A Runtime System for Interactive Web Services

Claus Brabrand Anders Møller Anders Sandholm
Michael I. Schwartzbach
BRICS,* Department of Computer Science
University of Aarhus, Denmark
{brabrand,amoeller,sandholm,mis}@brics.dk

March 1, 1999

Abstract

Interactive web services are increasingly replacing traditional static web pages. Producing web services seems to require a tremendous amount of laborious low-level coding due to the primitive nature of CGI programming. We present ideas for an improved runtime system for interactive web services built on top of CGI running on virtually every combination of browser and HTTP/CGI server. The runtime system has been implemented and used extensively in <bigwig>, a tool for producing interactive web services.

Keywords: CGI, Interactive Web Service, Web Document Management, Runtime System, Session Model.

1 Introduction

An interactive web service consists of a global shared state (typically a database) and a number of distinct sessions that each contain some local private state and a sequential, imperative action. A web client may invoke an individual thread of one of the given session kinds. The execution of this thread may interact with the client and inspect or modify the global state.

One way of providing a runtime system for interactive web services would be to simply use plain CGI scripts [5]. However, being designed for much simpler tasks, the CGI protocol by itself is inadequate for implementing the session concept. It neither supports long sessions involving many user interactions nor any kind of concurrency control. Being the only widespread standard for running web services, this has become a serious stumbling stone in the development of complex modern web services.

We present in this paper a runtime system built on top of the CGI protocol that among other features has support for sessions and concurrency control. First, we motivate the need for a runtime system such as the one presented here. This is done by

*Basic Research in Computer Science,
Centre of the Danish National Research Foundation.

presenting its advantages over a simple CGI script based solution. Afterwards, a description of the runtime system, its different parts, and its dynamic behavior is given. We round off with a discussion of related work, a conclusion, and directions for future work.

In the appendices, we briefly describe an implementation of the suggested runtime system. Also, we give a short presentation of `<bigwig>` [4], which is a tool for producing interactive web services that makes extensive use of the self-contained runtime system package.

2 Motivation

The technology of plain CGI scripts lacks several of the properties one would expect from a modern programming environment. In the following we discuss various shortcomings of traditional CGI programming and motivate our solution to these problems, namely the design of an improved runtime system built on top of the standard CGI protocol.

2.1 The session concept

First, we will describe and motivate the concept of an interactive web service.

The HTTP protocol was originally designed for browsing *static* documents connected with hyperlinks. CGI together with forms allows *dynamic* creation of documents, that is, the contents of a document are constructed on the server at the time the document is requested. Dynamic documents have many advantages over static documents. For instance, the contents of the documents can be *tailor-made*, and *up-to-date*.

A natural extension of the dynamic-document model is the concept of *interactive* services, which is illustrated in Figure 1. Here the client does not browse a number

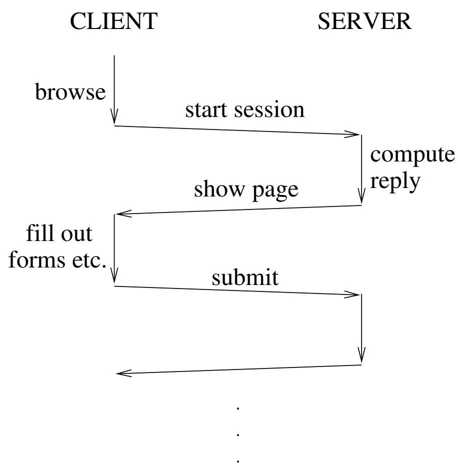

Figure 1: An interactive web session

of more or less independent statically or dynamically generated pages but is guided through a *session* controlled by a session thread on the server. This session can involve a number of user interactions. The session is initiated by the client submitting a "start session" request. The server then starts a thread controlling the new session. This thread generates a reply page which is sent back to the client. The page typically contains some input fields that are filled in by the client. That information is sent to the server, which then generates the next reply, and so on, until the session terminates.

This session concept allows a large class of services to be defined. However, a number of practical problems needs to be solved in order to implement this model on top of the CGI model.

