Network Programming Internals of the Dillo Web Browser

Jorge Arellano-Cid Horst H. von Brand
Departamento de Informática
Universidad Técnica Federico Santa María
Casilla 110-V, Valparaíso, Chile
{jcid,vonbrand}@inf.utfsm.cl

Abstract

Network programs face several delay sources when sending or retrieving data. This is particularly problematic in programs which interact directly with the user, most notably web browsers. We present a hybrid approach using threads and signal driven I/O, which allows a non-blocking main thread and overlapping waiting times.

1. About dillo

The idea of a lightweight web browser is not new. The Dillo project didn't start from scratch but mainly working on the code base of `gzilla` (a light web browser written by Raph Levien [6]). As the project went by, the code of the whole source was standardized, and the networking engine was replaced with a new, faster design.

The source code is currently in alpha test, the latest release (0.2.3 as of this writing) is intended for developers only. Dillo is available at `<http://dillo.sourceforge.net>` under the GNU General Public License [3].

2. Delay sources in network programs

Network programs face several delay-sources while sending or retrieving data. In the particular case of a web browser, they are found in:

- DNS querying
- Initiating the TCP connection
- Query sending
- Retrieving data
- Closing the TCP connection

In a WAN context, every single item of this list has an associated delay that is non deterministic and often measured in seconds. If we add several connections per browsed page (each one requiring at least the 4 last steps), the total latency can be considerable.

The Dillo networking engine addresses this with a hybrid approach that uses pthreads for DNS querying, and signal driven I/O for handling TCP connections. This scheme allows for non-blocking control flow in the main thread (this makes the interactive "feel" of the program much better), and a way of overlapping waiting times.

3. The traditional (blocking) approach

The main problems with the blocking approach are:

- When issuing an operation that can't be completed immediately, the process is put to sleep waiting for completion, and the program doesn't do any other processing in the meantime.

- When waiting for a specific socket operation to complete, packets that belong to other connections may be arriving, and have to wait for service.

- Web browsers handle many small transactions, the waiting times are not overlapped and the latency as perceived by the user can be very annoying.

If the user interface is just put to sleep during network operations, the program becomes unresponsive, confusing and perhaps alarming the user. Not overlapping waiting times and processing makes graphical rendering (which is arguably the central function of a browser) unnecessarily slow.

4. Dillo's hybrid engine

Dillo uses threads and signal driven I/O extensively to overlap waiting times and computation. Handling the user interface in a thread that never blocks gives a good interactive "feel." The use of GTK+ [2], a sophisticated widget framework for graphical user interfaces, helped very much to accomplish this goal. All the interface, rendering and I/O engine was built upon its facilities.

The design is said to be "hybrid" because it uses threads for DNS querying and reading local files, and signal driven I/O for TCP connections. The threaded DNS scheme is potentially concurrent (this depends on underlying hardware), while the I/O handling (both local files and remote connections) is definitively parallel.

To simplify the structure of the browser, local files are encapsulated into HTTP streams and presented to the rest of the browser as such, in exactly the same way a remote connection is handled. To create this illusion, a thread is launched. This thread opens a pipe to the browser, it then synthesizes an appropriate HTTP header, sends it together with the file to the browser proper. In this way, all the browser sees is a handle, whether data comes from a remote connection or from a local file.

To handle a remote connection is more complex. In this case, the browser asks the cache manager for the URL. The name in the URL has to be resolved through the DNS engine, a socket TCP connection must be established, the HTTP request has to be sent, and finally the result retrieved. Each of the steps mentioned could give rise to errors, which have to be handled and somehow communicated to the rest of the program. For performance reasons, it is critical that responses are cached locally, so the remote connection doesn't directly hand over the data to the browser; the response is passed to the cache manager which then relays it to the rest of the browser. The DNS engine caches DNS responses, and either answers them from the cache or by querying the DNS. Querying is done in a separate thread, so that the rest of the browser isn't blocked by long waits here.

