A Uniform Approach to Programming the World Wide Web

Danius Michaelides, Luc Moreau and David DeRoure
University of Southampton
{dtm,L.Moreau,dder}@ecs.soton.ac.uk

Technical Report ECSTR M98/4
June 29, 1998

Abstract

We propose a uniform model for programming distributed Web applications. The model is based on the concept of Web computation *places* and provides mechanisms to coordinate distributed computations at these places, including peer-to-peer communication between places and a uniform mechanism to initiate computation in remote places. Computations can interact with the flow of HTTP requests and responses, typically as clients, proxies or servers in the Web architecture. We have implemented the model using the global pointers and remote service requests provided by the Nexus communication library. We present the model and its rationale, with some illustrative examples, and we describe the implementation.

1 Introduction

Many Web applications require a significant amount of computation which may be distributed and requires coordination; these applications use the Web infrastructure to good advantage but are often constrained by the architecture, which is fundamentally client-server. A variety of workarounds are possible and in practice ad hoc solutions are often adopted. In this paper we present a middleware layer which provides a consistent and ubiquitous model, fully integrated with HTTP while supporting distributed computations with peer-to-peer communication and a uniform mechanism for initiation of remote computations.

Even though the architecture of the World Wide Web was initially based on the client-server model, computations may be performed in a variety of locations:

- In the client-server model, the server has the responsibility of delivering the information to be displayed by the browser. The CGI interface and servlets offer users the possibility to execute programs in the server; CGI scripts can only be installed by the server administrator.
- Java applets and Java scripts allow computations to be performed in the client, i.e., browser.
- Proxies [6] are computational elements that lie along the path of the Web transaction; they are able to observe and respond to HTTP requests, and they are able to modify both the requests and the resulting documents. Applications of proxies include caching [26], personal agents [35], or on-the-fly customised link insertion [13].

- Hierarchical Internet caches, which are a form of intermediary, involve non-trivial computations. They rely on the hierarchical organisation to provide faster document delivery; they use a dedicated protocol, the Internet Cache Protocol, for communications, and they maintain the consistency of cached document versions [39].

These different kinds of computations occurring during retrieval of WWW information provide some form of load balancing which results in a reasonably efficient system. The variety of protocols and document formats that the WWW supports has made it an ubiquitous information publishing tool.

However, we are researching a new range of WWW applications that depart radically from the traditional client-server approach. These are *societies of agents* that cooperate in order to provide users with new services such as: *(i)* link integrity [31] in a publishing environment, *(ii)* finding other users with common interest [35], *(iii)* user-customised on-the-fly generation of links [13], *(iv)* information retrieval via mobile agents [11]. These applications differ from the traditional client-server model, because they involve direct client-to-client, server-to-server, or intermediary-to-intermediary communications. The WWW is open enough to support such forms of communications and computations, but ease of development is hampered by the heterogeneity of the environment. Indeed, the development process is ad hoc because it depends on the different kinds of computations, which have different semantics, are initiated in specific ways, and use different communication mechanisms.

In this paper, we advocate Wexus[1], a uniform approach to programming WWW applications. This computation model is compatible with the existing WWW as it relies on its infrastructure. Therefore, it does not penalise existing applications that do not make use of it. However, it offers a new range of facilities, which we shall illustrate with simple examples.

By devising Wexus, our goal is to facilitate the development of WWW applications conceived as societies of agents. The essence of our approach is to provide a uniform interface to programming any locations where computations may occur; servers, clients, or proxies shall be called *Web computation places*. Wexus is based on the following key ideas:

1. Peer-to-peer communications between Web computation places.
2. A uniform mechanism to initiate computations on remote places.
3. Interaction with the flow of WWW transaction.

In Section 2, we develop the three key ideas underlying the Wexus programming model. Then, we illustrate Wexus by several non-trivial examples (Section 3). Two vital components form the core of the implementation: Section 4 describes a transport layer based on HTTP and an extensible programmable HTTP server (or proxy). We then compare our solution with other approaches in Section 5.

2 The Wexus Programming Model

The distributed programming community has investigated numerous paradigms of communication for distributed environments, such as message-passing libraries [18, 19], communication channels [23, 28], remote procedure calls RPC [4] and its object-oriented variant, remote method invocation [7, 38, 40].

[1] As a shall explain later in the paper, Wexus is based on the Nexus programming model[14, 32] that we have extended to program the WWW.

Wexus is an extension of Nexus [14, 32], a distributed programming paradigm, available as a library, which provides programmers with two key ideas: *global pointers* refer to remote objects and *remote service requests* start computations on remote places. In addition, we integrate this model into the Web by providing a mechanism which maps HTTP transactions onto remote service requests.

The Nexus programming model is language independent: it is currently supported by Java [16], Scheme [30], Perl [15], C, Fortran, C++ [32]. Applications may even be written in several languages; as opposed to the multi-lingual approach [24], cooperation between different languages is via distribution and common data structures and not via threads. Furthermore, higher-level communication paradigms can be built efficiently, including remote function calls, mobile communication channels [17], or message-passing libraries.

In addition, as in languages such as Obliq [7] or Java and RMI [40], global pointers are garbage collected across distributed places, using a distributed reference counting algorithm [30].

2.1 Extended Nexus

Nexus [14] is structured in terms of five basic abstractions: nodes, contexts, threads, global pointers, and remote service requests. A computation executes on a set of *nodes* and consists of a set of *threads*, each executing in an address space called a *context*. (For the purposes of this article, it suffices to assume that a context is equivalent to a process.) An individual thread executes a sequential program, which may read and write data shared with other threads executing in the same context. We then define a Web computation place as a Nexus context.

A global pointer[2] represents a communication endpoint, that is, it specifies a Web computation place to which a communication operation can be directed. The remote service request (RSR) is the communication mechanism supported by Nexus. It is a binary and one-way mode of communication between a sender and a receiver. GPs can be created dynamically; once created, a GP can be communicated between places by including it in an RSR. A GP can be thought of as a capability granting rights to operate on the associated endpoint. The endpoint consists of both an object and its associated communication meta-data.

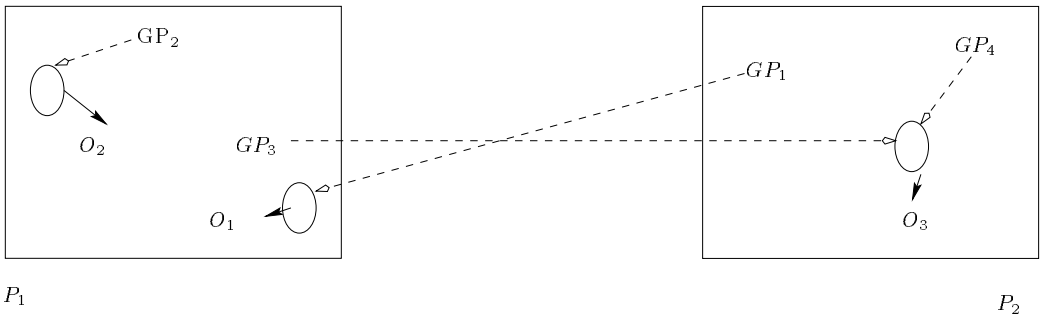

Figure 1: The Nexus Model

Figure 1 displays two Web computation places P_1, P_2. There are three endpoints, represented by circles. An endpoint is located on a Web computation place and consists of an object and

[2] A similar notion can also be found in the literature under the names of remote pointer or network pointer [5].

some meta-data. There are four globals pointers GP_1 to GP_4, each pointing at an endpoint. A global pointer may refer to a local or to a remote endpoint. We see that GP_2 refers to a local endpoint. On the other hand, O_3's endpoint is pointed at by GP_3 and GP_4.

Practically, an RSR is specified by providing a global pointer, a handler identifier, and some arguments. Issuing an RSR causes arguments to be serialised in a data buffer, the buffer to be transfered to the Web place designated by the global pointer, after which the routine specified by the handler is executed, potentially in a new thread of control. Both the arguments and the specific data contained in the endpoint associated with the global pointer are made available to the RSR handler.