2.2 CGI scripts and sequential session threads

As explained above, a web service session consists of a sequential computation that along the way presents information to the client and waits for replies. However, CGI is a state-less protocol, meaning that execution of a CGI script only lasts until a page is shown to the web client. This fact makes it rather tedious to program larger web services involving many client interactions. The sequential computation has to be split up into the small bits of computation that happen between client interactions. Each of these small bits will then constitute a CGI script or an instance of a CGI call.

Furthermore, to achieve persistency of the local state, one has to store and restore it explicitly between CGI-calls, for instance "hidden" in the web page sent to the client. For simple services where the full session approach is not needed this stateless-server approach might be preferable, but it is clearly inadequate in general.

Thus, the problem of forced termination of the CGI script at each client interaction is two-fold:

- Having to deal with many small scripts makes the *writing* and *maintenance* of a web service rather difficult because the control-flow of the service tends to become less clear from the program code.
- Starting up a whole new process every time a client interaction is performed is expensive in itself. On top of this a complete image of the local state has to be stored and restored each time a client interaction is required. The local state can potentially hold a lot of data, such as database contents. Thus one gets a *substantial overhead* in the execution of a web service.

We provide a simple solution which splits CGI scripts into two components, namely *connectors* and *session threads*. A connector is a tiny transient CGI script that redirects input to a session thread, receives the response from that thread, and redirects it back to the web client. The session threads are persistent processes running residently on the web server. They survive CGI calls and can therefore implement a long sequential computation involving several client interactions. The use of transient connectors and persistent session threads decreases the difficulty of writing and maintaining web services. Furthermore, it improves substantially on the overhead of the web server during execution of a service.

2.3 Other CGI shortcomings

Traditionally, reply pages from session threads are sent directly to the client. That is, the session thread (or the connector if using the system described above) writes the page to standard-output and the web server sends it on to the client browser. This basic approach imposes some annoying problems on the client:

- The client is not able to use "bookmarks" to identify the session, since selecting a bookmark might imply resending an old query to the server while the server expects a reply to a more recent interaction. It would be natural to the client if selecting a bookmarked session would continue the session from its current state. Obviously, this requires the server to always keep some kind of backup of the latest page sent to the client.

- In the session concept described in the previous section, it does not make sense to roll back execution of a session thread to a previous state. A thread can only be continued from its current point of execution. As a result of sending pages directly using the standard-output method, every new page shown to the client gets stacked up in the client's browser. This means that the stack of visited pages becomes filled up with references to outdated pages. One result is that the "back" button in the browser becomes rather useless.

We suggest a simple solution where—instead of sending the reply itself—the session thread writes its reply to a file visible to the client and then sends to the client a *reference* to the reply file. By choosing the same URL for the duration of the session, this reference can then function as an identification of that particular session. This solves both the problem with bookmarks and with the "back" button. Pressing "back" will now bring the client back to the web page where he started the session, which seems like a natural effect.

This method also opens up for an easy solution to another problem. Sometimes the server requires a long time to compute the information for the next page to be shown to the client. Naturally, the client may become *impatient* and lose interest in the service or assume that the server or the connection is down if no response is received within a certain amount of time. If confirmation in the form of a temporary response page is sent, the client will know that something is happening and that waiting will not be in vain.

This extra feature is implemented in the runtime system as follows. If a response is not ready within for instance 8 seconds, the connector responds with a reference to a temporary page (for instance saying "please wait") and terminates. This page will then automatically be loaded by the clients web browser and reload itself, say every 5 seconds. Once the session thread finishes its computation and the real response page is ready, the thread just replaces the temporary page with the real response page. This will have the effect that next time the page is reloaded, the real response page will be shown to the client.

This reloading can be done with standard HTML functionality. Of course the reloading causes some extra network traffic, but using this method is probably as close as one gets to server pushing in the world of CGI programming.

2.4 Handling safety requirements consistently

Another serious problem with traditional CGI programming is that concurrency control, such as synchronization of sessions and locking of shared variables, gets handled in an ad-hoc fashion. Typically, this is done using low-level semaphores supplied by the operating system.

As a result, web services often implement these aspects incorrectly resulting in unstable execution and sometimes even damaging behavior.

