Communicating Process Architectures – 2003
Jan F. Broenink (Eds.)
IOS Press, 2003

occwserv: An occam Web-Server

Fred BARNES
Computing Laboratory, University of Kent, Canterbury, Kent. CT2 7NF
frmb2@kent.ac.uk

Abstract. This paper presents 'occwserv', the occam web-server. This is a highly concurrent web-server, written in the occam multi-processing language, that supports the majority of the HTTP/1.1 protocol. Dynamic process and channel creation mechanisms are used to create scalable 'server-farms', each responsible for a particular web-server function — for example, reading client requests or running CGI processes. The design of the web-server is presented, along with some early performance benchmark results. Although performance may appear a limiting factor (when compared to other web-servers such as Apache), much is gained from the simplicity and security of occam. Extending the web-server with new functionality, for example, is intuitive and largely trivial — with the guarantees that code is free from race-hazard and aliasing errors. An experimental non-standard addition, the OGI (occam Gateway Interface), is also presented. This provides a mechanism for dynamically loading and attaching pre-compiled occam processes to the running web-server, that can then handle one or multiple client connections. A text-based style adventure game is examined briefly, that allows multiple clients to interact within a "multi-user dungeon" (MUD) style environment.

1 Introduction

Web-servers are naturally concurrent applications — they must be capable of processing more than one client connection/request simultaneously, if to be of any practical use. Theoretically, from the web-server viewpoint, serial processing of clients would provide the greatest degree of efficiency. However, this requires that clients (and the network) are capable of absorbing data at the maximum rate generated by the web-server — rarely the case in practice. A more typical situation is one in which the web-server writes data simultaneously to multiple 'slow' clients, and this model reflects the typical 'real world' situation, where the bandwidth available to end-users (using web-clients) is generally significantly less than the bandwidth available to web-servers (often housed in dedicated 'server warehouses').

Thus, concurrency is a requirement, but may be provided for in several different ways. For web-servers written in C, such as Apache [1], handling of multiple clients is performed using a mixture of *threads* and the '`select()`' system-call (that performs I/O multiplexing). The cost of '`select()`' is generally $O(n)$, where n is the number of descriptors involved. At some point, this overhead becomes significant, hence the use of multiple threads, each performing '`select()`' on a manageable number of descriptors. An extensive overview of the 'event-dispatch' mechanisms of Linux (as of 2001) can be found in [2].

However, programming concurrency this way is neither intuitive nor easy, and has a huge potential for error. The occam multi-processing language [3], based on the CSP process algebra [4, 5], provides the ideal environment for developing concurrent applications — particularly in that it safeguards against common aliasing and race-hazard errors.

The KRoC/Linux [6] system, on which this work is based, was initially unable to provide general socket I/O — due to a lack of support for handling multiple blocking system-calls (because KRoC runs as a single UNIX process, any blocking call performed by an occam process would suspend the entire program). The addition of blocking system-call support [7] has removed this limitation and provides a general mechanism for executing C code in another thread, thereby allowing the scheduling of occam processes to continue normally.

An initial version of the occam web-server was presented during a fringe session at the CPA-2001 conference in Bristol [8]. This version of the web-server was entirely functional, but somewhat limited. In particular, it lacked the ability to scale dynamically, using fixed-size process farms. Additionally, the sharing of channels was handled using a user-defined '`SEMAPHORE`' type [9] combined with explicit suppression of aliasing and race-hazard checks for the shared channel.

This paper presents version 2 of the occam web-server, that makes extensive use of new language features that provide the missing functionality. Particular use is made of mobile *channel-types* and dynamic process creation using the '`FORK`' mechanism [10, 11]. The new version is a significant step forward from the original, now providing a highly concurrent *dynamic* web-server, that scales automatically with demand (and whose implementation remains secure against aliasing and race-hazard errors).

The web-server supports the HTTP/1.0 and HTTP/1.1 protocols [12, 13], but lacks support for HTTP over SSL — due largely to the lack of an (Open)-SSL [14] (secure sockets layer) binding for occam.

1.1 Related Work

The quality of a web-server is typically measured in terms of its performance at processing client requests. This represents a very real-world requirement — web-servers are expected to be able to handle the demand placed on them. The effect of this is that producing web-servers that are able to compete in the same league as Apache (the web-server which most will be benchmarked against), is *hard*. In particular, implementations in interpreted languages suffer enormously when placed under high-load.

Two web-servers that are of interest are the Haskell Web-Server [15], and YAWS [16], a concurrent web-server written in Erlang. Both of these present novel approaches to web-server implementation, in the typically functional languages Haskell and Erlang. A significant gain comes from the functional abstraction of these languages — code is automatically safe from race-hazard and aliasing errors, and the necessary concurrency mechanisms are provided by the run-time systems [17, 18].

However, for both these web-servers, there is quite some distance between design and implementation — particularly in concurrent Haskell. This is largely due to the fact that these languages were originally designed to be functional, not concurrent languages. occam, on the other hand, was designed to be concurrent (using the CSP model), and thus makes a significantly more natural language for expressing concurrent systems such as web-servers (provided they can be modelled in CSP).

From another viewpoint, functional languages are generally highly dynamic, utilising mechanisms such as lazy evaluation and garbage-collection [19]. A comparable level of dynamic behaviour has only recently been added to occam, largely in the form of dynamic process and network creation/re-configuration. Digressing slightly, this leads to an interesting question: is a dynamic occam useful for implementing run-time systems for functional languages ? — with extensive use being made of dynamic processes and mobile channel-ends.

Another type of web-server, that is becoming increasingly common (largely through its inclusion in Linux), is the TUX Linux-kernel embedded web-server [20, 21]. This uses optimised (and direct) interaction with the Linux file-system and networking code, resulting in significant performance gains. An analysis of TUX (1.0) is given in [22].

1.2 Paper Overview

Section 2 describes the design (and parts of the implementation) of ‘occwserv’, with each core component of the web-server examined separately. The performance of the web-server is examined in section 3. It should be noted, however, that performance is not the only objective of the occam web-server — clarity of design and simplicity of implementation are of equal importance, since these affect the long-term maintainability of the software (scalability in particular).

Section 4 presents some initial conclusions and discusses future avenues of research. Of particular interest is improving the performance of the web-server, that is bottle-necked in several critical places.

2 Design and Implementation

Figure 1 shows the top-level design of the occam web-server — a network of concurrent components that communicate client ‘connections’. Also shown are the socket interactions performed on connected clients.