The endpoint of a global pointer refers not only to the user data but also to some communication meta-data. Such meta-data contains:

- *Supported protocols*. The Nexus communication library is multi-protocol and RSRs may be transported on top of TCP, UDP and others. In addition, we have implemented a new protocol module for Nexus on top of HTTP. By tunnelling RSRs into HTTP, distributed Wexus applications can use the same protocols as the WWW itself. This choice of a single protocol, amongst others, facilitates cross-firewall applications.

- RSR *handlers*. Each endpoint is associated with a table of handlers, from which a handler is selected upon reception of an incoming RSR. At runtime, new handlers may be added or removed, and handlers may be re-defined by the user. Tables of handlers may be shared between communication endpoints, which allows programmers to define notions of object classes, able to react to a common set of RSRs.

- *Reference counters*. In Wexus, global pointers are reference counted; this facility allows Wexus applications to know when a global pointer passed to another process is no longer used. Every time global pointers are communicated, reference counters are updated according to a published algorithm [30], whose description is beyond the scope of this paper. Control messages updating reference counters are exchanged between places using the RSR communication mechanism. Such messages are generated and automatically handled by the system. A variant of our algorithm is being implemented to support massively distributed computation involving a very high number of places [29].

We shall come back to the implementation details of the HTTP-based transport layer in Section 4.1. For the time being, it is sufficient to know that tunnelling works as follows. When a RSR is issued, the data buffer is encoded into a HTTP form; then, it is sent to a HTTP server, which calls a CGI script that decodes the data and sends it to the Nexus receiver. At an abstract level, the RSR mechanism provides peer-to-peer communications, but at the HTTP layer, messages *are routed* via the HTTP server.

The peer-to-peer communication mechanisms offered by RSRs provides the programmer with a powerful abstraction. Indeed, they give the programmer the illusion of homogeneity even though communications in the current Internet are not uniform:

1. Security restrictions typically authorise Java applets to communicate only with the server that delivered the applet. Such restrictions prevent direct communications, e.g., between two applications running inside two browsers. In the absence of our communication model, the user would have to program complex code working around the security restrictions, for instance by forwarding messages via a proxy.

2. The Internet can no longer be regarded as a flat space of places due to the presence of firewalls. Direct access via TCP inside a domain is more and more frequently prevented

by firewalls. Messages may have to be routed via firewalls and Wexus provides such a mechanism of transparent routing.

3. Local area networks, clusters of workstations, or even supercomputers usually rely on fast communication protocols. By its multi-protocol approach, Nexus may automatically use these protocols instead of HTTP. Even though we have the possibility to tunnel *all* RSRs in HTTP, automatic and dynamic switching between protocols may provide better performance.

The RSR mechanism also provides a uniform way of starting remote computations as receiving a RSR fires up a handler. Currently, all the kinds of computations supported by the WWW have their own ways of being started. Applets are started when HTML pages are downloaded and further applet computation may be created by communicating via sockets. CGI scripts are started by passing a sequence of keyword-value strings. Internet caches use their own protocol (Internet Cache Protocol). Our communication layer abstracts away from these place-dependent mechanisms by providing the single notion of RSR.

2.2 Interacting with the WWW Information Flow

The second facet of our Wexus programming model is the possibility to interact with the information flow of WWW transactions. A WWW transaction is typically initiated by a browser, passed to its proxy (an intermediary), which in turn passes it to the next intermediary, until it reaches the server. The latter returns a result, forwarded by each place in the chain. All places in both directions have the opportunity to modify the information that is being transfered.

At each place of computation, we want to provide programmers with the means to interact with the flow of information. Therefore, we offer callback mechanisms that allow user-definable code to be called in response to events happening during WWW transactions. In order to provide a uniform model of programming, the callback code is defined as a Nexus handler, which is called as if the WWW transaction had been a Nexus remote service request.

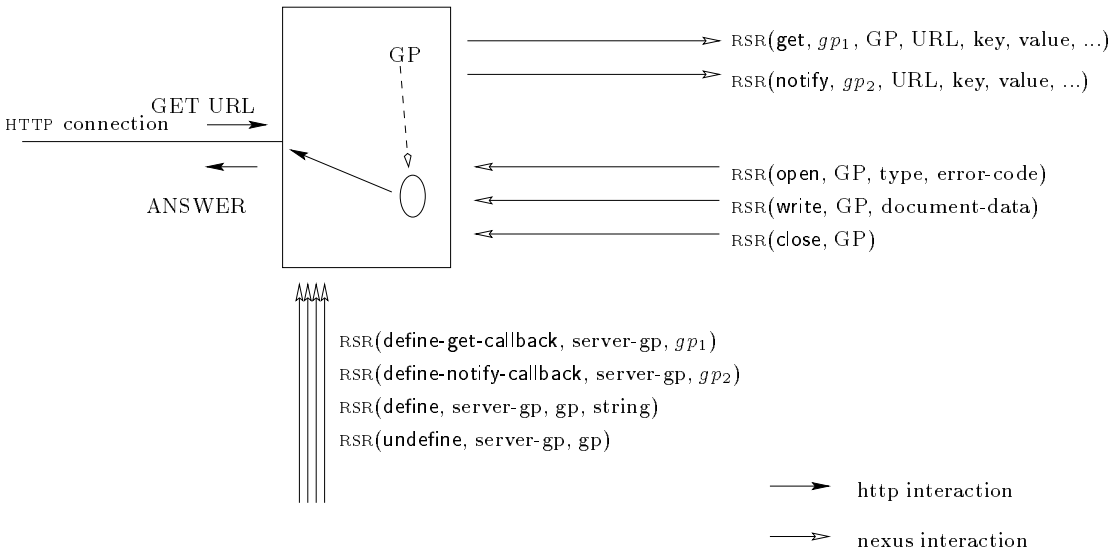

Figure 2: A Wexus Aware HTTP Server

Figure 2 describes the modes of interaction of a Wexus-aware HTTP server. The server is represented by a box accepting HTTP requests on the left-hand side. Such requests are established by a place creating a TCP connection to the server, and sending a HTTP GET request. The HTTP server is expected to return the resulting document on the same connection.

For each incoming HTTP request, the server sends a **get** RSR containing the requested URL, as well as any key-value pair specified in the URL or in the information contained in the HTTP request. This RSR is sent to a programmer-specifiable global pointer gp_1, which may be pointing either at a data inside the HTTP server itself, or at a data of another process, possibly residing on another host.

The handler which is called in response to this **get** RSR is expected to produce a document which, according to the HTTP specification, has to be returned eventually on the connection established between the client and the server. Therefore, before sending the **get** RSR, the server creates an internal data structure containing some information about the connection, and creates a global pointer GP pointing at it. This global pointer is also passed as an element of the **get** RSR.

In order to return the document to the server, a remote service request **write** has to be sent to the GP inside the server, passing it the document to be returned to the TCP connection. Several **write** RSRs can be sent and will be handled sequentially; the resulting document is obtained by concatenating each individual data. These **write** RSRs must be preceded by an **open** that indicates the type of the document; symmetrically, they must be followed by a **close**, which marks the end of the document.

The **get** RSR is said to be *synchronous* because it is called by the server, following an HTTP request, with the intent of generating the answer to the transaction. We also provide an *asynchronous* variant, called **notify**, which passes the same information (except the return GP) to another global pointer gp_2; the handler of the **notify** request it is not intended to return a document.

Besides accepting HTTP requests, issuing and receiving remote service requests, a Wexus-aware HTTP server must also be configurable: for instance, one wishes to specify the global pointers gp_1 and gp_2 to which remote service requests are sent. We further introduce two RSRs that an Wexus HTTP server must handle. The two RSRs **define-get-callback** and **define-notify-callback** specify the global pointers gp_1 and gp_2 to which **get** and **notify** must be sent, respectively.