Our solution allows one to put safety requirements, such as mutual exclusion or much more complex requirements, separately in a centralized supervising process called the controller. This approach significantly simplifies the job of handling safety requirements. Also, since each of the requirements can be formulated separately, the solution is much more robust towards changes in various parts of the code.

It is generally considered inefficient and unsafe to have centralized components in distributed systems. However, in this case the bottleneck is more likely to be the HTTP/CGI server and the network than the safety controller. In spite of that, we do try to distribute the functionality of our safety controller as discussed in Section 5.

3 Components in the Runtime System

At any time there will be a number of *web clients* accessing the *HTTP/CGI server* through the CGI protocol. On the server side we will have a *controller* and a number of *session threads* running. The session threads access the global data and produce response pages for the web clients. From time to time a *connector* will be started as the result of a request from a web client. The connector will make contact with the running session thread. A connector is shut down again after having delegated the answer from a session thread back to the web client.

In the following we give a more detailed description of these components. For an overview of the components in the runtime system, see Figure 2.

Web clients Web clients are the users of the provided web service. They make use of the service essentially by filling in forms and submitting HTTP/CGI requests using a browser.

The HTTP/CGI server The HTTP/CGI server handles the incoming HTTP/CGI requests by retrieving web pages and starting up appropriate CGI scripts, in our case connectors. It also directs response pages back to the web clients.

Session threads Session threads are the resident processes running on the web server surviving several CGI calls. They represent the actual service code that implements the provided web service. They do calculations, search databases, produce response web pages, etc.

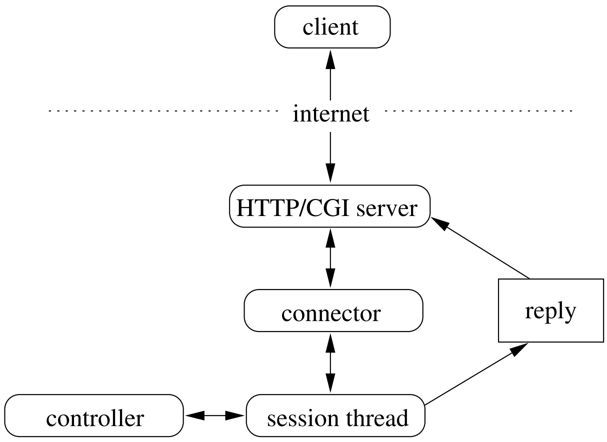

Figure 2: The runtime system

Connectors When a web client makes a request through the server, a connector is started up. If this request is the first one made, the controller starts up a new session thread corresponding to the request made by the web client. Otherwise—that is, if the web client wants to continue execution of a running session thread—the connector notifies the relevant session thread that a request has been made and forwards the input to that thread.

Reply pages Each session thread has a designated file which contains the current web page visible to the client of the session. When writing to this file, the whole contents is through a buffer updated atomically since the client may read the file at any time.

The controller The controller is a central component. It supervises session threads and has the possibility of suspending their execution at various points. This way it is ensured that the stated safety requirements are satisfied.

Furthermore, the runtime system also contains a *global-state database* (could be the file-system or a full-fledged database), and a *service manager*, which takes care of garbage-collecting abandoned session threads and other administrative issues.

4 Dynamics of the Runtime System

In this section we describe the dynamic behavior of the runtime system. We start by explaining the overall structure of the execution of a session thread. Starting from this, we present each of the possible thread transitions.

First, it is described how a session thread is started. Then, transitions involving interaction with a web client, that is, showing web pages and getting replies, are dealt with. Finally, the transitions involving interaction with the controller are presented.

For each transition we give a description of the components involved and their interaction.

4.1 Execution of a thread

The lifetime of a session thread is depicted in the diagram in Figure 3. When a thread

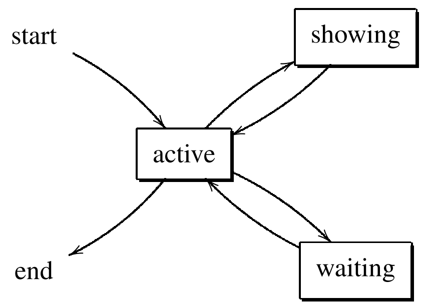

Figure 3: Possible states and transitions for a session thread

is first started, it enters the state *active*. Here it can do all sorts of computations.