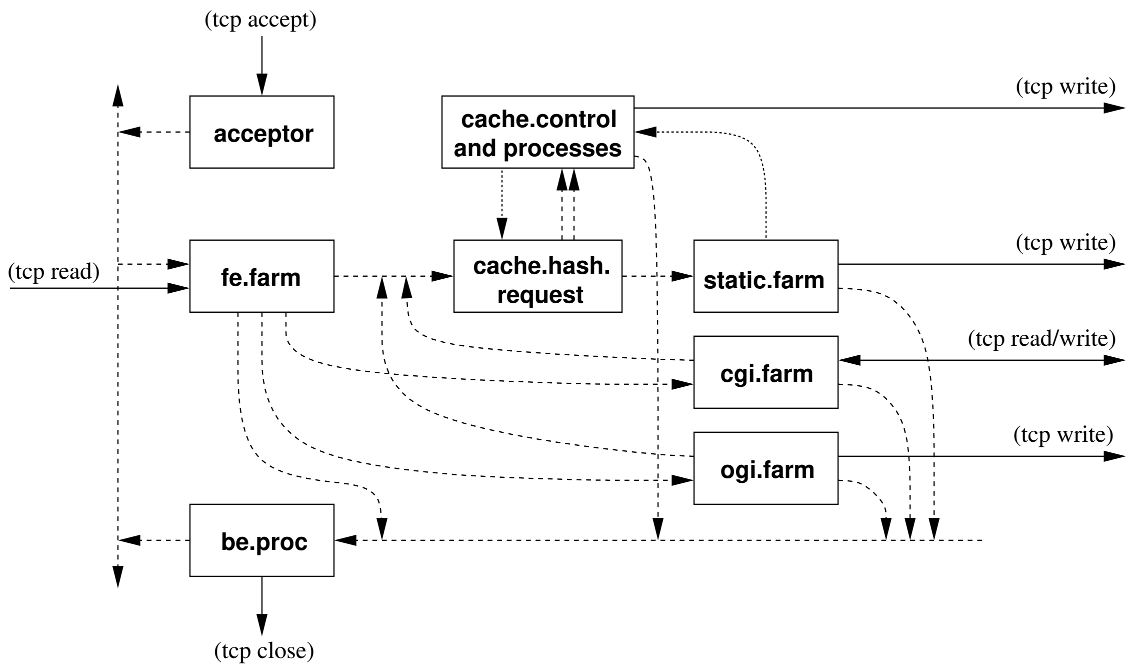

Figure 1: occam web-server main top-level processes

New connections originate in the ‘`acceptor`’ process and are communicated to the front-end process farm, ‘`fe.farm`’. The function of the front-end is to simply read the client request, then pass the connection on to another part of the web-server based on the request.

Requests for static content are passed through the ‘`cache.hash.request`’ process (that may divert them to a cache-handling process), into the ‘`static.farm`’. Processes within the ‘static’ process farm simply copy the contents of a file to the client, before passing the connection to the back-end (‘`be.proc`’).

The back-end process either closes the connection or, if marked as ‘keep-alive’, sends the connection back into the front-end farm for another request. Statistics are also collected at this point, and communicated to a separate process (not shown).

Requests for CGI scripts (the execution of a process whose output is returned to the client), are sent to the '`cgi.farm`' process. The operation of this is similar to the 'static' farm, where a separate 'worker' process exists for each active client connection.

The '`ogi.farm`' process network is used to handle requests for OGI ('occam gateway interface') modules — deliberately similar in name to 'CGI' (common gateway interface). OGIs are an experimental addition that enable pre-compiled occam processes to be dynamically loaded and attached to the running web-server. Furthermore, an OGI module can specify that *all* requests for a particular URL should be directed to that instance of itself. This has several significant advantages over the CGI and similar mechanisms, discussed in section 2.4.

2.1 Front-End Farm

The front-end farm is responsible for reading client requests, that determines where the connection will be sent next. The process 'pool' is managed by the '`fe.farm`' process, whose purpose is to ensure a readily available supply of '`fe.process`' worker processes, as shown in figure 2. Note that the incoming connection channel (from the '`acceptor`') is given to the worker processes directly — i.e. connections are not farmed out by '`fe.farm`', it simply ensures that enough free worker processes are available.

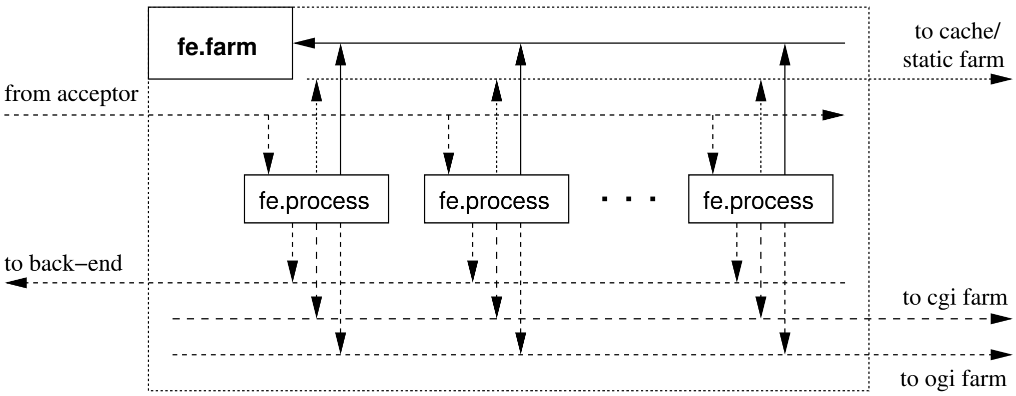

Figure 2: Front-end process farm

The '`fe.farm`' process maintains a count of the number of idle worker processes, initially zero. Following an idea suggested by Welch and presented in [23], the body of '`fe.farm`' is simply a loop — that first makes sure that there are n idle workers (`FORK`ing more if necessary), then performs an input from the shared channel connecting it to the workers. When a worker process starts processing a request, it communicates the value -1 on the shared channel. When it finishes processing a request, $+1$ is communicated. The value input inside '`fe.farm`' is added to the idle-worker count. Thus, whenever a worker process starts, this count is decremented, and incremented again when the worker finishes, with '`fe.farm`' constantly ensuring the availability of n workers.

Internally, the '`fe.process`' worker process invokes '`line.reader`' — a `PROC` whose purpose is to read lines of data (the request) from the client socket. Parts of the request that are of interest (specifically the 'action') are stored within the mobile connection structure, passed as a parameter.

Once the request has been read, the '`fe.process`' passes the connection to another part of the web-server, based on the file (name and path) requested. Requests that contain the sub-string '`/../`' are dropped, as are those that time-out (if read timeouts are enabled).

Requests whose path starts with the configured ‘CGI’ path (typically ‘`/cgi-bin/`’) are passed to the ‘`cgi.farm`’. Similarly, requests that start with the configured ‘OGI’ path (currently just ‘`/ogi/`’) are passed to the ‘`ogi.farm`’ process for handling. All other requests are passed towards the ‘`static.farm`’, via ‘`cache.hash.request`’.

2.2 Serving and Caching Static Pages

The original version of the occam web-server had no support for page caching. For frequently accessed files, caching can be an advantage, up to the point where the frequently requested set of files exceeds the cache size (where cache effectiveness starts to degrade, and may even become an overhead — if pages are repeatedly cached and un-cached).

Before sending a connection to the ‘`cache.hash.request`’ process, the front-end farm generates a hash-code for the request URI, that it stores inside the connection structure. The ‘`cache.hash.request`’ process maintains an array of hash-codes for currently cached pages, and is connected to a fixed number of ‘`cache.process`’s. When a connection is received, the hash-code array is scanned to find a match. If found, the connection is sent to the appropriate ‘`cache.process`’, otherwise the connection is forwarded to the ‘`static.farm`’ for ordinary processing.

The ‘`cache.process`’ processes and ‘`cache.hash.request`’ process are managed by the ‘`cache.control`’ process, as shown in figure 3. The ‘`cache.control`’ process controls what particular pages get cached, based on information sent from the ‘`static.farm`’ (requests for pages that were not cached) and the cache processes themselves (requests for pages that were cached).