The activities mentioned do not happen strictly in the order stated above. It is even possible that several URLs are being handled at the same time, in order to overlap waiting and downloading. The functions called directly from the user interface have to return quickly to maintain interactive response. Sometimes they return connection handlers that haven't been completely set up yet. As stated, I/O is signal-driven, when one of the descriptors is ready for data transfer (reading or writing), it wakes up the I/O engine.

Data transfer between threads inside the browser is handled by pipes, shared memory is little used. This almost obviates the need for explicit synchronization, which is one of the main areas of complexity and bugs in concurrent programs. Dillo handles its threads in such a way that its developers can handle it as if it was a single thread of control. This is accomplished by making the DNS engine call-backs happen within the main thread, and by isolating file loading with pipes.

Using threads in this way has three big advantages:

- The browser doesn't block when one of its child threads blocks. In particular, the user interface is responsive even while resolving a name or downloading a file.
- Developers don't need to deal with complex concurrent concerns. Concurrency is hard to handle, and few developers are adept at this. This gives access a much larger pool of potential developers, something which can be critical in an open-source development project [8].
- By making the code mostly sequential, debugging the code with traditional tools like `gdb` [9] is possible. Debugging parallel programs is very hard, and appropriate tools are hard to come by.

Because of simplicity and portability concerns, DNS querying is done in a separate thread. The standard C library doesn't provide a function for making DNS queries that don't block. The alternative is to implement a new, custom DNS querying function that doesn't block. This is certainly a complex task, integrating this mechanism into the thread structure of the program is much simpler.

Using a thread and a pipe to read a local file adds a buffering step to the process (and a certain latency), but it has a couple of significative advantages:

- By handling local files in the same way as remote connections, a significant amount of code is reused.
- A preprocessing step of the file data can be added easily, if needed. In fact, the file is encapsulated into an HTTP data stream.

4.1. DNS queries

Dillo handles DNS queries with threads, letting a child thread wait until the DNS server answers the request. When the answer arrives, a call-back function is called, and the program resumes what it was doing at DNS-request time. The interesting thing is that the call-back happens in the main thread, while the child thread simply exits when done. This is implemented through a server-channel design.

4.2. The server channel

There is one thread for each channel, and each channel can have multiple clients. When the program requests an

IP address, the server first looks for a cached match; if it hits, the client call-back is invoked immediately, but if not, the client is put into a queue, a thread is spawned to query the DNS, and a GTK+ [2] idle client is set to poll the channel 5 times per second for completion, and when it finally succeeds, every client of that channel is serviced.

4.3. Handling TCP connections

Establishing a TCP connection requires the well known three-way handshake packet-sending sequence [10]. Depending on network traffic and several other issues, significant delay can occur at this phase. Dillo handles the connection by a non blocking socket scheme. Basically, a socket file descriptor of `AF_INET` type is requested and set to non-blocking I/O. When the DNS server has resolved the name, the socket connection process begins by calling `connect(2)`[1]; which returns immediately with an `EINPROGRESS` error. After the connection reaches the `EINPROGRESS` "state," the HTTP query is created and submitted to the lower level I/O engine. This engine sets up a call-back for when the file descriptor is ready for writing. Another call-back function is set to receive the answer, on the same file descriptor, using the low-level I/O engine. The advantage of this scheme is that both sending and retrieving are completely set up before the socket even finishes the connection process [7]. The socket will generate a signal when I/O is possible.

4.4. Handling queries

In the case of a HTTP URL, queries typically translate into a short transmission (the HTTP query) and a lengthy retrieval process. Queries are not always short though, specially when requesting forms (all the form data is attached within the query), and also when requesting CGI programs.

Regardless of query length, query sending is handled in background. The thread that was initiated at TCP connecting time has all the transmission framework already set up; at this point, packet sending is just a matter of waiting for the `GDK_INPUT_WRITE` signal to come and then sending the data. When the socket gets ready for transmission, the data is sent using `writev(3)`. If the transmission succeeds, the call-back function (previously set by the client) is called to acknowledge success of the operation.