Eventually it reaches a point where it has composed a response HTML page. This page is shown to the web client and the thread enters the state *showing*. Here it waits for the web client to respond via yet another HTTP/CGI request. Upon re-submission the thread reenters the state *active* and resumes execution.

Note that in the world of naive CGI programming when moving from *active* to *showing* and back one would have to store a complete image of the local state before terminating the script. Then, when started again a new process would be started and the local state would have to be reconstructed from the image that was saved. This substantial overhead of saving and restoring local state is avoided completely by the use of transient connectors and resident threads.

While in state *active* a thread can get to a point in execution where safety critical computation, such as accessing a shared resource, needs to be carried out. When reaching such a point the thread asks the controller for permission to continue and enters the state *waiting*. When permission is granted from the controller the thread reenters the *active* state and continues execution.

With a traditional approach one would have to merge the code implementing the intricate details dealing with concurrency control with the service code. This intermixing would in addition to substantially reducing the readability of the code also increase the risk of introducing errors. Our solution separates the code dealing with concurrency control from the service code.

When the session is complete, the thread will leave the state *active* and end its execution.

4.2 Starting up a session thread

This section describes the transition from *start* to *active*.

When a new web client makes an HTTP/CGI request, the server will start up a new connector as a CGI script. Since this request is the first one made by the web client, a new thread is started according to the session name given in the request. As will be described later, a response page will be sent back to the client when the thread reaches a show call or a certain amount of time, for instance 8 seconds, has passed.

When a session thread is initiated or when it moves from *showing* to *active*, the contents of the reply file is immediately overwritten by a web page containing a "reply not ready—please wait" message and a "refresh" HTML command. The "refresh" command makes the browser reload the page every few seconds until the temporary reply file is overwritten by the real reply as described in the following section. The default contents of the "please wait" page can be overridden by the service programmer by simply overwriting the reply file with a message more appropriate for the specific situation.

4.3 Interaction with the client

During execution of a running thread the service can show a page to the web client and continue execution when receiving response from the client. In the following we describe these two actions.

Showing a page

This section describes the transition from *active* to *showing*.

During execution of a session thread one can do computations, inspect the input from the client, produce response documents, etc. When a response document has been constructed and the execution reaches a point where the page is to be shown to the client, the following actions will be taken:

1. First, the document to be shown is written to the reply file as indicated in Figure 2. This file always contains a "no cache" pragma-command, so that the client browser always fetches a new page even though the same URL is used for the duration of the whole session. Unfortunately we thereby lose the possibility of browser caching, but being restricted to building on top of existing standards we cannot get it all.

2. If the connector, that is, the CGI script started by the web client, has not already terminated due to the 8 second timeout, the session thread tells it that the reply page is ready. After this, the thread goes to sleep.

3. When the connector either has been waiting the 8 seconds or it receives the "reply ready" signal from the session thread, the connector writes a location-reference containing the URL for the reply page onto standard-output (using the CGI "location" feature), and then dies.

4. Finally, the HTTP/CGI server will transmit the URL back to the web clients browser which then will fetch the reply page through the HTTP/CGI server and show it to the client.

In Figure 2, these actions describe a flow of data starting at the session thread and ending at the client.

Receiving client response

This section describes the transition from *showing* to *active*.

While the session thread is sleeping in the showing state, the web client will read the page, fill out appropriate form fields, and resubmit. This will result in the following flow of data from the client to the session thread (see Figure 2):

1. First, a request is made by the client via the CGI protocol. This request can be initiated either by clicking on a link or by pressing a submit button.

2. As a result, the HTTP/CGI server starts up a CGI script, that is, a connector.

3. The connector will then see that the client is already associated with a running thread and thus wake up that sleeping session thread and supply its new arguments.

4.4 Interaction with the controller

The controller allows the programmer to restrict the execution of a web service in such a way that stated safety requirements are satisfied.

Threads have built-in checkpoints at places where safety critical code is to be executed. At these checkpoints the thread must ask the controller for permission to continue. The controller, in turn, is constructed in such a way that it restricts execution according to the safety requirements and only allow threads that are not about to violate the requirements to continue.