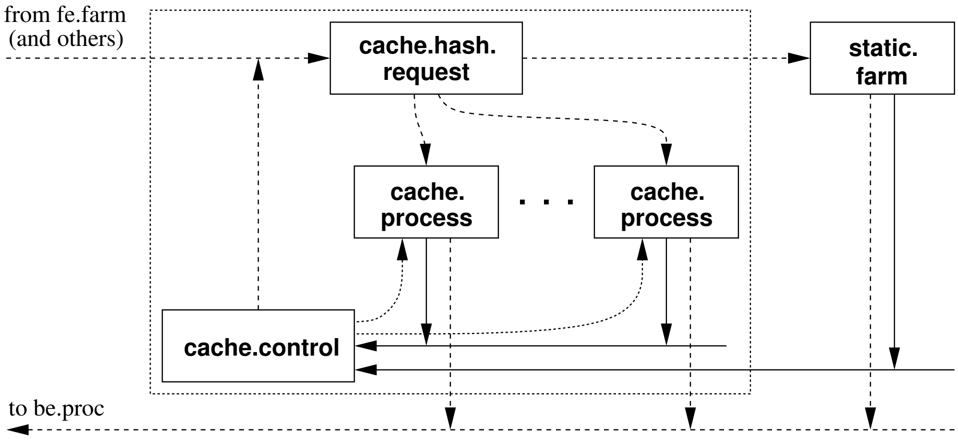

Figure 3: Cache control and processes

Internally, ‘`cache.control`’ maintains an array of cache-entry information, twice as big as the number of actual ‘`cache.process`’ processes. The entries are sorted (via a separate ‘lookup’ array) based on a ‘count’. This value is incremented by page hits and reduced over time, but is always kept between 1 and the number of cache processes — that prevents eviction through aging alone. The top-half of the sorted array maintained by ‘`cache.control`’ contains pages that are actually cached. The bottom-half of the array contains pages that are not currently cached, and acts as a ‘buffer’ area where pages compete before being loaded into the cache proper. Whenever a page joins or leaves the ‘top-half’ (cached pages), appropriate messages are sent to the cache processes and ‘`cache.hash.request`’ process. The implementation of this is done particularly carefully, in order to avoid deadlock and to prevent connections being sent to cache processes that are unable to handle them. Some use of the extended-rendezvous [10] is made by various components to achieve this.

Pages that are not cached are simply passed to the '`static.farm`'. This is a process farm that operates in very much the same way as the '`fe.farm`', but with incoming connections being shared amongst '`get.page`' processes. The job of '`get.page`' is to simply open the requested file and send the contents back to the client (prefixed with standard headers), before closing the file and passing the connection to the back-end ('`be.proc`'). The Linux-specific [1] '`sendfile()`' system-call is used to transfer file-contents to the client, that performs the transfer in the OS (Linux) kernel to avoid the unnecessary copying of data to and from user-space.

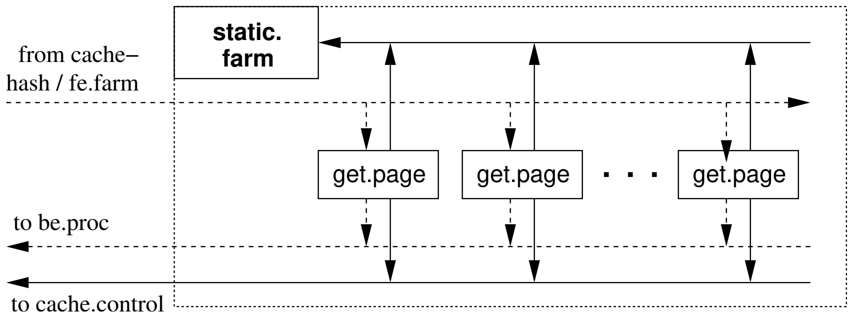

Figure 4: Static-page process farm

The '`get.page`' processes are also connected back to the '`cache.control`' processes, to which they send information about missed cache requests, as shown in figure 4. If the requested file cannot be opened (in '`get.page`'), the request is re-written to '`/error....html`', with the dots replaced by the appropriate error-code (404 for 'file not found', for example).

2.3 Running CGI Scripts

CGI (Common Gateway Interface) [24] provides a mechanism for running programs, scripts in particular, from within a web-server whose output is returned to the (remote) client. This can be used from anything such as interfacing legacy applications via shell-scripts or C programs, to generating dynamic content from databases using PHP [25]. Interaction is still strictly request-response, but the response may contain substantial amounts of embedded code (such as Java-Script [26]).

The '`cgi.farm`' process-farm is constructed in the same way as the '`fe.farm`' and '`static.farm`' networks, using '`cgi.page`' processes to handle individual requests. The occam 'process' library [27] is used to execute CGI processes, that have their 'standard-output' connected directly to the client socket.

If the requested program/script cannot be run (because it does not exist or cannot be executed), the connection is passed to the static handlers (via the cache), with the request modified to return an error-file.

A reasonable amount of effort is involved in setting up environment for a CGI script (as per the CGI specification), but is relatively straight-forward. The occam process library handles the execution and termination of other programs transparently. All the application ('occwserv' in this case) need do is call '`proc.run`', giving the path to the executable, an environment array and a set of file-descriptors for communication (one for each of the streams '`stdin`', '`stdout`' and '`stderr`'). A more occam-friendly interface is available through '`proc.wrapper`', that uses `BYTE` channels for communication instead of UNIX file-descriptors.

[1] The '`sendfile`' system-call, that copies data between file-descriptors within the kernel, exists on various UNIX/POSIX systems, but often with different semantics and restrictions.

2.4 OGI 'plug-in' Modules

Although CGI provides a versatile mechanism for interfacing with arbitrary programs via a HTTP interface, it does have limitations. The most restrictive of these is performance — creating and executing a new OS process is expensive. Since this overhead is generally constant, the impact on small (quick running) CGI scripts is large, particularly noticeable when benchmarking (section 3). Long-running CGI processes, for example, performing complex remote database queries that may take several seconds to execute, tolerate the process startup/shutdown overhead more readily. For such cases, performance in benchmarking should depend on the efficiency of the remote server(s) — if the web-server is simply acting as a 'gateway'.

For Apache, some of the CGI performance issues are solved by building script interpreters into Apache run-time loadable modules. Two examples are the Perl and PHP modules, that interpret those scripting languages from within the web-server, without requiring a separate OS process. This results in a significant performance improvement when 'executing' CGI scripts of these types. There is, however, no protection between Apache and its modules — an erroneous module could easily take down the web-server (or more seriously, compromise the security of the system).

The other significant limitation of CGI scripts is the handling of persistent (client) state. Many applications interfaced through the CGI mechanism require client state to be maintained in some way. For an on-line ordering system, this might be the contents of the remote client's 'shopping-basket', or the current 'state' in an on-line banking transaction. Current implementations range from storing the whole state in the client (by ping-pong-ing the state with each request-response), to maintaining the state in the server and providing clients with a *session* identifier, that is used to access the server-stored state for subsequent requests. This state can either be stored on disk between requests, or use can be made of a dedicated back-end server that retains the state in memory, with CGI scripts that simply communicate data to and from this server (assuming it generates HTML or other suitable output).

Although these approaches to managing client state do work, they have their own limitations. In a transaction based system, for example, a 'false' client could send a number of requests concurrently, that each modify the server state in a different way. The outcome in these cases should ideally be predictable — an error message, for instance, or the successful completion of one and appropriate 'state consistency' errors generated for the others. In general, interaction with transaction based systems are (per-client) sequential, and this must be enforced somehow — by locking the server-held client 'state' during the transaction, for example.

Ultimately, CGI can be complex to program, particularly if separate requests must 'interact' with each other (in a real-time on-line transaction, for example). The general non-concurrent approach to CGI programming introduces its own problems, limited scalability in particular.

2.4.1 Introduction

For the occam web-server, the OGI (occam Gateway Interface) has been introduced. Instead of executing 'common' code behind the web-server, pre-compiled occam code is dynamically loaded into the web-server — as a concurrent process network in its own right. In a similar way to Apache modules, 'occwserv's OGI modules become part of the web-server handling connections internally. Furthermore, an OGI can specify (when loading) that it handles *all* requests for that OGI — i.e. the OGI can, if it wishes, handle multiple concurrent connections simultaneously — or it can serialise them (simply by not accepting a new connection into the OGI until an earlier one has left). Within a concurrent system, however, concurrent client interactions are much easier to manage. Furthermore, the OGI can remain loaded (when it specifies that *all* requests should be directed to it). This almost instantly solves much of the state-retaining problem, the OGI simply stores the client state in variables/processes and clients use a form of '`session`' identifier to access that state (the occam web-server currently does not support 'cookies' — state that is stored in the client but transfered in HTTP headers rather than HTML code).

