, or due to an untrap).

   The first two callbacks you provide return whether to keep the trap alive. It is recommended that the shutdown trap always return FALSE.

2. You can *untrap* a hook whenever you want, if you trapped it.

3. You can *block* and *unblock* a hook. The hook will only notify you when it is unblocked. You can call block many times; you must call the unblock method an equal number for the hook to be unblocked.

4. You can *shutdown* a hook, although some hooks may choose to ignore you. Other hooks will shutdown eventually, but may notify you a few times anyway.

From the object-implementor's perspective:

1. you define a hook's class from within your class' `class_init` function.

2. you initialize the hook from within your class' instance-`init` function. At that point you can specify a few things about the hook: is it available? does it block? does it support shutdown? can shutdown fail?

3. you may respond to a method in your `Class` usually called `set_poll`, which is passed either TRUE or FALSE, depending on whether the hook should be notifying. (For a resource-based fork, this turns on or off system-level polling; other sources may not need it.)

4. you may enable or disable *idle-notification.* Idle-notification means that the hook is notified exactly once per main-loop, but the main-loop doesn't block until there are no idle-notifications to be done.

8.2 Designing Custom Notification Methods

Of course, event-notification is a broad area– we don't pretend that `gobject`'s signals or `GskHook` will solve your problems.

Therefore, it is necessary sometimes to write your own event notification. This prospect may seem daunting, but really, C makes implementing it pretty easy. The bigger challenge is describing what you've implemented. For that, I feel it is valuable to build a design vocabulary for event notification.

So, the remainder of this section will be a rambling discussion of events, with italics indicating when I am attempting to define a term.

Commonly, event notification comes in one of two basic flavors.

One-shot events are those that occur exactly one time. Sometimes the event itself will have several possible outcomes. For example, name resolution can have several results: found, not found but no error, not found due to error, timeout. Another example is process termination: it can only happen once. Of course, one-shot events can fail to happen due to failures, power outages, etc; the point is that you are not responsible if they fail to happen.

Example: The function `GDestroyNotify` is intended as a callback for when some data is destroyed. (For example, when an signal-connection itself is destroyed). This function is always provided as one-shot. That's why:

```
g_signal_connect_data (object, signal_name, handler,
                       g_strdup (name), (GClosureNotify) g_free, 0);
```

Other Examples?

One-shot events provide an opportunity for a small optimization: there is no real need for a separate event-notification destroyed notifier: it can always be inferred. However, we suggest that unless space is very precious (because you have a lot of traps) you forgo this optimization. At the least, it's annoying because you cannot write reliably convenient code like the above `g_signal_connect_data` example.

Multi-shot events are also commonplace. (XXX: is there a better name for these?) The `gobject`-signal is based on this premise: a mouse motion, a button click, a keypress, a window opening, a sound playing, etc.

Such events firmly need handler functions for destruction, to avoid memory leaks, as it's not very convenient to track when the hook is destroyed otherwise.

In most cases, it is obvious where event emission should be done: typically there is an object which has "connect" or "trap" methods and is responsible for emitting the events as necessary.

Occasionally, it is not obvious. One common case arises when there is no real object emitting the events, because the events have a sort of global scope. `atexit` is a typical example. One pattern to consider is to define a singleton object that has these methods and responsibilities. Or you can just use globals.

Sometimes there is more than one object seemingly involved in the emission. For example, you may want to register a handler to deal with the collision of two objects, say in a video game or virtual-reality or robot-control context. Often, to handle the assorted details that arise, it is helpful to develop a manager object.

- what level of detail is allowed in filtering events pre-handler

Is the handler itself an object? What type of methods does it have? Is it refcounted?

Does it represent a request? If so, is it cancellable?

Chapter 9

`GskIO`: Generic read/write Notification

The earliest versions of Gsk had no unified read/write hooks. Once read/write hooks were implemented, it was realized that the situation of having one read hook and one write hook was very common. In particular both stream-oriented and message-oriented objects have read and write hooks. So we made `GskIO` to serve as a base-class for the two classes.

Of course, the subclass needs some sort of concrete read and write methods. In the case of streams, we use the `raw_read`, `raw_read_buffer`, `raw_write`, `raw_write_buffer` methods. The packet-queue uses just `read` and `write` which have `GskPacket` arguments.

So what do all `GskIO`s have in common?

First, and most importantly, is a uniform notification and shutdown semantics, based on `GskHook`. These semantics were described in the `GskStream` chapter, but we'll reiterate them. [TODO: write stuff]

Second, a mechanism for causing construction to fail was introduced, but it hasn't turned out very well. Basically, you can override the `open()` virtual to supply a check to see if the object is viable. Unfortunately, this makes construction a two-step process, and so tends to go ignored.

These reasons aren't really all that compelling: it would have been easy to just add the two hooks into packet-queue and stream, and made them derive from `GObject`, but it's slightly convenient to use `GskIO`.

9.1 Exercises

1. Write a `MyMidiProcessor` derived from `GskIO`, that is an abstract class for midi input and output. (MIDI is an event protocol for electronic music devices.) Then write a `MyMidiDevice` that uses a `GskStream` for its raw transport, serializing and deserializing the MIDI events to the stream.

Chapter 10

Name Service: Looking up low-level addresses from names

We believe that a *name-service* is any procedure by which a hostname is mapped to a `GskSocketAddress`.

The only implementation currently provided is `IPV4`, which uses `DNS` to perform the mapping.

A single name-resolution job is embodied in the `GskNameResolverTask` object. Its only nontrivial method is `cancel` (it has `ref` and `unref` methods as well, but they are standard).