In the following we describe in further detail the controller itself, what happens when session threads ask for permission, and how permission is granted by the controller.

The controller

The controller consists of three parts: some control logic, a number of checkpoint-event queues, and a timeout queue. Figure 4 gives an overview of the controller.

The control logic The control logic is the actual component representing the safety requirements. It controls whether events are enabled, and hence when the various session threads may continue execution at checkpoints. One could imagine various approaches, such as, the use of finite state machines or petri-nets. For that reason, the internals of the control logic are not specified here. The only requirement is that the interface must contain the following two functions available to the runtime system:

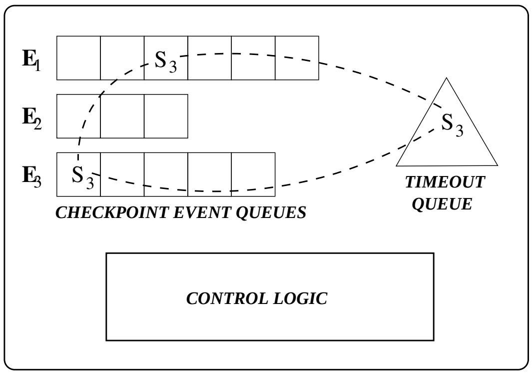

Figure 4: Components of the controller

- `check_enabled` — takes a checkpoint-event ID as argument and replies whether that event is currently enabled.
- `event_occurred` — takes the ID of an enabled checkpoint-event as argument and updates the internal state of control logic with the information that the event has occurred.

We explain in the following how these functions are used in the controller.

Checkpoint-event queues The *checkpoint-event queues* form the interface to the running threads of the service. There is a queue for each possible checkpoint event. When a thread reaches a checkpoint it asks the controller for permission to continue by adding its process-ID onto the queues corresponding to the events it wants to wait for at the checkpoint.

Timeout queue As an extra feature one can specify a *timeout* when asking the controller for permission to continue. For this purpose the controller has a timeout queue. If permission is not granted within the specified time bound, the controller wakes up the thread with the information that permission has not been granted yet, but a timeout event has occurred. The specified timeouts are put in the special timeout queue (which is implemented as a priority queue).

Asking for permission at checkpoints

This section describes the transition from *active* to *waiting*.

As mentioned earlier, one has the possibility of adding checkpoints to session code where critical code is to be executed. The runtime system interface makes some functions available to the service programmer for specifying checkpoints. Conceptually, the programmer uses them to specify a “checkpoint statement” as illustrated with an example in Figure 5. This example would have the effect that whenever a thread instance of this session reaches this point it will do the following:

```
wait {
   case E1:
      ...
   case E3:
      ...
   timeout 20:
      ...
}
```

Figure 5: A checkpoint example

1. First, it will tell the controller that it waits for either an E_1 event, an E_3 event, or a timeout of 20 seconds.
2. Having sent this request to the controller, the thread goes to sleep waiting for a response.

Controller actions

When the controller is up and running, it loops doing the following:

- If it receives a request to pass a checkpoint from a client, the controller pushes the ID of the client onto the appropriate queues. These entries are chained so that later, when permission is granted, they can all be removed at once. Figure 4 illustrates the effect of the example from Figure 5 where entries belonging to a session, S_3, are in the E_1, E_3 and TIMEOUT queues.
- If a timeout has occurred, the controller deletes the affected entries in the queues and informs the involved thread.
- Otherwise, it will look for an enabled event using the `check_enabled` function from the control logic. If the queue corresponding to an enabled event is non-empty then the controller makes the event occur by doing the following:
  1. It removes the linked entries with the thread-ID of the enabled event from the respective queues,
  2. tells the control logic that the event has occurred using the `event_occurred` function, and
  3. wakes up the involved thread with a "permission granted" signal containing the name of the event.

  If several events become enabled, a token-ring scheduling policy is used. This ensures fairness in the sense that if a thread waits for an enabled event, it will at some point be granted permission to continue.

Permission granted

This section describes the transition from *waiting* to *active*.