2.4.2 OGI Process Handling

Figure 5 shows the structure of the '`ogi.farm`' process network in the web-server. Currently, a fixed-size pool of '`ogi.handler`' processes is used, but this could be made dynamic in the future (the required support for some nested `MOBILE`s has only recently been added to occam).

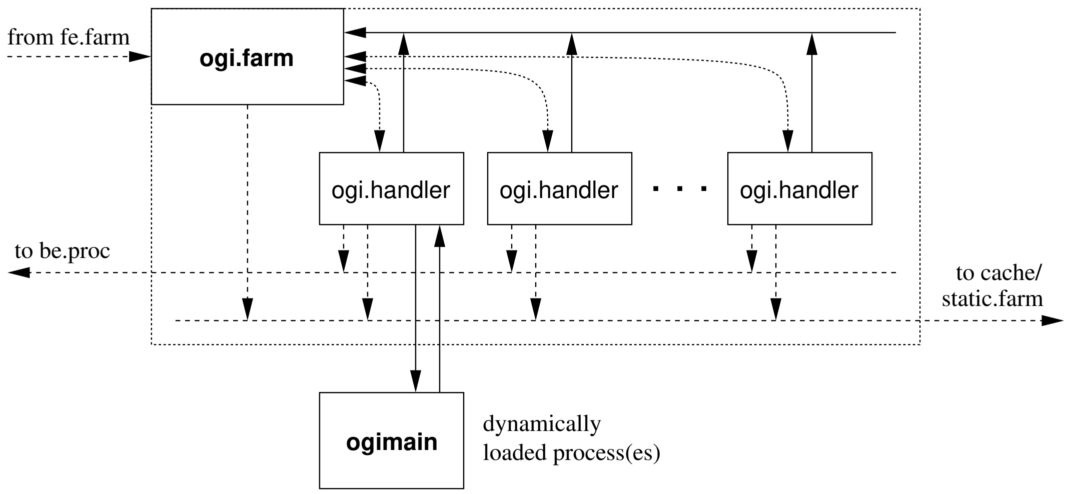

Figure 5: OGI process network

Unlike other process farms, connections are input by the '`ogi.farm`' process, then passed to the worker processes. The main body of the '`ogi.farm`' process, after creating the worker processes, forever `ALT`s between incoming connections (from the front-end farm) and 'signals' from the worker processes (using a shared local channel).

The '`ogi.handler`' worker processes exist in one of three states, that is maintained in the '`ogi.farm`' process: idle; busy and not accepting connections; or busy *and* accepting connections for the same OGI (matched using the connection's hash-code).

When a new connection is received by the '`ogi.farm`' process, the internal state is scanned to see if any OGI handler is currently accepting all requests for that OGI. If so, the connection is simply forwarded to that worker process. Otherwise, the internal state is re-scanned to find an idle worker. If none can be found, the connection is passed to the static farm (via the cache) to return an error to the client[2].

If a free worker is found, the connection is forwarded to that worker, but instead of looping back to the `ALT`, the '`ogi.farm`' process waits for a response from the worker to which the connection was sent. The response is only sent by a worker after it has started the actual OGI process, and indicates whether the OGI will handle the single connection given to it *only*, or will handle *all* requests for that OGI. This information is used to update the internal state of '`ogi.farm`', that then loops back to the `ALT`. If the OGI failed to load (signalled

[2]With an unbounded dynamic OGI process farm, the error of 'no free slots' will not occur — until memory or other system resources are exhausted.

in this response), the connection is passed to the static farm (to return an error) and the 'ogi.handler' process returns to its idle state.

The shared 'signal' channel is used by the 'ogi.handler' processes to notify 'ogi.farm' when they have finished/terminated, sending their 'ID' (allocated from zero upwards). When communicated, the 'ogi.farm' process simply changes its internal state for that worker to 'idle'. Before it returns to the idle state internally, the 'ogi.handler' process communicates the connection to the back-end of the web-server (section 2.5).

2.4.3 OGI Modules

OGI modules are pre-compiled occam processes (or networks of processes), accessed using a PROC named "ogimain", with the following signature:

```
PROC ogimain (VAL DPROCESS me, []CHAN ANY x.in?, x.out!)
```

This is the generic interface used by the "dynamic loadable processes" extension to KRoC/-Linux [28]. The actual interface used by the web-server for OGIs is recovered using channel RETYPEs:

```
CHAN OGI.LINK.IN in? RETYPES x.in[0]?:
CHAN OGI.LINK.OUT out! RETYPES x.out[0]!:
CHAN SHARED LOG! log.in? RETYPES x.in[1]?:
```

Where the 'OGI.LINK.IN' and 'OGI.LINK.OUT' protocols are defined by:

```
PROTOCOL OGI.LINK.IN IS D.CONN; MOBILE []BYTE:

PROTOCOL OGI.LINK.OUT
  CASE
    persist; BOOL
    conn; D.CONN
:
```

The 'D.CONN' structure is the *mobile* 'connection' record, passed into the OGI module through 'OGI.LINK.IN' and retrieved from the 'conn' variant of the 'OGI.LINK.OUT' protocol. The following 'MOBILE []BYTE' (dynamic BYTE array) on 'OGI.LINK.IN' contains the 'query-string' from the request, with appropriate de-mangling of encoded characters, "%20" meaning 'space' for example[3]. The third channel, 'log.in', is used to communicate a shared (client) channel-end to the OGI, that is remotely connected to the system 'log' (described in section 2.6).

When an OGI starts up (run in parallel within an 'ogi.handler' process), it inputs its first connection, then responds with a 'persist' message. Ideally, this should be done as quickly as possible, since the 'ogi.farm' will not accept any more new connections until it has received this response. If 'TRUE', the 'ogi.farm' will re-direct all requests for that OGI into the newly activated handing process. If 'FALSE', other requests for the same OGI will result in new handlers being activated.

The code within 'ogi.handler' communicates on all three channels in parallel, so an OGI module can send the 'persist' message first (if no interaction with the remote client is required first). As soon as this is received, the response is relayed to the 'ogi.farm' process so it can continue accepting connections.

[3]The first 128 characters are typically ASCII, the rest depend on the character-set used by the server (and are interpreted appropriately) — generally one of the extended ASCII character sets.

As far as error-handling is concerned, the dynamic processes extension ‘catches’ errors from loaded process networks. This stops an erroneous module from bringing down the web-server, but at the expense of possibly ‘losing’ system resources (for the web-server, dynamic memory and file-descriptors). The connection itself can be recovered (by copying, with `CLONE`, the connection structure before it is passed to ‘`ogimain`’). This is indicated to the ‘`ogi.handler`’ locally, when the call that ‘runs’ the dynamic process returns:

```
ccsp.run ("ogimain", libhandle, [xx.in,xx.log], [xx.out], res)
```

The ‘`libhandle`’ is a reference to the dynamically loaded library that contains an ‘`ogimain`’ process. ‘`xx.in`’ and ‘`xx.out`’ are the reverse `RETYPE`s for the channel connections. The ‘`xx.log`’ channel is used to communicate the *shared* mobile channel-end (connected to the system ‘log’) to the OGI process. The result is left in the `INT` parameter ‘`res`’. On successful termination, this will hold the constant ‘`DPROCESS.FINISHED`’, or ‘`DPROCESS.FAULTED`’ if a run-time error occurred[4].