4.5. Receiving data

Although conceptually similar to sending queries, retrieving data is very different as the data received can easily exceed the size of the query by many orders of magnitude (for example when downloading images or files). This is one of the main sources of latency, the retrieval can take several seconds or even minutes when downloading large files.

The data retrieving process for a single file, that began by setting up the expecting framework at TCP connecting time, simply waits for the `GDK_INPUT_READ` signal. When it happens, the low-level I/O engine gets called, the data is read into pre-allocated buffers and the appropriate call-backs are performed. Technically, whenever a `GDK_INPUT_READ` signal is generated, data is received from the socket file descriptor, using the `readv(3)` function. This iterative process finishes when `readv(3)` returns EOF.

4.6. Closing the connection

Closing a TCP connection requires four data segments, not an impressive amount but twice the round trip time, which can be substantial. When data retrieval finishes, socket closing is triggered. There's nothing but a shutdown call on both reading and writing for the socket. This process was originally designed to split the four segment close into two partial closes, one when query sending is done and the other when all data is in. This scheme is not currently used due to apparent bugs in the socket implementation of Linux.

4.7. The low-level I/O engine

Dillo I/O is carried out in the background. This is achieved by using low level file descriptors and signals. Anytime a file descriptor shows activity, a signal is raised and the signal handler takes care of the I/O.

The low-level I/O engine (*I/O engine* from here on) was designed as an internal abstraction layer for background file descriptor activity. It is intended to be used by the cache module only; higher level routines should ask the cache for its URLs. Every operation that is meant to be carried out in background should be handled by the I/O engine. In the case of TCP sockets, they are created and submitted to the I/O engine for any further processing.

The submitting process (client) must fill a request structure and let the I/O engine handle the file descriptor activity, until it receives a call-back for finally processing the data. This is better understood by examining the request structure (see figure 1).

To request an I/O operation, this structure must be filled and passed to the I/O engine using the `a_IO_submit()` function.

- `Op` and `IOVec` must be provided.
- `Callback` must be provided for `IORead`, and should be provided for `IOWrite`.

[1]We use the UNIX convention of identifying the manual section where the concept is described, in this case section 2 (system calls).

```
typedef void
  (*IOCallback_t)(int Op, void *CbData);

typedef struct {
   int Op;
       /* IORead | IOWrite | IOWrites */
   int GdkTag;
       /* gdk_input tag
          (used to remove) */
   ssize_t Status;
       /* Number of bytes read,
          or -errno code */
   struct iovec IOVec;
       /* Buffer place and length */
   IOCallback_t Callback;
       /* Callback when Op is done */
   void *CbData;
       /* Callback function data */
   int FD;
       /* Current File Descriptor */
} IOData_t;
```

Figure 1. The request structure

- `CbData` should be provided.
- `GdkTag`, `Status`, and `FD` are set by I/O engine internal routines.

When there is new data in the file descriptor, `IO_callback` gets called (by GDK). Only after the I/O engine finishes processing the data the user-provided `Callback` function is called.

Note that the client call-back definition allows an operation parameter to be passed back (`Op`). This parameter is used to tell the client what action it should take to process the data, as well as when to close or abort an operation.

4.8. The I/O engine transfer buffer

The `IOVec` field of the request structure provides the transfer buffer for each operation. This buffer must be set by the client (to increase performance by avoiding copying data).

On reads, the client specifies the amount and where to place the retrieved data; on writes, it specifies the amount and source of the data segment that is to be sent. Although this scheme increases complexity, it has proven very fast and powerful. For instance, when the size of a document is known in advance, a buffer for all the data can be allocated at once, eliminating the need for multiple memory reallocations. Even more, if the size is known and the data transfer is taking the form of multiple small chunks of data, the client only needs to update the `IOVec` to point to the next byte in its large preallocated reception buffer (simply by adding the chunk size). On the other hand, if the size of the transfer isn't known in advance, the reception buffer can remain untouched until the connection closes, but the client must then accomplish the usual buffer copying and reallocation.