Having sent a request for permission to continue the thread is sleeping, waiting for the controller to make a response. If a "permission granted" signal is sent to the thread, it wakes up and continues, branching according to the event signaled by the controller. In the example checkpoint in Figure 5, if the controller grants permission for an E_1 event, execution is continued at the code following `case` E_1. If the controller sends a "timeout" signal, execution continues after `timeout`.

5 Extending the Runtime System

The runtime system described in the previous sections can be extended in several ways. The following extensions either have been implemented in an experimental version of the runtime system package or will be in near future. With these extensions, we believe that we begin reaching the limits of what is possible with the standard CGI protocol and the current functionality of standard browsers.

Distributed safety controller

To smoothen presentation, we have so far described the controller as one centralized component. In most cases it is possible to divide the control logic into independent parts controlling disjoint sets of checkpoint events. The controller can then be divided into a number of distributed control processes [10, 11]. This way the problem of the controller being a bottleneck in the system is successfully avoided.

Service monitors

Using the idea of connectors and controllers, one can construct a "remote service monitor", that is, a program run by a super-client, which is able to access logs and statistics information generated by the connectors and controllers, and to inspect and change the global state and the state of the control logic in the controllers. This can be implemented by having a dedicated *monitor process* for each service.

Secure communication

The system presented here is quite vulnerable to hostile attacks. It is easy to hijack a session, since the URL of the reply file is enough to identify a session. A simple solution is to use random keys in the URLs, making it practically impossible to guess a session ID. Of course, all information sent between the clients browser and the server, such as the session ID and all data written in forms, can still be eavesdropped. To avoid this, we have been doing experiments with cryptography, making all communication completely secure in practice. This requires use of browser plug-ins, which unfortunately has not been standardized. The protocols being used in the experiments are RSA, DES3, and RIPE-MD160. They prevent hijacking, provide secure channels, and verify user ID—all transparently to the client.

Document clusters

In the session concept illustrated in Figure 1, only one page is generated and shown to the client at a time. However, often the service wants to generate a whole “cluster” of linked documents to the client and let the client browse these documents without involving the session thread. With the current implementation, a solution would be to program the possibility of browsing the cluster into the service code—inevitably a tedious and complicated task.

Document clusters can be implemented by simply having a reply file for each document in the cluster. Recall, however, that in the presented setup, the name of the reply file was fixed for the duration of a session. That way, the history buffer of the browser got a reasonable functionality. Therefore, to get that functionality we need a somewhat different approach: the reply files are not retrieved directly by the HTTP server but via a connector process. This connector receives the ID of the session thread in the CGI query string and the document number in a hidden variable.

Single process model

If all server processes (the session threads, safety controllers, etc.) are running on the same machine, that is, the possibility of distributing the processes is not being exploited, they might as well be combined into a single process using light-weight threads. This decreases the memory use (unless the operating system provides transparent sharing of code memory) and removes the overhead of process communication. The resulting system becomes something very close to being a dedicated web server. The important difference being that it still builds upon the CGI protocol.

6 Related Work

The idea of having persistent processes running residently on the server is central in the FastCGI [8] system. One difference is that FastCGI requires platform- and server-dependent support, while our approach works for all servers that support CGI. Also, our runtime system is tailored to support more specific needs.

A more detailed and formal description of how one can make use of safety requirements written separately in a suitable logic can be found in [11, 2]. A language for writing safety requirements is presented, the compilation process into a safety controller is described, and optimizations for memory usage and flow capacity of the controller are developed. A recent paper [10] generalizes these ideas resulting in a standard scheme for generating controllers for discrete event systems with both controllable and uncontrollable events.

The Mawl language [1, 3, 7] has been suggested as a domain-specific language for describing sequential transaction-oriented web applications. Its high-level notation is also compiled into low-level CGI scripts. Mawl directly provides programming constructs corresponding to global state, dynamic document, sessions, local state, imperative actions, and client interactions. This system shows great promise to facilitate the efficient production of reliable web services. While Mawl thus offers automatic synthesis of many advanced concepts, it still relies on standard low-level semaphore programming for concurrency control. Also, it does not have a FastCGI-like solution but in instead it is possible to compile a service into a dedicated server for that particular service. Though being faster than using simple CGI scripts this solution is, as opposed to using a FastCGI-like solution, not easily ported between different machine architectures.