Also, we would like HTTP requests to start a computation on any object. To this end, we provide a mechanism by which the **get** RSR can be sent to any global pointer. Requests for a URL of the type

```
protocol://host.domain:port/path1/path2/.../pathi?key_1=value_1&key_n=value_n,
```

not only package all key-value pairs into the RSR, but also handle in a particular way the reserved key **gp**. An HTTP server maintains a table associating global pointers with their string representations. When a URL uses the key **gp**, the associated string is searched in the table, and the matching global pointer is the pointer to which **get** and **notify** are sent. Consequently, we provide mechanisms to define an association between a global pointer and a string by the **define** RSR; the **undefine** RSR allows one to delete such an association.

Our approach has several benefits:

- *Conversion*. As we have adopted the single mechanism of remote service requests and associated handlers, a **get** (or **notify**) handler may be activated by sending remote service

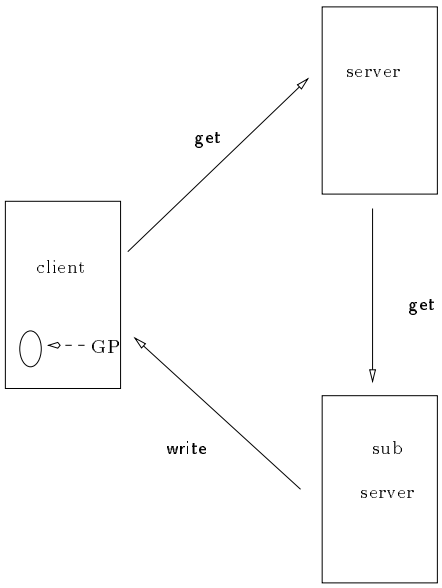

Figure 3: A Wexus-Aware HTTP Server

requests immediately to the place where the handler is defined; vice-versa, the Nexus world opens itself to non-Nexus applications via HTTP requests.

- *Composition*. All WWW computation places may adopt the same strategy, from clients to servers. They can be composed in interesting ways, such as in Figure 3, where all components are supposed to be Wexus aware. Using an HTTP GET request (or a **get** RSR), the client requests a document, but specifies a local GP as the location to return the document. The WWW server delegates the handling of the **get** request to a subserver, which can directly return the result to the client.

  If the client is not Wexus aware, the same triangular organisation can be obtained by connecting the client to a Wexus-aware proxy.

- *With and Without Connections*. The World Wide Web is connection oriented, which forces results to be returned on the same connection chain as the request. On the other hand, the Nexus model is connection-less because a request has to specify the global pointer to which a result has to be sent. Both approaches integrate nicely in this framework, where a Wexus-aware component converts GET HTTP requests into **get** RSRs, and a **write** sends results back onto a connection. The connection-oriented approach provides a model where each place in a chain can act as a filter on the requests/data. The connection-less approach permits more dynamically configurable networks and facilitates load-balancing as illustrated in Figure 3.

2.3 Summary of the Wexus Programming Model

Wexus is an extension of Nexus to the WWW. Numerous publications are available about Nexus, its different language bindings, transport protocols and performance [32]. In a previous paper, we have defined the formal semantics of Nexus as an abstract machine [30]. In this section we

present a summary of the primitives offered by Nexus, as used in this paper, and their informal definition.

The Nexus programming model may be summarised by the three constructs displayed in Figure 4. The function *make-gp* takes any object and returns a global pointer pointing at this object. The created global pointer is a first-class data-structure, which may be passed to other places using remote service requests.

Remote service requests are initiated by the **rsr** construct, which expects a handler name, a global pointer, and some arguments. The arguments will be marshalled into a data buffer, which will be sent to the place containing the endpoint associated with the provided global pointer.

(*make-gp object*)
(**rsr** rsr-name *receiver-gp val1 val2 ...*)
(**define-rsr-handler** write-rsr (*receiver-object a1 a2*)
body)

Figure 4: Three Nexus Constructs

Upon receiving the data buffer, the arguments will be extracted. The handler name given when the RSR was issued determines the routine which is called on the receiver. Such a routine must have been defined previously using the **define-rsr-handler** construct. This construct is similar to a function declaration, specifying the formal parameters and the body. When the handler is called, its formal parameters will be bound to the arguments extracted from the received data buffer. In addition, the handler is also given the user data associated with the endpoint, which is passed as a first argument, before the data contained in the buffer.

---

get-rsr	gp, URL, key_1, $value_1$, ...
notify-rsr	URL, key_1, $value_1$, ...
open-rsr	type, error-code
write-rsr	document-data
close-rsr	
define-get-callback-rsr	gp
define-notify-callback-rsr	gp
define-rsr	string, gp
undefine-rsr	string

Figure 5: Remote service requests supported by Wexus servers

Figure 5 displays the predefined RSRs and their arguments as supported by Wexus. A get request transmits a global pointer, a URL, and a variable number of key-value pairs; the handler is expected to return the document designated by the URL to the specified *gp*. The notify requests has the same arguments except the global pointer; it provides information that the URL was requested.

A get handler is expected to return a document to a global pointer. First, the type of the document (a string) and an error code (success or failure) must be sent via an open request.

Then, the document itself is sent via a sequence of write requests. Finally, a close request, without any argument, indicates the end of the document.

The define-get-callback specifies the default global pointer to which get requests must be sent. Similarly, define-notify-callback specifies the global pointer for notify.

The request define takes a string and a global pointer and creates an association between them. The undefine request takes a string and removes it from the table of associations. Names are used during the conversion of a HTTP GET request into a get RSR. Let us consider the following URL:

```
protocol://host.domain/path?key_1=value_1&key_n=value_n&gp=string_gp.
```

It is converted into the following get request

(**rsr** get-rsr *object-gp browser-gp* "path" "key_1" "value_1" "key_n" "value_n"),

with *object-gp* the global pointer associated with *string_gp*.

3 Some Applications

The authors have experience of a number of distributed Web applications which have adopted an architecture based on proxies. These include:

1. The Distributed Link Service [13], which uses a proxy to insert hypertext links on-the-fly as a document passes through towards the browser. The links are obtained by interrogating a link database. It can also report the links asynchronously.

2. In MEMOIR [35] the proxy notifies other components when a document is obtained, so that they can record the users "trail". Components register their interest in events of certain kinds and subsequently receive asynchronous notifications. The user can make complex queries which result in distributed computations, and there are also persistent computations which process data.

It is however impossible to present such real applications due to the code size. Instead, we describe several small examples which capture key functionality inherent in these applications and illustrate our programming model.

1. We implement access counters to illustrate the callback mechanisms, and synchronous or asynchronous handlers. We then show how to extend the program by involving clients in order to provide reference counters on URLs.

2. The personal cache is an example of a distributed application across several places; it also illustrates the benefits of the connection-less approach.

3. The summary example shows response to user requests and communications between servers to pre-compute some information.

4. The distributed traversal engine (Section 3.3) is an application distributed across multiple servers. It also illustrates the benefit of reference counters on global pointers.

   Note Most of the code presented in this section is written in Scheme [36]. In order to show that our approach is language-independent, we have programmed one example in C. Other languages can also be used, including Java for which there is a Nexus binding. We have adopted Scheme because it provides us with a high-level interface [30] to Nexus and powerful abstraction primitives, therefore reducing code size.

3.1 From Access Counters to Reference Counters

Sometimes, users wish to display the number of times a page was hit. To this end, they embed in the page an anchor with a URL that activates a CGI script whose purpose is to return a graphical representation of a counter. Such an anchor actually represents two different actions: incrementing the counter associated with this URL and displaying the counter value. Both actions can easily be implemented in our uniform programming model.

The essence of our solution is to create a "*counting place*" that maintains a counter with every requested URL. Such a counting place must be inserted in the WWW transaction path, somewhere between the client and a server. If it is installed as a proxy to the browser, it will count requests emitted from the browser; on the other hand, if it is attached to a server, it will count the requests arriving to the server.