The I/O engine also lets the client specify a full length transfer buffer when sending data. It doesn't matter (from the client's point of view) if the data fits in a single packet or not, it's the I/O engine's job to divide it into smaller chunks if needed and to perform the operation accordingly.

4.9. Handling multiple simultaneous connections

The previous sections describe the internal work for a single connection, the I/O engine handles several of them in parallel. This is the normal downloading behavior of a web page. Normally, after retrieving the main document (HTML code), several references to other files (typically images) and sometimes even to other sites (mostly advertising today) are found inside the page. In order to parse and complete the page rendering, those other documents must be fetched and displayed, so it is not uncommon to have multiple downloading connections (every one requiring the whole fetching process) happening at the same time.

Even though socket activity can reach a hectic pace, the browser never blocks. Note also that the I/O engine is the one that directs the execution flow of the program by triggering a call-back chain whenever a file descriptor operation succeeds or fails.

5. Conclusions

Dillo is currently in alpha test. It already shows impressive performance, and its interactive "feel" is much better than that of other web browsers. One of the effects of its design is that the user never has to wait to continue interacting with the browser.

Performance of the various subsystems of competing programs is almost impossible to measure, as doing so implies detailed instrumenting of their code. Separating times for rendering, querying, downloading, and assorted network delays is a titanic task. In any case, the exact values of each of these measures in isolation are only remotely related to the end user's satisfaction.

The modular structure of Dillo, and its reliance on GTK+ [2] allow it to be very small (only 200Kb compressed source code in C [5], some 840Kb uncompressed, the dynamically linked executable for i386 on Linux is 140Kb). Not every feature of HTML-4.0 [4] has been implemented yet, but no significant problems are foreseen in doing this.

The fact that Dillo's central I/O engine is written using advanced features of POSIX [11] and TCP/IP networking [1, 12] makes its performance possible, but on the other hand this also means that only a fraction of the interested hackers are able to work on it.

Up to recently there has been little documentation of the browser's internals; now there is much effort on correcting this, particularly as the publication of the project has attracted new developers that have to be brought up to speed (the current paper is one side-effect of this effort).

A simple code base is critical when trying to attract hackers to work on a project like this one. Using the GTK+ framework helped both in creating the graphical user interface and in handling the concurrency inside the browser. By having threads communicate through pipes the need for explicit synchronization is almost completely eliminated, and with it most of the complexity of concurrent programming disappears. A clean, strictly applied layering approach based on clear abstractions is vital in each programming project. A good, supportive framework is of much help here.

References

[1] D. E. Comer and D. L. Stevens. *Internetworking with TCP/IP, Volume Three: Client-Server Programming and Applications (BSD Socket Version)*. Prentice Hall, second edition, 1996.

[2] T. Gale and I. Main. The GTK+ tutorial. `http://www.gtk.org/tutorial/`, 2000.

[3] GNU General Public Licence. Free Software Foundation, June 1991. (Version 2).

[4] HTML 4.0.1 specification: W3c recommendation 24 december 1999. `http://www.w3.org/TR/1999/REC-html401-19991224`.

[5] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, second edition, 1988.

[6] R. Levien. Gzilla web-browser. `http://www.levien.com/free.html`.

[7] D. J. Lilja. Exploiting parallelism available in loops. *IEEE Computer*, 27(2):13–26, February 1994.

[8] E. S. Raymond. *The Cathedral and the Baazar*. O'Reilly, 1999.

[9] R. Stallman, R. Pesch, S. Shebs, et al. *Debugging with gdb: The GNU Source Level Debugger*. Free Software Foundation, 8 edition, 2000.

[10] R. W. Stevens. *TCP/IP Illustrated, Volume One: The Protocols*. Addison-Wesley, 1993.

[11] W. R. Stevens. *Advanced Programming in the Unix Environment*. Addison-Wesley, 1993.

[12] W. R. Stevens. *Unix Network Programming: Volume 1: Networking APIs: Sockets and XTI*. Prentice-Hall, second edition, 1998.