7 Conclusions and Future Work

The implementation as briefly described in Appendix A constitutes the core of the <bigwig> tool which currently is being developed at BRICS. In the <bigwig> tool, the runtime system we propose here has shown to provide simple and efficient solutions to problems occurring more and more often due to the increased use of interactive web services. Furthermore, the session concept seems to constitute a framework which is very natural to use for designing complex services. By basing the design of the runtime system on very widely used protocols, the system is easy to incorporate. The further development of the runtime system can be followed on the <bigwig> homepage [4].

A Implementation

A UNIX version of the runtime system has been implemented (in C) as a package "`runwig`" containing the following components (corresponding to Figure 2):

- The *connector*. It provides connection between the other components and the clients through the HTTP/CGI server.
- The *safety controller*, which handles synchronization and concurrency control. For the reasons described in Section 4.4, the control-logic is not included in the package but needs to be supplied separately.
- The *runtime library*, which is linked into the service code. It provides functions for easy interaction with the other components.

An experimental version of the runtime package implements the extensions described in Section 5. The `runwig` package—including all source code, detailed documentation, and examples—is available at `http://www.brics.dk/bigwig/runwig/`.

B <**bigwig**>

<bigwig> is a high-level programming language for developing interactive web services. Complete specifications are compiled into a conglomerate of lower-level technologies such as CGI-scripts, HTML, JavaScript, Java applets, and plug-ins running on top the runtime system presented in this paper. <bigwig> is an intellectual descendant of the Mawl project but is a completely new design and implementation with vastly expanded ambitions.

The <bigwig> language is really a collection of tiny domain-specific languages focusing on different aspects of interactive web services. To minimize the syntactic burdens, these contributing languages are held together by a C-like skeleton language. Thus, ¡bigwig¿ has the look and feel of C-programs with special data- and control-structures.

A <bigwig>service executes a dynamically varying number of threads. To provide a means of controlling the concurrent behavior, a thread may synchronize with a central controller that enforces the global behavior to conform to a regular language accepted by a finite-state automaton. That is, the 'control logic' in <bigwig> consists of finite-state automata. The controlling automaton is not given directly, but is computed (by the MONA [6, 9] system) from a collection of individual concurrency constraints phrased in first-order logic. Extensions with counters and negated alphabet symbols add expressiveness beyond regular languages.

HTML documents are first-class values that may be computed and stored in variables. A document may contain named gaps that are placeholders for either HTML fragments or attributes in tags. Such gaps may at runtime be plugged with concrete values. Since those values may themselves contain further gaps, this is a highly dynamic mechanism for building documents. The documents are represented in a very compressed format, and the plug operations takes constant time only. A flow-sensitive type checker ensures that documents are used in a consistent manner.

A standard service executes with hardly any security. Higher levels of security may be requested, such that all communications are digitally signed or encrypted using using 512 bit RSA and DES3. The required protocols are implemented using a combination of Java, JavaScript, and native plug-ins.

The familiar struct and array datastructures are replaced with tuples and relations which allow for a simple construction of small relational databases. These are efficiently implemented and should be sufficient for databases no bigger than a few MBs (of which there are quite a lot). A relation may be declared to be external, which will automatically handle the connection to some external server. An external relation is accessed with (a subset of) the syntax for internal relations, which is then translated into SQL.

An important mechanism for gluing these components together is a fully general hygienic macro mechanism that allows ¡bigwig¿ programmers to extend the language by adding arbitrary new productions to its grammar. All nonterminals are potential arguments and result types for such macros that, unlike C-front macros, are soundly implemented with full alpha-conversions. Also, error messages remain sensible, since they are threaded back through macro expansion. This allows the definition of Very Domain-Specific Languages that contain specialized constructions for building chat rooms, shopping centers, and much more. Macros are also used to wrap concurrency constraints and other primitives in layers of user-friendly syntax.

Version 0.9 of <bigwig> is currently undergoing internal evaluation at BRICS. If you want to try it out, then contact us for more information. The documentation is very rough as yet, but this has a high priority in the next few months. The project is scheduled to deliver a version 1.0 of the <bigwig> tool in June 1999. This will be freely available in an open source distribution for UNIX.