In a first instance, we define a handler for the get request, which is called every time a URL is requested. By default, this handler calls the function *get-and-count-handler* displayed in Figure 6. It gets the requested document, it returns it to the specified *gp* (cf. line [1]), it increments the counter associated with the URL (cf. line [2]), and it adds an HTML epilogue displaying the counter value.

```
(define-rsr-handler get-rsr (object gp string . args)
  (if (default-object? object)
      (get-and-count-handler gp string args)
      (other-get-and-count-handler object gp string args)))
(define get-and-count-handler
  (lambda (gp url args)
    (let ((result (return-url-to-gp url gp)))                    ;; [1]
      (if (result-error? result)
          (generate-error-msg gp url args result)
          (let ((counter (increment-count! url)))                ;; [2]
            (if (equal? (result-type result) "text/html")
                (generate-counter-information gp url counter)
                (rsr close-rsr gp)))))))
(define generate-counter-information
  (lambda (gp url counter)
    (let* ((callback-gp (make-gp *table*))
           (gp-string (gp->string! callback-gp)))
      (rsr define-rsr gp gp-string callback-gp)                  ;; [3]
      (rsr write-rsr gp (make-counter-info-string url counter gp-string))
      (rsr close-rsr gp))))
(define make-callback-url
  (lambda (url string)
    (make-anchor (string-append url "?gp=" string)
                 "Click Here to Display Table")))
(define other-get-and-count-handler
  (lambda (table gp url args)
    (rsr write-rsr gp *accessed-docs-title*)
    (rsr write-rsr gp (display-table table))
    (rsr close-rsr gp)))
```

Figure 6: Page Access Counters (1)

The function *generate-counter-information* returns some HTML code displaying the counter value. In addition, it embeds in the code a URL, generated by *make-callback-url*, that contains the gp keyword associated with a string value. The counting place maintains a table associating global pointers with strings; a define RSR is used at line [3] to define the association between

the string value and the global pointer.

When the server gets a request for a URL with the gp keyword, the associated string is searched and a get RSR is sent to the corresponding global pointer. The get handler recognises it is called on a different object, and the function *other-get-and-count-handler* displays the counter table.

If we are not interested in generating a document containing the number of times it has been requested, we can also use asynchronous notifications. Such notifications are called in parallel with the retrieval of the document by the server. Figure 7 displays the handler for the notify remote service request.

```
(define-rsr-handler notify-rsr (object gp string . args)
  (notify-handler gp string args))
(define register-asynchronous-counter
  (lambda (counting-place-gp default-gp)
    (let ((table-gp (make-gp *table*)))
      (rsr define-notify-callback-rsr counting-place-gp default-gp)
      (rsr define-rsr counting-place-gp "show-table" table-gp))))
(define notify-handler
  (lambda (gp url args)
    (increment-count! url)))
```

Figure 7: Page Access Counters (2)

Again, the counter value (and the table content) can be obtained by following a special URL, created by *make-callback-url*, containing the keyword gp and the associated string `"show-table"`. The string `"show-table"` must have been defined by a define RSR (cf. line [4]).

Figures 6 and 7 show that we can define URLs that initiate computations on specific objects when they are requested by browsers. In the first case, the URL was generated on-the-fly by the counting place: such a mechanism provides a customised callback to the user. In the second case, the URL can be embedded in a document, when the document is authored, and made accessible to all users.

Page access counters can easily be transformed into *reference counters* if clients cooperate with servers. An access counter is the total number of times a document was accessed, whereas a reference counter is the number of times a document is *currently held* by browsers; in other words, the reference counter is the number of users currently viewing the document. By further cooperation from browsers, we can also maintain a reference counters of users who have bookmarked a URL.

When a client requests a URL, it allocates some space in memory for the document. As the user navigates, the client may run out of memory space and therefore decide to free the document. At that time, we can envisage that the client informs the server that the space used by the document is being freed. This can easily be implemented by issuing a remote service request to the HTTP server, requesting it to decrement the counter associated with the URL:

(**rsr** decrement-counter-rsr *server-gp url*).

Cooperation between clients and servers, or more generally between any two computation places, can be based on remote service requests. However, this mechanism requires one place to have access to a global pointer pointing at another place in order to be able to send it requests. Such a bootstrap problem is common to most distributed systems. They usually use a "registry" [40], situated at a well-known location, from which a first global pointer may

be retrieved. Nexus also provides such a mechanism in the form of a process listening on a predefined port; another process can "attach" itself to the first one, resulting in the exchange of a global pointer.

In our context, a Wexus-aware client does not, a priori, know whether a server is also Wexus-aware. However, if the client requests some data from the server, the latter may embed some information in the result. The Wexus-aware client keeps watching for such information by which it may bootstrap a communication with the server, independently of the normal flow of HTTP requests.

At the moment, we are experimenting with two techniques to propagate bootstrap information.

1. Such information may be encoded as meta-data in the returned documents. Even though this solution is simple, it requires the client to parse the resulting document. Furthermore, it is only applicable to data format which have meta information (such as HTML).

2. Another solution is to embed the information in the HTTP header, preceding the returned document. In particular, embedding this information into a cookie generated by the server is an interesting venue, because it allows servers to exercise control on places that try to establish connections via Nexus.

3.2 Webpage Summarizer

This example demonstrates the communication that may occur between servers as a result of a user getting a document. Using an interface to W3C's libwww library [33], we define a function that, given a URL, fetches the document and extracts all its anchors. This list of anchors is used to generate a summary page, containing a list of all the URLs, links to the URLs, and if the destination URL is within our domain, a link to the summary page for that URL.

The process of summarising a document takes an appreciable time, and to speed up response times, we could speculatively perform this computation. When the request for a document summary is made, the summariser first checks in its cache of summarised documents. If the document is not there, it fetches it in the usual way and constructs the summary page. In addition, it takes all the URLs in the page, and takes each of the URLs that are within our domain and notifies the relevant local webserver for that document. On receipt of such a notification, a webserver starts summarising a document, which it will put in its local cache.

The server maintains a vector of global pointers to other webservers and a vector containing the hostnames of the webservers. The function `find-id` searches the vector trying to match the hostname to the given URL. If a match is found, it calls the notify-url RSR using the relevant global-pointer. We use a regular expression for the matching, so we could match on more complex strings that just the hostname.

3.3 Distributed Traversal Engine

WWW masters often have to perform administrative tasks on HTTP servers belonging to their domain. Such tasks are, for instance, finding dangling links, building statistics on page content, discovering unaccessible islands of information, displaying the graph structure, and so on. Many of these tasks may be programmed by a single application, which relies on a library such as libWWW [33] to access remote documents via HTTP servers.

A distributed implementation uses several processes running on different servers. It permits parallel processing and usually provides better performance if the hypergraph is reasonably

```
(define notify
  (lambda (url)
    (if (html-url? url)
        (let ((index (find-id url)))
          (if (>= index 0)
              (rsr notify-url-rsr (vector-ref vhostgps index) url))))))
(define get-summary-handler
  (lambda (object gp url)
    (let* ((cache-structure (find-in-cache url cache))
           (summary (if (null? cache-structure)
                        (summarize url)
                        (return-cached-string cache-structure))))
      (rsr open-rsr gp "html/text" 0)
      (rsr write-rsr gp summary)
      (rsr close-rsr gp)
      (for-each notify (extract-urls url)))))
(define-rsr-handler notify-url-rsr (object url)
  (if (null? (find-in-cache url cache))
      (add-summary-to-cache url)))
(define add-summary-to-cache
  (lambda (url)
    (let* ((entry (create-cache-entry url))
           (summary (summarize url)))
      (store-in-entry summary entry))))
```

Figure 8: Summary Server

balanced between the different servers. Some tasks, such as creating separate link bases [12] or content indexes, require local access to the local file system; these tasks can only be performed by a distributed version, where processes running on each HTTP server have access to local resources, e.g., the local filesystem.

In this Section, we describe a generic distributed traversal engine. We call a *search task*, an instance of the traversal engine that solves a specific goal. A search task is composed of one master, where the search is initiated, and slaves cooperating with the master in order to solve the goal. Several search tasks may run in parallel, may involve different hosts, and may be initiated by different users. In Figure 9, three nodes are involved in a search. Each node recursively traverses its local HTML pages; every time a URL pointing to another member of the search task is met, a remote service request is sent to the corresponding process, which adds it to its set of URLs remaining to be visited.

