Web&: An Architecture for Non-Interactive Web

S. H. Phatak, V. Esakki, B. R. Badrinath and L. Iftode
Department of Computer Science
Busch Campus, Rutgers University
Frelinghuysen Road
Piscataway, NJ 08855
e-mail:{phatak,esakki,badri,iftode}@cs.rutgers.edu

Contents

Abstract

As the use of the world wide web grows, the profile of web use has changed significantly. Web use has gone from purely information access (soft data) to critical operations such as e-commerce (hard data). However, the synchronous model which the world wide web currently supports is time consuming. Thus web browsing can easily become a frustrating experience if the network or servers incur delays, are not accessible, or the same interaction must be repeated many times. This model is even more painful for clients who are disconnected most of the time or are accessing the web via low bandwidth connections and resource constrained devices such as PDAs. Thus, there is a legitimate need to support non-interactive, asynchronous transactions between client and servers on the web.

In this paper, we propose a novel non-interactive service architecture for the web called Web&. The architecture incorporates server and service discovery, support for disconnected and heterogenous clients, web transactions via a uniform server interface, and persistent client state.

We also present a prototype that we have implemented using JAVA, XML and JDBC. The prototype consists of a client proxy, server proxies and a directory structure. The client proxy stores the client state and performs web queries on the client's behalf. Server proxies provide a uniform interface to servers on the web based on XML. The directory service allows clients to discover servers for a service and enforces a standard protocol for communication between the client proxy and the server proxies.

The benefits of asynchronous interaction include: concurrent activities, client mobility, device independence, reduced sensitivity to network/server latencies/unavailability. These lead to a more efficient interaction with the web.

1 Introduction

The number of users using the world wide web has seen exponential growth in the last few years. As this number grows, the profile of such users is changing from home users surfing the web at leisure to mobile users who are always under time pressure when accessing the web. For these people, interactive web is often painful because it consumes time and needs attention. What they would like to have instead is a non-interactive architecture that accepts queries, finds the appropriate servers and performs web tasks without user intervention.

In addition, more and more potential users are using devices that are not always connected to the web, such as laptops and PDAs. Such users often need access to the web, but are forced to wait until they can connect to the web either through dial up lines or through fixed terminals. This is unfortunate since many of the tasks that such a user may perform could often be performed without the users intervention.

As the profile of users is changing, the use of web data is also changing. In the past much of web use was related to recreational or soft data. Today much of web use has migrated to hard or critical data. This is exemplified by the growing popularity of e-commerce sites. Often interactive web browsing involves long waits (hence the acronym world wide wait) and needlessly repetitive tasks. As data becomes more and more critical users are forced to spend more and more fruitless time in such non-productive tasks.

Consider for example, a user who wants to buy tickets for a baseball game from an e-commerce site on the net. A variety of circumstances can lead to a frustrating web experience:

the site might be inaccessible or slow; the user might be using a PDA or other device that is incapable of rendering the site's home page; the tickets may not be available until a later date/time, which may be unknown at present; the user may not want to be online until a window of opportunity to actually complete the purchase manifests itself. The question is how to automate such a web transaction to the point where it can be performed when the user is offline.

In this paper we propose a new architecture that allows users to perform tasks on the world wide web in a non-interactive fashion. This architecture addresses the problems created by interactive web browsing. Our goal is to provide non-interactive users with an online experience and to relieve the frustration of interactive users performing repetitive tasks. We achieve this by interposing an intermediate layer of proxies between the client and server that allows us to replace the synchronous client server interaction with an asynchronous interaction between the client and this layer and moving the synchronous interaction between this layer and the server.

In the above example, this new layer of proxies would take over the task of actually purchasing the tickets. The user would no longer be required to control the interaction. Thus, the user can disconnect and reconnect at leisure to check the status of this task. The proxies are always online and can avail themselves of any window of opportunity to buy the tickets. Also this layer exports a standard interface which can be accessed by different types of clients.

2 The Non-Interactive Web Model

Traditionally, the web interaction model is synchronous (see Figure 1) which means that client and server have to be online for any web transaction to be performed. The client usually controls the transaction and keeps the state. In case the server is not accessible or fails, the client will repeat the query again possibly with a different server. Interactive web is easy to support but could be extremely inefficient when user wants to perform many different queries at the same time or repeat, condition or compose queries.