Five OGI modules have currently been implemented — mostly for demonstration purposes. Two of the modules are simple tests, one persistent (with a ‘hit-counter’) and one non-persistent (handling only one connection then terminating). The traditional ‘finger’ script has also been OGI’d. This uses the occam socket library to open a connection to the requested machine (given as the query-string), and then returns the information sent back — or an error message if the connection times out (after 5 seconds) or is refused.

Another OGI module is one simply designed to crash in various ways — to test the dynamic process crash-handling mechanism as well as the web-server. Supported errors range from simple `STOP`s to array-bounds and floating-point errors. The final OGI is the “occam adventure game”, described in the following section.

2.4.4 The occam Adventure Game

The occam adventure game is a basic “multi-user dungeon” (MUD), that allows (multiple) users to login and interact via the web. Rather than being a pure client to web-connections, the adventure game is an autonomous process network, with built-in ‘bots’ that randomly wander around the game.

The current design is fairly static, using a centralised ‘game’ process that manages locations, objects and users. Processes representing objects and users (either internal ‘bots’ or remote web-clients) are connected to the game process using a shared channel-type. The client used for interacting with such environments is typically ‘telnet’, or a similar text-based TCP/IP clients. The client/server interaction is largely asynchronous — the server may send data to the client whenever it wishes; the client to server data is provided by the user.

This presents a slight problem when interfacing with remote web-clients, whose interaction with the web-server is strictly request-response. The solution is to introduce of buffering and polling — the returned web-page instructs the client to auto-refresh after 30 seconds, that will then collect up any game ‘events’ received since the last request-response.

In addition to the web-interface, the game also offers an interface through the IRC (Internet Relay Chat) protocol [29, 30]. With this interface, the game connects to an IRC server and registers itself as a user (currently ‘occgam’). Users (on the same IRC *network*) interact with the game using IRC ‘private messages’ (to and from ‘occgam’). IRC supports the asynchronous behaviour of the game more naturally than HTTP, with ‘events’ sent directly to remote clients (instead of being queued, as they are for web-clients).

[4]There is a third possibility for dynamically loaded processes, ‘`DPROCESS.SUSPENDED`’. This is not currently supported by the web-server, however.

Figure 6 shows the game process network. Web-clients are handled within the '`ogimain`' process only. The main '`game`' process maintains internal arrays of users 'logged in' to the game, that are either web-clients or IRC clients. Each interacting IRC client is handled by an individual '`irc.client`' process, controlled through the '`irc.interface`' process. The '`game`' process generates a potentially infinite stream of 'events' (for example, "person X entered room Y"). The '`event.filter`' process receives these events and selects only those which are relevant to connected IRC clients. These events are passed through an overwriting-buffer (within the '`irc.interface`' process) to the '`irc.client`' processes, that handle them — either by discarding (if the user *just* left a room for which an event was received) or reporting back to the remote IRC user (again via the '`irc.interface`' process).

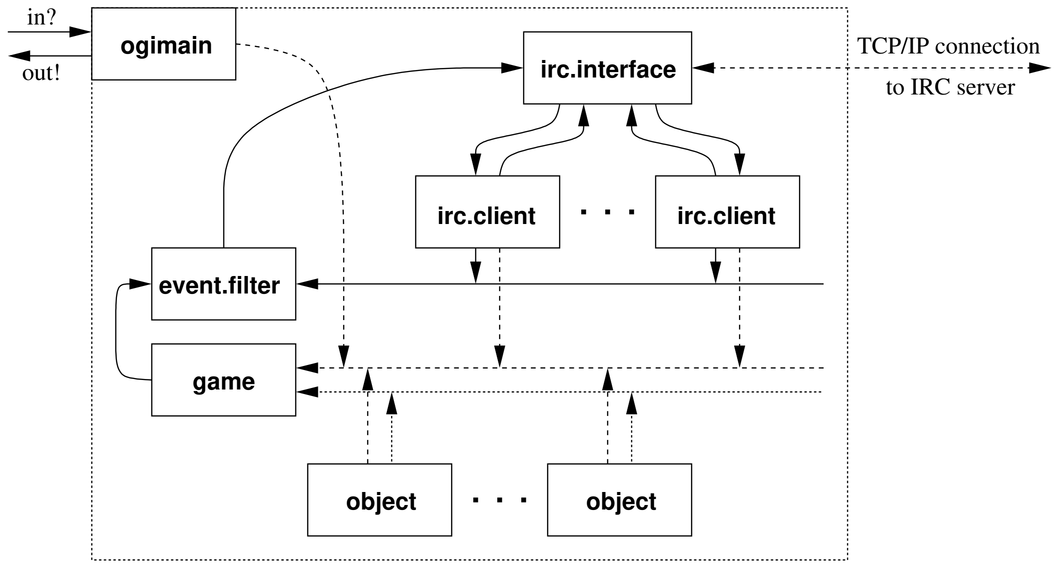

Figure 6: occam 'adventure game' process network

The other entities within the game are '`object`'s. These can either be 'held' by a user, or 'free' in a room. The particular state of objects is maintained by the '`game`' process, that controls user interactions with those objects.

The current implementation, of 'object's in particular, is somewhat unsatisfactory — particularly in scalability. With the recently added support for basic nested `MOBILE` types (particularly dynamic arrays of mobile channel-ends), the opportunity for a more natural implementation arises: with rooms, users and objects as individual processes connected by mobile channel-types. The existing implementation converges in the '`game`' process, that maintains the global state. In a future implementation (already begun), this state is recorded by the run-time connectivity of the network, and no longer centralised. Figure 7 shows an example of the proposed process network.

Such a network, besides being easier to implement, also provides the opportunity for dynamic re-configuration. This is already used in some respect, to *move* objects and users between rooms. But it also allows the layout of rooms to be changed at run-time — the inserting of new rooms or removal of existing rooms, or even more exotic, the migration of rooms (that can be used to implement fun features such as 'lifts').

The proposed implementation, unlike the existing network, uses '`web.client`' processes to handle individual web-clients. Previously, the functionality of these was handled within the '`game`' process (for instance, the queueing of messages for clients). Internally, the rooms in the game see only 'users' and 'objects', in addition to their immediate neighbours. They are not aware of whether a particular user is interacting through the web-server, the IRC