In Figure 10, we display the code of the generic distributed traversal engine. Distribution is visible in the function *distributed-graph-search*, where a remote service request is issued in the line before the last, to follow a non-local URL. This remote service request activates the handler follow-rsr that adds the URL to the "todo" list, and restarts the search locally, if it was not currently active. This algorithm is generic because every time a new URL is met or a URL is undefined, user-specifiable hooks are called.

An interesting problem in this algorithm is to detect the search termination. Indeed, each node has temporarily finished a local search when its "todo" list becomes empty; however, it can be resumed as soon as it receives a remote service request. The global termination of the search only occurs when all participating nodes have terminated their local search *and* when there is no RSR in transit requesting further search.

In fact, it was proven that detecting termination of a distributed algorithm was equivalent

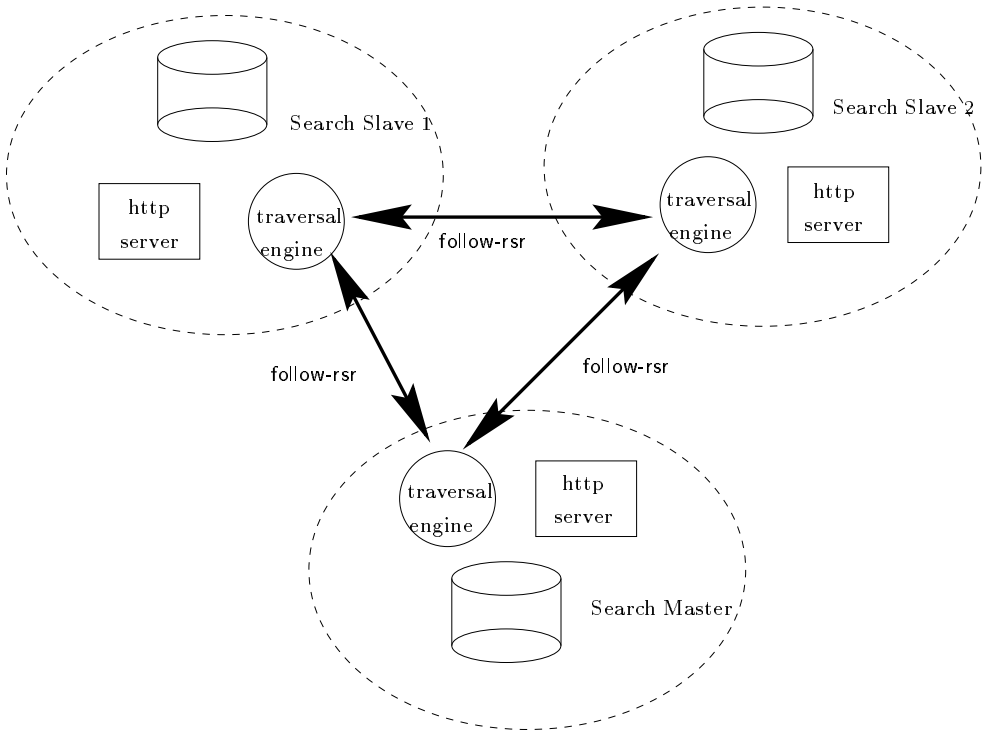

Figure 9: A Search Task Involving One Master and Two Slaves

to the problem of distributed garbage collection [41]. We therefore used this theoretical result to detect termination in this algorithm. When a search is initiated, a new object is allocated on the master and a finalizer [21] is installed for this object. When a remote service request is sent to a node in order to follow a URL, a pointer to the object, called *token*, is passed. When a node locally terminates its search, it explicitly loses its reference to the token. Termination is detected when the garbage collector detects that no reference to the token is active, and the associated finalizer is called.

3.4 Personal Caching

In this example, we demonstrate how we can program computations that occur close to the browser, using the Wexus programming model. The use of caches during Web browsing is common; browsers typically maintain a cache of recently accessed information in both memory and disk. To try and reduce both precious bandwidth and user response times, Web caches are deployed amongst groups of people. Typically, these caches communicate with each other. This example combines these two types of caches into a personal cache that communicates with other personal caches. Each user runs a personal cache on his/her machine, and this cache interrogates other personal caches for documents before making a request.

A cache process attaches to a Wexus server and registers a callback for the get method. First, we establish a RSR that queries the local cache and responds to the supplied global pointer if the the document is in the local cache. The handler responds by calling the IHave handler, which will be defined further on. Secondly, our other cache operation is to send a document to a supplied global pointer. This uses the open, write, and close RSRs that we have

```
(define distributed-graph-search
  (lambda (url search-task)
    (let loop ((seen (search-task-seen search-task))
               (todo (list url)))
      (if (null? todo)
          (let ((val (continue-search? search-task seen)))
            (if (not (null? val))
                (loop seen val)))
          (let ((url (car todo)))
            (if (member url seen)
                (loop seen (cdr todo))
                (let ((gp (get-host-gp url search-task)))
                  (if (local-gp? gp)
                      (loop (cons url seen)
                            (append! (cdr todo)
                                     (process-and-extract-urls url search-task)))
                      (begin
                        (rsr follow-rsr gp url (search-task-token search-task)) ;;; sends a RSR and passes token
                        (loop (cons url seen) (cdr todo)))))))))))
(define continue-search?
  (lambda (search-task seen)
    (nexus-mutex-lock (search-task-lock search-task))
    (let ((val (search-task-todo search-task)))
      (set-search-task-todo! search-task '())
      (if (null? val)
          (begin
            (set-search-task-seen! search-task seen)
            (set-search-task-token! search-task #f)   ;;; loses reference to token
            (set-search-task-restart?! search-task #t)))
      (nexus-mutex-unlock (search-task-lock search-task))
      val)))

(define-rsr-handler follow-rsr (search-task url token)
  (nexus-mutex-lock (search-task-lock search-task))
  (let ((restart? (search-task-restart? search-task)))
    (if restart?
        (begin
          (set-search-task-restart?! search-task #f)
          (set-search-task-token! search-task token))    ;; gets a reference to the token
        (set-search-task-todo! search-task (cons url (search-task-todo search-task))))
    (nexus-mutex-unlock (search-task-lock search-task))
    (if restart?
        (distributed-graph-search url search-task))))
```

Figure 10: Generic Distributed Traversal of WWW documents

```
static void have_document_rsr(globalpointer_t *reply_to, char *url)
{
  if (lookup_cache(url)>-1)
    call_ihave_rsr(reply_to, &myself);
}

static void send_document_rsr(globalpointer_t *send_to, char *url)
{
  int doc_id=lookup_cache(url);

  call_open_rsr(send_to, document_type(doc_id));
  call_write_rsr(send_to, document_data(doc_id));
  call_close_rsr(send_to);
}
```

Figure 11: Personal Cache (1)

defined elsewhere. These two handlers are shown in Figure 11.

Using these above basic handlers we are able to develop the **get** handler. This handler first checks if the document is in its local cache and if so, returns the document to the browser by using its own send document handler. If the document is not in the local cache, then the handler queries the other personal caches using the Have RSR. These handlers may respond via the IHave RSR. The IHave handler will issue a send document to the first response it gets, since we only want to send one copy of the document to the user. We create a data structure that stores whether the document has been sent, the URL of the required document, and the global pointer used to communicate with the browser. The **get** handler creates a new instantiation of this structure and a new global pointer to it. This global pointer is passed as an argument to the Have queries, which intern may make the IHave RSR call on this global pointer.

The interesting feature of our personal cache is that when a document is found in a neighbourhood cache, it is transfered directly back to the browser bypassing the original cache process. This demonstrates one of the benefits of passing around a global pointer for browser communication.