To alleviate some of the problems related to synchronous web interaction, we propose an asynchronous model for web interaction called **Web&**. Web& mediates between the client and the server and relieves the client of having to stay online to control the interaction with the server. In operating systems such as UNIX [10], analogous support for asynchronous jobs exists through the & background shell operator. However, in the context of the web support for executing tasks asynchronously, many web paradigms must be revisited. In the following discussion, we will briefly discuss the most significant changes required or facilitated by the non-interactive web.

Server and Service Discovery is absolute necessary in order to leverage the asynchronous web support. Once the user can send queries to be processed automatically at a later time, the scope of the queries can be extended from a server to a list of servers that provide similar service. For example, the user might specify that he/she wants to buy a given book at the cheapest price from any bookseller. To service such a request, the proxy must be able to discover all booksellers that provide a book buying service. The question is how to find the servers corresponding to that service. Furthermore, finding the list of

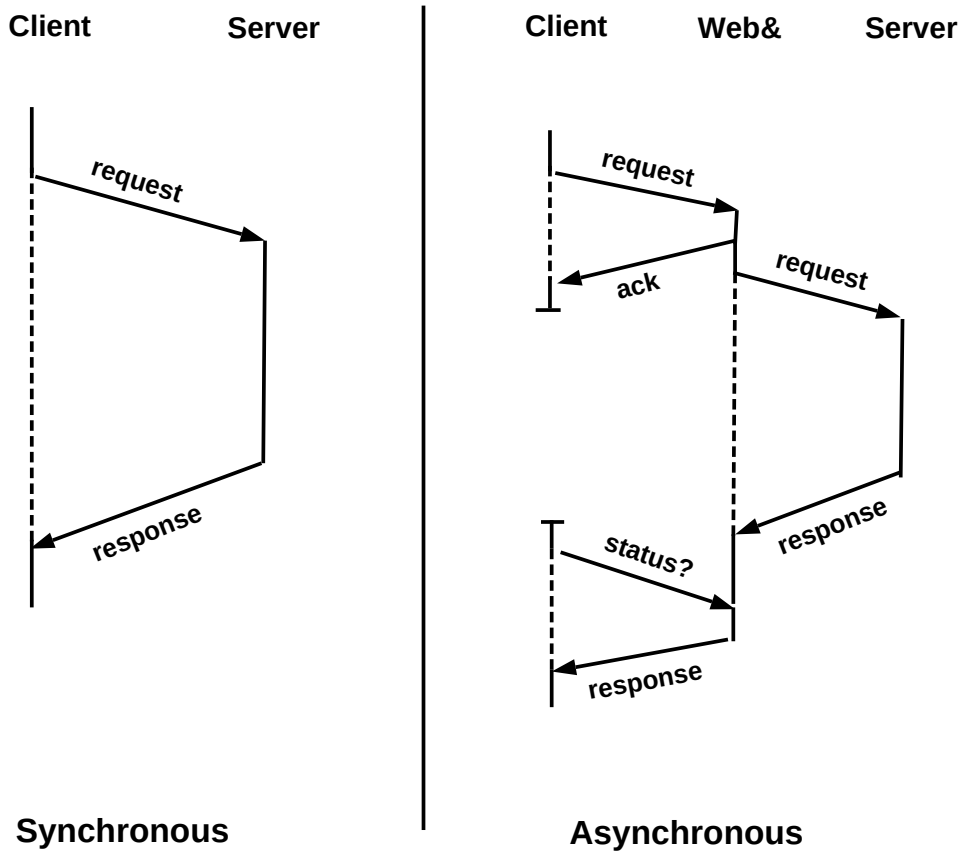

Figure 1: Client Server Interaction for Synchronous and Asynchronous Models

servers is not enough for automatic processing unless the servers can be accessed through the same interface. Today's synchronous nature of the interaction allowed servers to have distinct user-interfaces. In the future, standard interfaces to server will become an essential prerequisite for non-interactive web use.

The support for executing web tasks asynchronously also facilitates the use of **heterogeneous and disconnected devices**, ranging from fully featured desktop machines to hand held mobile computing devices. In particular, the user might submit a web task from one type of device, disconnect and later complete the task (access results) using yet another device. For example, a client may issue a request for a flight reservation from a laptop and check the status using a Palm Pilot. To provide such clients with "online even when offline" experience, a mechanism that can accept requests from all kinds of client devices is required. Obviously, with this model, client state for such clients must be formulated and stored somewhere when the client is not online.