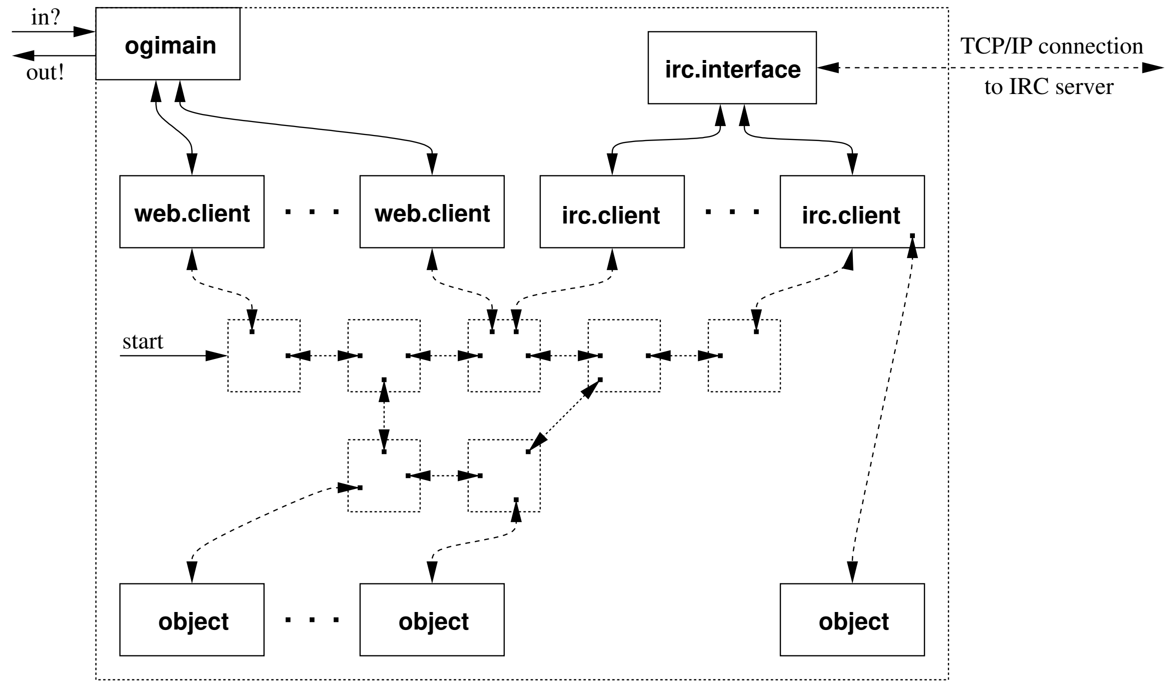

Figure 7: New 'adventure game' process network

interface, or through any other interface that may be introduced at a later date.

The '`start`' client-shared channel is connected to the '`ogimain`' and '`irc.interface`' processes (not shown). This is used to feed 'user' channel-ends into the network when they first connect to the game (when the appropriate client process is `FORK`ed to handle them).

2.5 Back-end Processing

The back-end of the web-server consists of the single process '`be.proc`'. This acts as a drain for 'used' connections from various processes within the network. If the client request resulted in an error being returned, or the connection was not marked as 'keep-alive', then the connection to the client is terminated. Connections marked as 'keep-alive' (and that did not cause an error), have their internal state cleared and are output into the front-end farm to wait for another request.

In addition to recycling connections, '`be.proc`' uses the information gathered within the connection structure to update a global '`stats`' process. This includes the number of bytes received from and sent to the client, and the time spent in various parts of the web-server. These run-time statistics can be accessed through the special URL "`/stats.html`", that the '`cache.hash.request`' process intercepts (and re-directs to the '`stats`' process).

This information is also used to report a 'hit' in the web-server log-file, including the source IP address, client request string and server response code.

2.6 Auxiliary Processes

In addition to the main components shown in figure 1 (page 253) are a number of auxiliary processes: the '`stats`' run-time statistics process; the '`log`' process; and the '`config`' process.

The '`stats`' process collects various run-time statistics, reported by the web-server components using a shared channel. The information collected includes general connection statistics, the sizes of the various process farms and the contents of the page cache (including non-cached pages). The statistics can be extracted by requesting the page '`/stats.html`', that '`cache.hash.request`' re-directs into '`stats`'. The contents of the page-cache are retrieved by querying the '`cache.control`' process, rather than having '`cache.control`' report the cache state whenever it changes.

The '`log`' process simply acts a multiplexer between two channels in a channel-type and the top-level '`screen`' channel. The input to the '`log`' process is a server-end of the '`LOG`' channel-type, defined as:

```
CHAN TYPE LOG
  MOBILE RECORD
    CHAN P.LOG log?:
    CHAN BYTE c?:
:
```

The '`P.LOG`' protocol defines three variants for `BYTE`s, `INT` counted `BYTE` arrays, and mobile `BYTE` arrays. Ideally, processes should try to use the mobile variant, since communication is unit-time. The separate `BYTE` channel '`c`' is provided so that the standard (KRoC) library functions, '`out.string`' for example, can be used easily.

The '`config`' process provides the web-server components with their configuration. This includes, for example, the file-system path for static-pages, prefix for CGI scripts, and any limits on the sizes of the various dynamic process farms. Currently, this information is provided statically at compile time (to the '`config`' process), and typically accessed only once by the various components as they initialise. Although there is currently no mechanism for adjusting the configuration (within the '`config`' process) at run-time, other web-server components can be forced to reload their configuration by means of the '`acceptor`' sending out a specially marked connection.

3 Performance

This section presents some early performance benchmark results for the occam web-server. The 'httperf' [31] benchmark program is used for the majority of benchmarks. This tries to 'hit' a web-server at a specified rate, by maintaining multiple connections. At a rate of 100 hits/second, for example, 'httperf' will attempt a new request every 10 ms. New requests may create new connections, or re-use old connections that have been marked as 'keep-alive'. This reduces some of the TCP overheads (disconnect and reconnect between requests), but most HTTP clients (web-browsers) do not make sensible use of the functionality (although this is gradually improving).

Figure 8 shows the request and response rates for Apache and the occam web-server, with increasing 'attempted' requests-per-second. As can be clearly seen in the results, the occam web-server starts to lose performance at 700 requests/second (175 connections/second). Apache only starts to lose performance at around 1300 requests/second (325 connections/-second), at which point the network throughput reaches its limit (approximately 10 megabytes per second).

Figure 9 shows the corresponding network bandwidth and maximum number of concurrent TCP connections. As soon as the web-server being benchmarked becomes unable to satisfy requests, the number of concurrent connections increases sharply — 'httperf' opens more connections and sends more requests in an attempt to maintain the requested rate. The result of this is that performance is degraded even further, as significantly more connections must be managed concurrently.

The benchmark results shown here are for a standard Apache installation (without any specific performance tweaks), and the occam web-server with both statistics-collection and

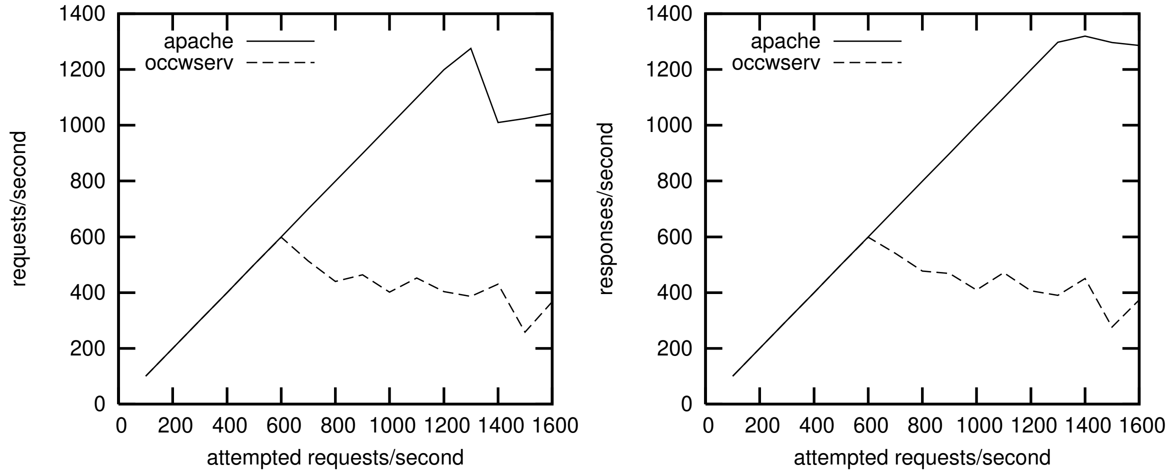

Figure 8: Request and response benchmark results ($20,000$ hits, 4 requests/connection, 8k static page)