The V6 engine [26] is a client proxy, which can act as a cache, but is also able to filter and redirect URLs. Some of their goals are similar to ours such as providing callbacks in response to HTTP events. Our model is, however, more general because it supports distributed programming, is language independent, and uniformly integrates HTTP transactions with a form of RSR.

3.5 Other Applications

Our model of computing for the WWW is language independent. In particular, there is an implementation of Nexus in Java [16], which allows us to program WWW applications in this language. A major benefit of Java is its byte-code portability and the availability of the Java Virtual Machine in many browsers. It is therefore possible to download the java_nexus library into a browser and establish communications between any browser and computation places, using our model of WWW programming.

We have built a library that provides a remote window facility over Nexus [34]. Clients are allowed to open a display server, create windows, buttons, and other usual graphical objects, with associated actions. A point that is relevant to our discussion is that the display server is

```
typedef struct _doc_request_t
{
  int sent;
  char *url;
  globalpointer_t browser_gp;
} doc_request_t;

static void get_rsr(global_pointer_t *browser_gp, char *url)
{
  global_pointer_t *local_gp;
  int i, doc_id;

  doc_id=lookup_cache(url);
  if (doc_id>-1) {
    call_senddoc_rsr(&myself, browser_gp, url);
  } else {
    doc_request_t *user_ptr=(doc_request_t *)malloc(sizeof(doc_request_t));

    user_ptr->sent=FALSE;
    user_ptr->url=url;
    global_pointer_copy(&user_ptr->browser_gp, browser_gp);

    local_gp=make_gp(user_ptr);

    for(i=0; i<num_other_caches; i++)
      call_have_rsr(&other_caches[i].gp,url,local_gp);
    nexus_usleep(TIMEOUT);
    if (!user_ptr->sent) {
      insert_cache(url,get_document(url));
      call_senddoc_rsr(&myself, browser_gp, url);
      }
    }
}

static void ihave_rsr(nexus_endpoint_t *endpoint, globalpointer_t *reply_to)
{
  doc_request_t *user_ptr=nexus_endpoint_get_user_pointer(endpoint);

  if (!user_ptr->sent) {
    user_ptr->sent=TRUE;
    call_senddoc_rsr(reply_to, &user_ptr->browser_gp, user_ptr->url);
    }
}
```

Figure 12: Personal Cache (2)

written in Java and communications between clients and the display server use Nexus. This facilitates the programming of graphical applications that run on servers and that display information directly in browsers. In particular, agent applications [35] that are developed the Multimedia Research Group at the University of Southampton intercept WWW requests, and asynchronously return more sophisticated information, in the hope that it will be useful to the user. The Java graphical layer, combined with the Nexus communication, provide again the same uniform interface to the programmer.

Browsers, with their Java Virtual Machine, have become ubiquitous platforms of computing [2, 16]. An interesting challenge is to program applications that are distributed across several browsers. However, security restrictions usually only allow applets to establish connections with the server they were downloaded from. At an abstract level, our uniform model of computing provides peer-to-peer communications facilities via global pointers. In terms of implementation, messages are automatically routed via HTTP servers by our tunnelling layer. Our architecture to programming the WWW allows us to install intermediaries that forward messages to browsers, and the solution described in [2] is readily implemented in our system. In the related work section, we address the issue of firewalls and discuss a mechanism for programming routing of remote service requests, which would even further generalise [2].

4 Implementation

4.1 Tunnelling of RSRs

Nexus provides a simple asynchronous communication abstract via the RSR mechanism. Nexus has a flexible communications model which is built upon a number of protocol modules. These modules provide a common interface to a number of different communication mechanisms, such as TCP, UDP and Shared Memory. This enables Nexus processes to communicate by the most effective means available between them. For example, Nexus may have a protocol module for the high performance communications backbone found on a parallel machine; any Nexus application run on the machine would take advantage of the faster messaging. The model also provides for heterogeneity in the communications protocols, by allowing a number of different protocol modules with the library. So for example, an application may consist of a number of processes running on a supercomputer and a remote process running on a user's workstation. The processes running on the supercomputer can communicate between themselves using the high performance protocol, but when communicating with a workstation, they would use a different protocol, such as TCP.

Nexus requires that delivery of messages is both reliable and preserves order. The asynchronous nature of the RSR means that two processes need not maintain a connection to communicate. As a result, communications can be easily mapped onto HTTP and CGI-BIN mechanisms. We have implemented an HTTP protocol module for Nexus. Hence, when a Nexus processes wishes to send a message to another, via HTTP, it packages up the message into a form, and makes a HTTP request to the webserver. This request launches a CGI-BIN script which decodes the form and passes the block of data to the destination Nexus process. The binary data is encoded using `base64` encoding. Figure 13 shows this process.

For this operation to occur, two pieces of information are required. The sending Nexus process needs to know which webserver to contact, and the CGI-BIN script needs to know where to pass the information onto. Both of these pieces of information are encoded in the global pointer. A global pointer contains a block of data that contains information about how to communicate with the process that the global pointer points to. This block of data is generated

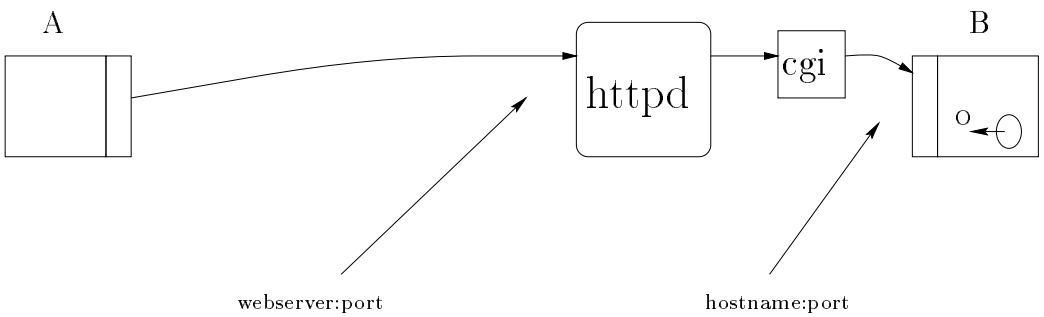

Figure 13: The HTTP protocol module for Nexus.

by querying each of the protocol modules that the Nexus process is using. Each module encodes in an array of bytes the information required to communicate with that process. When a global pointer is passed to a process, the communicated data is extracted from the the global pointer and a string of each bytes is sent to each communication protocol. The protocol module then decodes the data and recreates the relevant information. In the case of TCP and UDP, the data consists of a hostname and port number on which the source process is listening. In the case of HTTP, we include the `host:port` of the webserver as well as the `host:port` of the source process. For example, in Figure 13, process B created a global pointer to a local object. The HTTP protocol module encoded `webserver:port` and `hostname:port` in the global pointer. Let us assume that this global pointer was passed to process A, which extracted the protocol information. When process A performs an RSR on the global pointer, its HTTP protocol module connects to the correct webserver for process B and specifies the `hostname:port` that process B's HTTP protocol module is listening on.

On the receiving end, the protocol module awaits for connections on a port. When the CGI-BIN script is called, it decodes the `base64` encoding of the Nexus message, connects to the relevant `host:port` and writes the Nexus message. We chose to decode the message in the CGI script because we wanted to minimise the time spent by the protocol module in a critical section. The receiving process also needs to know how much data it should expect. This information is sent by the CGI script as a header to the actual data, and is another field in our form. Figure 14 shows the complete HTTP request of a Nexus message. The CGI-BIN script is written in Perl and uses the `cgi-lib` package to extract the fields in the form.

In order to preserve order of messages, the protocol module maintains an outgoing queue for each destination `host:port`. A new message is not sent until the process receives an indication from the CGI-BIN script that the previous message was delivered to the Nexus process.