References

[1] David Atkins, Thomas Ball, Michael Benedikt, Glenn Bruns, Kenneth Cox, Peter Mataga, and Kenneth Rehor. Experience with a domain specific language for form-based services. In *Usenix Conference on Domain Specific Languages*, Santa Barbara, CA, October 1997.

[2] Claus Brabrand. Synthesizing safety controllers for interactive web services. Master's thesis, Department of Computer Science, University of Aarhus, December 1998. Available from `http://www.brics.dk/~brabrand/thesis/`.

[3] K. Cox, T. Ball, and J. C. Ramming. Lunchbot: A tale of two ways to program web services. Technical Report BL0112650-960216-06TM, AT&T Bell Laboratories, 1996.

[4] Michael I. Schwartzbach *et al.* `<bigwig>` project homepage. `http://www.brics.dk/bigwig/`.

[5] Shishir Gundavaram. *CGI Programming on the World Wide Web*. O'Reilly & Associates, Inc., 1996.

[6] N. Klarlund and A. Møller. *MONA Version 1.3 User Manual*. BRICS Notes Series NS-98-3 (2.revision), Department of Computer Science, University of Aarhus, October 1998.

[7] D. A. Ladd and J. C. Ramming. Programming the web: An application-oriented language for hypermedia services. In *4th Intl. World Wide Web Conference*, 1995.

[8] Open Market, Inc. FastCGI: A high-performance web server interface. Technical White Paper, `http://www.fastengines.com/whitepapers/`, April 1996.

[9] Anders Møller. MONA project homepage. `http://www.brics.dk/mona/`.

[10] Anders Sandholm. Mona-based control synthesis. Submitted, January 1999.

[11] Anders Sandholm and Michael I. Schwartzbach. Distributed safety controllers for web services. In Egidio Astesiano, editor, *Fundamental Approaches to Software Engineering, FASE'98*, Lecture Notes in Computer Science, LNCS 1382, pages 270–284. Springer-Verlag, March/April 1998. Also available as BRICS Technical Report RS-97-47.

Biographies

Claus Brabrand received his M.Sc. degree in computer science from the University of Aarhus, Denmark, in January 1999. He is currently a research assistant employed at BRICS at the University of Aarhus on the `<bigwig>` project. In August 1999, he will start as a Ph.D. student also at BRICS. Areas of interest and research include: domain specific languages (DSL), compilers and tools for rapid DSL construction, syntactic-level macro languages, programming languages, and Internet services.
Homepage: `http://www.brics.dk/~brabrand/`

Anders Møller is a Ph.D. student at BRICS at the University of Aarhus, Denmark, and consultant for AT&T Labs Research. From June through August 1998 he visited the Algorithms and Specification group at AT&T Labs Research. His main research interests include programming languages, logic, and verification. In particular, he works on the BRICS MONA project and the `<bigwig>` project.
Homepage: `http://www.brics.dk/~amoeller/`

Anders Sandholm received the B.Sc. and M.Sc. degrees in computer science from the University of Aarhus, Denmark, in 1994 and 1997, respectively, and expects to hand in his Ph.D. thesis during the summer of 1999.
From June through August 1996 he was Member of Technical Staff in the Computing Sciences Research Department at Bell Labs, Lucent Technologies, from June through August 1997, a Member of Technical Staff in the Algorithms and Specification group at AT&T Labs Research, and from September through December 1998, a visitor in the Software Production Research department at Bell Labs, Lucent Technologies.
He has worked in the areas of formal verification, semantics, and programming languages. His current research interests are in domain specific languages with emphasis on language design, static program analysis, and applications to control robotics and web programming.
Homepage: `http://www.brics.dk/~sandholm/`

Michael I. Schwartzbach received his Ph.D. (Computer Science) from Cornell University in 1987. He is an associate professor at the University of Aarhus and a kernel researcher at the BRICS Research Center. Michael I. Schwartzbach has studied design and implementation of programming languages, type systems, static analysis, and applications of logic.
Homepage: `http://www.brics.dk/~mis/`