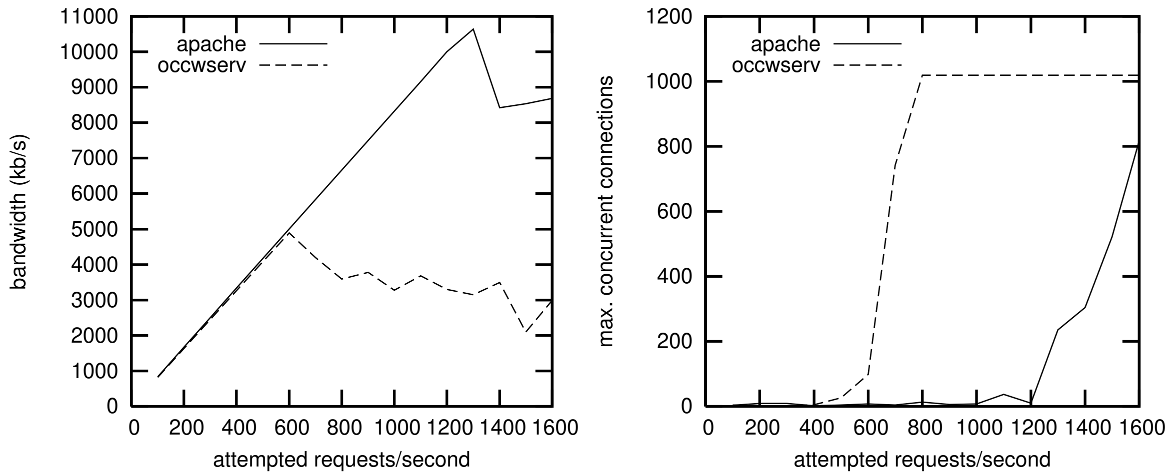

Figure 9: Network bandwidth and max. concurrent connections benchmark results ($20,000$ hits, 4 requests/connection, 8k static page)

post-mortem debugging [32] enabled. This gives a moderate disadvantage to the occam web-server, but not enough to account for the difference in performance. Furthermore, the occam web-server was compiled without any inlining optimisations. When enabled, these improve the performance of communication considerably.

A certain amount of overhead is due to the blocking system-calls mechanism [7], that has a *minimum* overhead of 9 us for the dispatch-collect cycle (measured on the same P3-800). Each connection, with four requests, requires 13 blocking calls — one for the initial 'accept' and three for each request (read request, write headers, send data).

The blocking system-calls mechanism is currently undergoing some optimisations, that should dramatically improve their performance, particularly for the web-server. It is hoped that these optimisations will be in place before the conference, where new results can be presented.

4 Conclusions and Future Work

Although the occam web-server may not perform as well as Apache, there is much value in the security gained by using occam. In particular, there is no opportunity for 'buffer-overflow' exploits, which have previously been found in Apache and Microsoft's IIS. Much effort is often spent in the formal verification of C (and C++) programs, particularly as regards the use of pointers. The purpose of such verification is largely to determine the absence of buffer-overflow opportunities and ensure the correct use of pointers (that they are initialised, for example). For occam, verification (of this type) is performed automatically by the compiler, and is hugely aided by the lack of explicit pointer types. Automatic verification for C code is difficult, but has been the subject of much research. With particular regard to internet servers and buffer-overrun possibilities is [33], that advocates the use of static checking — techniques that the occam compiler already employs, although at a less complex level. occam, however, has only recently become suitable for constructing such server applications.

There is clearly no functional deficiency from using occam to implement a web-server — this one works. Performance, however, is a major issue which must be addressed if the occam web-server is become a viable alternative to servers such as Apache and IIS. Fortunately, work is underway which should significantly improve the performance of the web-server (that has, until recently, been constrained by a lack of time).

In providing mechanisms that allow multiple clients to interact, the occam web-server does particularly well. This is largely the result of utilising concurrency, where creating a process that manages multiple client connections is simple — the server can control the clients (or process representations of those clients). In a non-concurrent web-server, interaction of this nature will typically be client-driven — driven by the sequential *flow-of-control*. Concurrent implementations, as demonstrated by the 'adventure game' OGI, are largely intuitive — and are much easier to program.

The use of occam for the construction of a web-server is just one example from a wider field of application — all 'internet' servers. One server type that is planned for implementation is a mail server, at least RFC-821 [34] (SMTP) compliant and with some local delivery capability. Mail servers have a less urgent requirement for real-time responses (most connections are automated), but have an equal, if not greater, need for security and reliability. This can certainly be seen from past failings, particularly in the widely-used '`sendmail`' mail-server (some of which are examined in [35]).

4.1 Handling Large Numbers of Simultaneous Clients

One area in which the occam web-server suffers slightly is the handling of large numbers of simultaneous clients. Because each client connection is handled individually, the overheads increase linearly with the number of simultaneous connections. This is particularly relevant to the blocking call dispatch/collect mechanism, whose overhead is significant (around 9 us). A more significant issue, however, is that the number of concurrent blocking-calls supported is limited (in order to avoid excessive numbers of 'clone' processes being created).

Non-concurrent web-servers typically use '`select()`' to 'wait' for activity from a number of clients (either for reading or writing), an operation that only involves a single blocking call — and is reasonably efficient for small numbers of 'clients' (socket file-descriptors). Placing the 'multiple wait and read' functionality within a single blocking-call offers two advantages for the occam web-server: reducing the number of simultaneous blocking calls; and following from that, reducing the impact of the blocking-call overhead itself.

Figure 10 shows an example of how this network might be constructed (for reading client requests). In principal, the total number of '`n.select.read`' blocking-call 'processes' is limited (regardless of the number of connections), but the frequency of their use varies. The network is constructed using a pipeline of '`timed.buffer`' processes, with the most-active process at the left and least-active at the right. Each '`timed.buffer`' process acts as a 'collector' for connections, until a pre-determined timeout is reached. It then makes the blocking call to wait and read from all connections, until that same timeout is reached again (if desired). When the blocking-call returns, connections that have remained inactive are forwarded to the next '`timed.buffer`' process (with a longer timeout). Connections on which the entire request has been received are forwarded to a modified front-end farm — for distribution to the rest of the web-server (this distribution could be done locally, however, to be efficient). Connections that have only sent a partial request stay put.

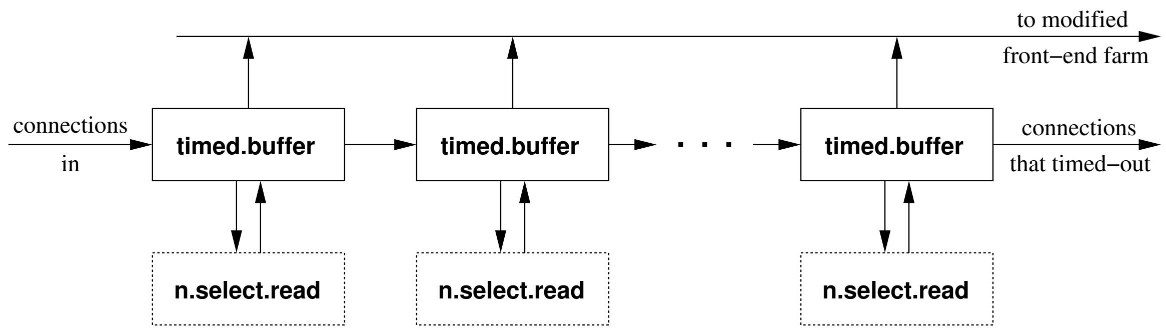

Figure 10: Example client request-reading process pipeline

In theory, the use of such a connection-handling process pipeline should increase performance dramatically. Client connections that are slow (perhaps deliberately — as is the case in 'denial of service' attacks), migrate towards the right-hand side of the process pipeline, where they will not interfere with other clients. Ultimately, inactive connections simply time-out, at which point they discarded (to avoid wasting resources associated with them).

A similar approach can also be taken for handling multiple client writes. However, the current implementation makes use of the '`sendfile()`' system-call, that simply copies data between descriptors — completely within the OS kernel, making it a very suitable operation for returning static files to client connections. Whether using a 'select' based approach to writing data to clients instead remains to be investigated.