Although we have used HTTP as our transport layer, the textual representation of Nexus message and asynchronous communications enable the use of other transport mechanisms, such as Email. Using Email as a transport layer would allow Nexus processes to communicate a much wider variety of environments, and in situations where a process or machine is not permanently attached to a network. The Email transport layer is both connection-less and unreliable, so some form of sequence numbering would have to be used to maintain reliable delivery of messages in order.

```
POST /cgi-bin/nexus-send.pl HTTP/1.0
Content-type: multipart/form-data; boundary=----PPP
Content-Length: 478

------PPP
Content-Disposition: form-data; name="host"

roobarb.ecs.soton.ac.uk:1107
------PPP
Content-Disposition: form-data; name="size"

184
------PPP
Content-Disposition: form-data; name="data"

ALgAAAAEUPb/vwEAAAAAAAEAAAABAAAAFwAAAHJvb2JhcmIuZWNzLnNvdG9uLm
FjLnVrAQAAAAAAAAACAAAAAQEABAAAALTVBwhmAAAAAAAAAQAAFgpyb29iYXJi
LmVjcy5zb3Rvbi5hYy51awAADAAoAAAEVXJvb2JhcmIuZWNzLnNvdG9uLmFjLn
VrAAAAAFByb29iYXJiAAABAAUAAARXAAAJAA0AAAAIAACzBAAAAAAAAAAAAA==
------PPP--
```

Figure 14: HTTP form data.

4.2 Server and Proxy Implementation

In this section, we describe the implementation of our Wexus-aware webserver. The server must respond to HTTP requests as well as Nexus RSR calls. The Nexus library provides a standard interface to threads, mutexs and callbacks. Nexus is optimised for fast communications and has a number of functions for registering threaded callbacks on I/O operations. Using these facilities, we are able to build a high performance server.

Firstly, the server establishes the port on which it will service HTTP requests. Nexus provides a facility to listen on a port and to call a function when a connect is received. This function reads the HTTP request and parses the URL from it. If there is no registered global pointer for synchronous computation, then the document requested must be served. If the document is local to the machine, then the server can access the file directly, but in the case of the server acting as a proxy, then we use libWWW to get the specified document. If there is a registered global pointer, then a Nexus buffer is constructed containing the path component of the URL, followed by key and value pairs if these were present in the URL. A data structure containing the file descriptor of the socket to the browser and a global pointer pointing to it are created. This global pointer is put in the buffer and the buffer is sent to the registered get global pointer. This buffer is also sent to all the global pointers that were registered using the define-notify-callback RSR which are stored in an array.

The server registers the three RSR handlers that perform operations on sockets to the browsers. These handlers use the given endpoint to find the local data structure. The operations that these handlers perform are very simple; namely write header information, write data and close socket. The close operation also disposes of the datastructure and endpoint.

The server must respond to handlers for registering callbacks; namely for the get and notify callbacks. In the case of get this simply copies the global pointer because there can only be on callback registered for get. For notify we have an array of global pointers, and so the handler

simple adds the supplied global pointer to the array. To improve performance and flexibility, we intend investigating the extension of the registering process to include a regular expression for those URLs of interest.

The other operation that the server must perform is to convert string representations to actual global pointers. This facility enables browsers to specify global pointers in the arguments of the request. A mapping must be registered in the webserver which takes a string and returns a global pointer. The mapping in the server is stored in a hash table for fast lookup. Two handlers to define and undefine strings manage the table. The strings are mapped to global pointers when the server parses the URL and comes across a `gp=string` key value pair. In the case where the URL contains such a pair, the server uses the global pointer of the last pair to send a `get` to.

5 Related Work

Java appears to many as *the* language for programming the WWW and indeed has strong credentials in its favour. The applet mechanism provides transparent execution of Java code in browsers and the byte-coded architecture is platform independent; combined with RMI, it also offers a platform for distributed computing, which also contains a distributed garbage collector as in our approach.

First, let us also mention that Java is not only the language to provide transparent code execution when WWW pages are downloaded: for instance, CAML and its camlets [37] offer similar facilities, but the widespread usage of the Java virtual machine makes Java the language of choice. More importantly, we believe that Java and RMI do not provide a satisfactory answer to our problem of programming the WWW. *(i)* We prefer a language independent programming paradigm, which gives us the freedom to choose the language most adapted to our use. *(ii)* Remote method invocation is a two-way kind of communication, where a request is expected to be followed by an answer in the opposite direction; remote service requests, like communication channels, are one-way operations, which are more efficient in some situations (cf. Figure 3): if the receiver decides to delegate the handling of a message, to a third process, the latter can answer directly to the emitter, hereby short-cutting the receiver. *(iii)* Java and RMI do not provide the routing mechanism described in this paper, even though Java method invocations can also be tunnelled into HTTP. *(iv)* Java and RMI do not provide any callback mechanism to interact with the flow of information of WWW transactions.

Other groups have integrated distributed object approaches with the WWW. Amongst others, CorbaWeb [27] provides gateway between HTTP servers and CORBA objects, which allows browsers to navigate through CORBA object links using dynamically generated URLs for each object. They do not however provide a uniform programming approach as we do; furthermore, we believe that our solution is bound to be lightweight because it relies on Nexus which was designed for high performance computing. W3objects [22] is an object-based WWW infrastructure that provides support for naming, sharing, mobility, and referencing. Wexus is lower-level abtraction than W3objects because it is only concerned with starting remote computations, communications, and interaction with WWW transactions.

Wexus provides mechanisms for coordinating distributed computations, and as such can be compared with coordination architectures such as PageSpace [10]. The PageSpace architecture has broadly similar goals to Wexus and is also asynchronous and decentralised; it emphasises interactive applications. It is based on a library which enhances Java with the Linda coordination model.

Other languages for programming the World Wide Web have been designed but they have different goals. Cardelli and Davies' WWW combinators [8] and Kistler and Marais [25] WebL share a common goal: they provide a set of constructs that can be used to program applications that emulate the behaviour of a user interacting with his/her browser.

Barret and Maglio [3] reach similar conclusions to ours: they observe that HTTP is not always well suited for all types of applications, in particular wireless links. They use their intermediaries to provide a conversion to and from other protocols. Our approach generalises theirs because we adopt a general, language-independent model of distributed computing, which abstracts from the communication layer. We can then easily convert between protocols, and let the transport layer automatically choose the most suitable protocol available at the time of execution. Intermediaries [3] also provide a programming interface that seems rather orthogonal and complementary to ours; we therefore see no difficulty in implementing it in our framework.

Cardelli and Gordon's Mobile Ambients [9], and Vitek and Castagna's seal-calculus [42] are two calculi for explicitly modelling and reasoning about firewalls and mobile agents. On the other hand, our uniform approach to programming abstracts away from the heterogeneity issued from firewalls. In fact, we regard the two approaches as complementary: firewall-related calculi are useful to model the interaction, negotiation, authentication between agents and firewalls. Once this phase has been completed, our abstract model provides a convenient abstraction that hides the routing that may have to take place during each communication.

This discussion about firewalls naturally lead to the general question of security. Our primary goal was to develop societies of agents interacting on the WWW. Therefore, we have hardly addressed the issue of security as we run our processes on trusted hosts inside a single domain. However, we plan to take this important issue into account. Again, our assumption is that the communication layer can already provide important security properties. Nexus, is in fact, part of a more general system, called Globus [20], which aims at providing resource management and negotiation, trusted host authentication, and encryption. We intend to make use of these services in order to provide dynamic and secure configurability of the system. In addition, Abadi's [1] secure tunnelling would be an interesting route to investigate to allow our computations to cross firewalls.

6 Conclusions

Experience in building a number of distributed Web applications has established clear requirements for a model supporting arbitrary peer-to-peer communication and ease of coordination of distributed computations. The Wexus model was designed to address these needs, through remote service requests, global pointers, HTTP integration and a reference counting mechanism.

In this paper we have presented the model and described an implementation based on Nexus. The implementation provides a language-independent layer and we have presented examples in two programming languages. We believe the model will extend naturally to the larger distributed Web applications which motivated this work, and we plan to investigate its application as an infrastructure for agents in complex applications and in support of collaborative applications.