The procedure for looking up a host is:

```
task = gsk_name_resolve (GSK_NAME_RESOLVER_FAMILY_IPV4,
                         hostname,
                         handle_name_resolution_succeeded,
                         handle_name_resolution_failed,
                         func_data,
                         destroy_func_data);
/* if you don't need to cancel it,
   then you can safely call
   gsk_name_resolver_task_unref() now. */
```

Chapter 11

HTTP: A Very Important Protocol

This chapter describes the most protocol you will likely deal with more than any other: HTTP. It is literally the defining protocol of the web: it stands for Hyper-Text Transfer Protocol and was designed with linked documents in mind. Now, it has succeeded: practically all websites use HTTP. It is known to even non-programmers as the common prefix of URLs.

The HTTP protocol is essentially transactional: it is divided into client and server roles. The client issues a request that begins the transaction, and the server responds.

Most transactions begin with a new TCP connection, but the protocol is transport independent. Also, in HTTP 1.1 a mechanism called “keep-alive” was introduced that allows more than one transaction to be issued on a single transport stream. This can boost performance dramatically when there are many small files to be downloaded.

A huge fraction of the ordeal of implementing an HTTP client or server is parsing and outputting the HTTP headers that provide meta-data about the transaction. The request and response headers have somewhat different, but similar, header formats. Additionally the request and response may have a stream of data attached to it. The request and response indicate whether data is attached.

The request specifies a *verb* and *path*. The verb indicates what the client wants to do. The most common verbs are `GET` and `POST`. The difference is simple: a `GET` request has no data associated with the request, whereas the `POST` request does. In the protocol, there are no semantics associated with `GET` and `POST` requests: for example, it is unspecified whether a request is trying to modify the server state or merely query it. The path is a string that either is like a file-system path or a URL. The file-system path is standard for most servers, but proxies often take fulls URLs as the path.

The protocol contains a mechanism for describing the type of content that is associated with either a request or a response. It wisely uses the same header tags for both.

Dual to information that describes the request or response is information that describes the types of requests or responses that are allowed.

Chapter 12

HTTP Content: Easy Server Implementation

Well, that's great. If you read the last chapter, now you saw that implementing a general HTTP server from its basic pieces is a pain. There are a few reasons for that:

1. you typically want the HTTP server to be presented as a unified whole. But using the raw HTTP classes having objects on a connection-by-connection basis.

2. you typically want a file-system structure, with a sequence of handlers. You may want a few other filters: user-agent and virtual-host are the principal considerations here.

`GskHttpContent` is intended to address these issues. It addresses the first issue by being a unified object that handles all incoming streams. It addresses the second issue by providing a good framework for mapping the virtual file-system space.

Here are a list of `GskHttpContent`'s features: [XXX: elaborate: these are too tersely written]

1. path, user-agent, virtual-host filtering

2. cgi

3. fs mapping

4. mime-type mapping (based on filenames)

Chapter 13

Debugging GSK Programs

You cannot program unless you can debug. But of course, you do not need fancy tools: a single bit of output to distinguish cases is sufficient. However, more likely, you want to get things working fast, well and reliably. With all these objects and memory-allocation schemes, how do you check you program?

Nothing will compare with designing correctly to begin with. No debugging is that fastest debugging of all, and if you can make your problem (or type) sufficiently small that it's trivial, then *that's far better* than what any debugging tool will do for you.

But that's not the way things work. In practice your program will crash and it'll be your problem. This chapter is to distill some technical advice learned from debugging GSK on linux with x86. Hopefully some of it is portable to your situation.

13.1 General Ideas

Before getting into specific tools, let's describe a set of general procedures for debugging a program.

Of course, it's crazy to not use 'gdb' or 'valgrind' first, just because the problem may be trivial, like a NULL-pointer dereference.

What follows are suggestions for problems that don't necessarily crash, but give garbled data.

1. *Identify the first problem.* A rather common error made by novices generally is to assume that the last error message on their screen caused the problem. No. The first problem is the one to look at: after that, everything can be expected to violate your expectations.

2. *Make a test case.* Locate a function call whose outputs seem broken given its inputs. Try to reproduce the problem in a small test case: a standalone C program that does as little as possible: running and compiling quickly are important here.

3. *Binary search of a small test case.* In the small test case, binary-search the problem. Identify points halfway and print out all the inputs and outputs. Add any assertions you can think of at these points as well.

   Theoretically, these printouts will tell you if the problem is in the first half or the second half. Of course, it is quite typical during your analysis to find you were mistaken in your bug analysis. Question assumptions liberally: skepticism is definitely recommended, as is modestly and a willingness to revert even further up: debugging has led to many redesigns. But you are better off redesigning than "debugging" a broken design.

   But if the printouts tell you, then iterate: subdivide the code and repeat.

   It is actually not at all uncommon for a binary-search to yield a single broken statement: ie debugging printouts before and after the statement suggest almost that the language or compiler is at fault. That's possible! But probably not: here's a list of possible suggestions:

   (a) if using C or C++, check out the preprocessor output using the '-E' option to many compilers. Maybe a macro is foiling you.
   (b) you debugger variable watches to find when the crucial element is changing.
   (c) check for local-variable scoping.
   (d) check for namespacing and overloading problems. Make sure, using prints or debuggers that the functions being called are the right ones. (This problems doesn't happen in C, which lacks namespacing and overloading)
   (e) watch out for bugs like objects being copied or not copied incorrectly.
   (f) check out the assembly. in many environments, the assembly can be debugged and analysed itself.

4. *Handling Large but Repeatable Problems.* Sometimes a small test case simply cannot be made: for example, maybe there's no way to construct the state which leads to the failure condition.

   ...

5. *Handling Infrequent but Nagging Problems.* ..

6. *Handling Memory Leaks.* ...