4.2 Raw-Metal Web-Serving

Currently under development is a CSP based operating-system environment, RMoX [36]. This is programmed in occam and makes extensive use of the same dynamic facilities (particularly dynamic process creation and channel-end mobility).

Within this environment, blocking system-calls (and their associated overheads) simply do not exist — any interaction with the operating-system is via a (low-cost) channel communication. Thus, the full parallelism can remain, resulting in a simpler implementation.

Integrating the web-server into RMoX is expected to be relatively straight-forward, once the necessary functionality (TCP socket communication) has been implemented.

References

[1] Apache Software Foundation. Apache HTTP Server Project, June 2003. Available at: `http://httpd.apache.org/`.

[2] Abhishek Chandra and David Mosberger. Scalability of Linux event-dispatch mechanisms. In *Proceedings of the 2002 USENIX Annual Technical Conference*, Boston, Massachusetts, USA,

June 2001. Available from: `http://www.usenix.org/publications/library/proceedings/usenix01/chandra%.html`.

[3] Inmos Limited. occam 2.1 Reference Manual. Technical report, Inmos Limited, May 1995. Available at: `http://www.wotug.org/occam/`.

[4] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. ISBN: 0-13-153271-5.

[5] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997. ISBN: 0-13-674409-5.

[6] P.H. Welch, J. Moores, F.R.M. Barnes, and D.C. Wood. The KRoC Home Page, 2000. Available at: `http://www.cs.ukc.ac.uk/projects/ofa/kroc/`.

[7] F.R.M. Barnes. Blocking System Calls in KRoC/Linux. In P.H. Welch and A.W.P. Bakkers, editors, *Communicating Process Architectures*, volume 58 of *Concurrent Systems Engineering*, pages 155–178, Amsterdam, the Netherlands, September 2000. WoTUG, IOS Press. ISBN: 1-58603-077-9.

[8] F.R.M. Barnes. occwserv – an occam Web-Server, September 2001. CPA-2001 SIG presentation, University of Bristol, UK. Available at: `http://frmb.org/publications.html`.

[9] Peter H. Welch and David C. Wood. Higher Levels of Process Synchronisation. In A. Bakkers, editor, *Parallel Programming and Java, Proceedings of WoTUG 20*, volume 50 of *Concurrent Systems Engineering*, pages 104–129, Amsterdam, The Netherlands, April 1997. World occam and Transputer User Group (WoTUG), IOS Press. ISBN: 90-5199-336-6.

[10] F.R.M. Barnes and P.H. Welch. Prioritised Dynamic Communicating Processes: Part I. In James Pascoe, Peter Welch, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, WoTUG-25, Concurrent Systems Engineering, pages 331–361, IOS Press, Amsterdam, The Netherlands, September 2002. ISBN: 1-58603-268-2.

[11] F.R.M. Barnes and P.H. Welch. Prioritised Dynamic Communicating Processes: Part II. In James Pascoe, Peter Welch, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, WoTUG-25, Concurrent Systems Engineering, pages 363–380, IOS Press, Amsterdam, The Netherlands, September 2002. ISBN: 1-58603-268-2.

[12] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol – HTTP/1.0. RFC 1945, Internet Engineering Task Force, May 1996.

[13] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2068, Internet Engineering Task Force, January 1997.

[14] Mark J. Cox, Ralf S. Engelschall, Stephen Henson, and Ben Laurie et al. OpenSSL (Secure Sockets Layer), April 2003. Available at: `http://www.openssl.org/`.

[15] Simon Marlow. Writing High-Performance Server Applications in Haskell Case Study: A Haskell Web Server. Technical report, Microsoft Research Ltd., Cambridge, UK., July 2000. Available at: `http://citeseer.nj.nec.com/marlow00writing.html`.

[16] Claes Wikstrom et al. YAWS: yet another web-server, June 2003. Available at: `http://yaws.hyber.org/` or `http://sourceforge.net/projects/erlyaws/`.

[17] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, January 1996. ACM Press, New York. ISBN: 0-89791-769-3. Available at: `http://citeseer.nj.nec.com/jones96concurrent.html`.

[18] Joe Armstrong. Concurrency Orientated Programming in Erlang, November 2002. Invited talk at the Lightweight Languages Workshop 2002, MIT. Available at `http://www.sics.se/~joe/talks/ll2_2002.pdf`.

[19] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, New York, 1996, reprint 1997.

[20] Ingo Molnar et al. Answers from Planet TUX: Ingo Molnar Responds, July 2000. Interview, available at: `http://slashdot.org/interviews/00/07/20/1440204.shtml`. Page last visited June 2003.

[21] Red Hat, Inc. *Red Hat Content Accelerator Manuals*, 2003. Available at: `http://www.redhat.com/docs/manuals/tux/`.

[22] Chuck Lever, Marius Eriksen, and Stephen Molloy. An analysis of the TUX web server, November 2000. CITI Technical Report 00-8. Available from: `http://citeseer.nj.nec.com/lever00analysis.html`.

[23] F.R.M. Barnes and P.H. Welch. Prioritised dynamic communicating and mobile processes. *IEE Proceedings – Software*, 150(2), April 2003.

[24] NCSA Software Development Group. *CGI/1.1: Common Gateway Interface Version 1.1*. W3C: World Wide Web Consortium, 1999. Available from: `http://www.w3.org/CGI/`. Last accessed June 2003.

[25] PHP Group. PHP: Hypertext Preprocessor. Available at: `http://www.php.net/`.

[26] Netscape. *JavaScript Reference*, 2001. Available from: `http://devedge.netscape.com/central/javascript/`. Last visited June 2003.

[27] Fred Barnes. *Socket, File and Process Libraries for occam*. Computing Laboratory, University of Kent at Canterbury, June 2000. Available at: `http://www.cs.ukc.ac.uk/people/rpg/frmb2/documents/`.

[28] Frederick R.M. Barnes. *Dynamics and Pragmatics for High Performance Concurrency*. PhD thesis, University of Kent at Canterbury, June 2003.

[29] J. Oikarinen and D. P. Reed. Internet relay chat protocol. RFC 1459, Internet Engineering Task Force, May 1993.

[30] C. Kalt. Internet relay chat: Client protocol. RFC 2812, Internet Engineering Task Force, April 2000.

[31] David Mosberger and Tai Jin. httperf: A Tool for Measuring Web Server Performance. In *First Workshop on Internet Server Performance*, pages 59—67, Madison, WI, June 1998. ACM. Available at: `http://citeseer.nj.nec.com/mosberger98httperf.html`.

[32] D.C. Wood and F.R.M. Barnes. Post-Mortem Debugging in KRoC. In P.H. Welch and A.W.P. Bakkers, editors, *Communicating Process Architectures, Proceedings of WoTUG 23*, volume 58 of *Concurrent Systems Engineering*, pages 179–192, Amsterdam, the Netherlands, September 2000. WoTUG, IOS Press. ISBN: 1-58603-077-9.

[33] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000. Available at: `http://citeseer.nj.nec.com/wagner00first.html`.

[34] J. B. Postel. Simple mail transfer protocol. RFC 821, Internet Engineering Task Force, August 1982.

[35] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 6th USENIX Security Symposium*, San Jose, CA, USA, 1996. Available at: `http://citeseer.nj.nec.com/goldberg96secure.html`.

[36] F.R.M. Barnes, C.L. Jacobsen, and B. Vinter. RMoX: a Raw Metal occam Experiment. In *Communicating Process Architectures 2003*, WoTUG-26, Concurrent Systems Engineering, IOS Press, Amsterdam, The Netherlands, September 2003.