7 Acknlowedgments

This research was supported in part by the Engineering and Physical Sciences Research Council, grants GR/K30773 and GR/K73060, and by the Joint Information Systems Committee, grant

Bibliography

[1] Martin Abadi, Andrew Birrell, Raymie Stata, and Edward Wobber. Secure web tunneling. In *Proceedings of the 7th International World Wide Web Conference*, volume Computer Networks and ISDN Systems 30, pages 531–539, 1998.

[2] A. Baratloo, M. Karaul, H. Karl, and Z. M. Kedem. An infrastructure for network computing with java applets. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, February 1998. Available at `http://www.cs.ucsb.edu/conferences/java98/program.html`.

[3] Rob Barrett and Paul P. Maglio. Intermediaries: new places for produicing and manipulating web content. In *Proceedings of the 7th International World Wide Web Conference*, volume Computer Networks and ISDN Systems 30, pages 259–270, 1998.

[4] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[5] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. *Software–Practice and Experience*, 25(S4):87–130, December 1995. Also available as Digital Systems Research Center Research Report 115.

[6] C. Brooks, M. Mazer, S. Meeks, and J. Miller. Application-Specific Proxy Servers as HTTP Stream Transducers. In *Proceedings of the 4th International World Wide Web Conference*, volume World Wide Web Journal 1 (1), pages 539–548, 1995.

[7] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995. Also available as Digital Systems Research Center Research Report 122.

[8] Luca Cardelli and R. Davies. Service Combinators for Web Computing. In *Usenix Conference on Domain Specific Languages DSL97*, Santa-Barbara, California, October 1997.

[9] Luca Cardelli and Andrew Gordon. Mobile ambients. In *Foundations of Software Science and Computational Structures (ETAPS'98)*, volume 1378 of *lecture notes in computer Science*, pages 140–155, May 1998.

[10] P. Ciancarini, A. Knoche, R. Tolksdorf, and F. Vitali. PageSpace: An Architecture to Coordinate Distributed Applications on the Web. In *Proceedings of the 5th International World Wide Web Conference*, volume Computer Networks and ISDN Systems 28 (7–11), pages 941–952, 1996.

[11] Jonathan Dale. *A Mobile Agent Architecture for Distributed Information Management*. PhD thesis, University of Southampton, January 1998.

[12] Hugh Davis, Wendy Hall, Ian Heath, Gary Hill, and Rob Wilkins. Towards an Integrated Environment with Open Hypermedia System. In *Proceeding of the Second European Conference of Hypertext (ECHT'92)*, pages 181–190, 1992.

[13] David DeRoure, Les Carr, Wendy Hall, and Gary Hill. A Distributed Hypermedia Link Service. In *Third International Workshop on Services in Distributed and Networked Environments (SDNE'96)*, pages 156–161, Macao, June 1996.

[14] Ian Foster, Carl Kesselman, and Steven Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing*, 37:70–82, 1996.

[15] Ian Foster and Robert Olson. A Guide to Parallel and Distributed Programming in nPerl. Mathematics and Computer Science Division, October 1995. http://www.mcs.anl.gov/Projects/nexus/nperl/.

[16] Ian Foster and Steve Tuecke. Enabling technologies for web-based ubiquitous supercomputing. In *Proc. 5th IEEE Symp. on High Performance Distributed Computing*, 1997.

[17] Ian T. Foster, David R. Kohr, Robert Olson, Steven Tuecke, and Ming Q. Xu. Point-to-Point Communication Using Migrating Ports. In *Languages, Compilers, and Run-time Systems for Scalable Computers*, pages 199–212. Kluwer Academic Publishers, 1995.

[18] Al. Geist and al. PVM 3 User's Guide and Reference Manual. Technical report, Oak Ridge National Laboratory, Knoxville, Tennessee, May 1993.

[19] Al Geist, William Gropp, et al. MPI-2: Extending the message-passing interface. In *Second International Europar Conference (EURO-PAR'96)*, number 1123 in Lecture Notes in Computer Science, pages 128–135, Lyon, France, August 1996. Springer-Verlag.

[20] The Globus Home Page. `http://www.mcs.anl.gov/globus/`.

[21] Barry Hayes. Finalization in the Collector Interface. In *Proc. 1992 International Workshop on Memory Management*, pages 277–298, Saint-Malo (France), September 1992. Springer-Verlag.

[22] D. B. Ingham, M. C. Little, S. J. Caughey, and S. K. Shrivastava. W3objects: Bringing object-oriented technology to the web. In *Proc. Fourth International World-Wide Web Conference*, pages 89–105, Boston, Mass., USA, December 1995.

[23] Inmos. *Occam^{TM} Programming Manual*. Prentice-Hall, 1984.

[24] A. Kind and J.A. Padget. Multi-lingual threading. In *Proceedings of 6th Euromicro Workshop on Parallel and Distributed Processing*, pages 431–437. IEEE, IEEE Computer Society, 1998.

[25] Thomas Kistler and Hannes Marais. Webl — a programming language for the web. In *Proceedings of the 7th International World Wide Web Conference*, volume Computer Networks and ISDN Systems 30, pages 259–270, 1998.

[26] Bernard Lang and François Rouaix. The V6 Engine. In *Programming the Web - a Search for APIs*, May 1996. Available at `http://pauillac.inria.fr/~lang/Papers/v6/`.

[27] Philippe Merle, Christophe Gransart, and Jean-Marc Geib. CorbaWeb : A Navigator for CORBA Objects. *Dr. Dobb's Sourcebook Distributed Objects*, pages 1–29, Jan/Feb 1997.

[28] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes (part I and II). LFCS Report Series ECS-LFCS-89-85 and ECS-LFCS-89-86, LFCS, Department of Computer Science, University of Edinburgh, 1989.

[29] Luc Moreau. Hierarchical distributed reference counting. Technical report, University of Southampton, May 1998. Submitted for publication.

[30] Luc Moreau, David DeRoure, and Ian Foster. NeXeme: a Distributed Scheme Based on Nexus. In *Third International Europar Conference (EURO-PAR'97)*, volume 1300 of *Lecture Notes in Computer Science*, pages 581–590, Passau, Germany, August 1997. Springer-Verlag.

[31] Luc Moreau and Nicholas Gray. A Community of Agents Maintaining Links in the World Wide Web (Preliminary Report). In *The Third International Conference and Exhibition on The Practical Application of Intelligent Agents and Multi-Agents*, pages 221–235, London, UK, March 1998. Also available at `http://www.ecs.soton.ac.uk/~lavm/papers/gcWWW.ps`.

[32] The Nexus Home Page. `http://www.mcs.anl.gov/nexus/`.

[33] Henrik Frystyk Nielsen. Libwww - the w3c sample code library. `http://www.w3c.org/Library/`, 1997.

[34] Albert Peck. Remote windowing system. Third year project, University of Southampton, May 1998.

[35] Aggelos Pikrakis, Tilemahos Bitsikas, Stelios Sfakianakis, Mike Hatzopoulos, Dave DeRoure, Wendy Hall, Sigi Reich, Gary Hill, and Mark Stairmand. Memoir — software agents for finding similar users by trails. In *The Third International Conference and Exhibition on The Practical Application of Intelligent Agents and Multi-Agents (PAAM'98)*, London, UK, March 1998.

[36] Jonathan Rees and William Clinger. Revised^4 Report on the Algorithmic Language Scheme. *Lisp Pointers*, 4(3):1–55, July-September 1991.

[37] Francois Rouaix. A web navigator with applets in caml. In *Proceeding of the fifth World Wide Web conference*, 1996.

[38] Jon Siegel. *CORBA fundamentals and programming*. Wiley, 1996.

[39] Squid internet object cache. `http://squid.nlanr.net`.

[40] Sun MicroSystems. Java Remote Method Invocation Specification, November 1996.

[41] Gerard Tel and Friedemann Mattern. The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes. *ACM Transactions on Programming Languages and Systems*, 15(1):1–35, January 1993.

[42] Jan Vitek and G. Castagna. Towards a calculus of secure mobile computations. In *Workshop on Internet Programming Languages (WIPL'98)*, May 1998.