For users who want tasks to be performed in the future we need the capability to schedule tasks arbitrarily. (This is equivalent to running cron daemon on a UNIX system.) This is because the user might request access to web resources that are not available at the time at which the request is made. For example, the user might request purchase of Baseball tickets for a particular game when they go on sale at some point in the future. Alternatively the user might asked to be notified of an event on the web which might occur sometime in the future, e.g., the user might ask to be notified whenever a particular band plays in his/her area. However, **scheduling** in itself is also not enough for such a system to be useful. Often, while performing web based tasks the users have implicit notions of transactions, i.e. tasks which depend on each other in some critical fashion. For example, a user might want to buy tickets for the baseball game only if flight reservations to the city in which the game is being played are available. Hence, any non-interactive web service model must support **web transactions**. For example, while planning a trip a user would want to make a flight reservation and hotel reservation. If the flight reservation is successful but the hotel reservation is not, then the user might have to cancel the flight reservation. Thus, many tasks might be interdependent and this interdependence must somehow be captured by the system. Analogously in UNIX, the shell allows users to specify dependencies between tasks using ; (sequence), —— (OR) and && (AND) operators, though web transactions would have a richer set of operators.

Finally, **fault tolerance** is important because in a non-interactive model the user is unavailable to restore state. Thus the model must guarantee **persistence** of all user and transaction specific state. It must be able to recover from transient infrastructural failures.

3 The Web& Architecture

An overview of the Web& architecture is shown in Figure 2. The Web& architecture is composed of three major components. These components use a standard protocol and language to communicate among themselves and with the clients.

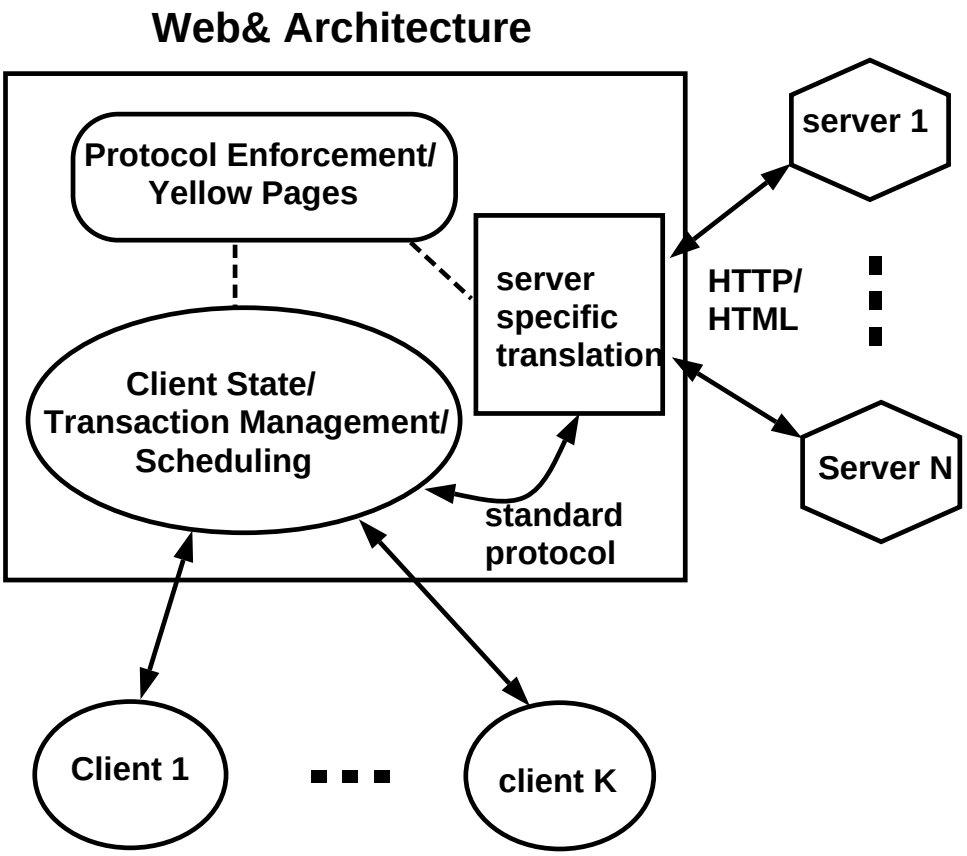

Figure 2: The Web& Architecture

3.1 The Client State and Transaction Manager

This is the key component of the non-interactive web model that is responsible for storing the state for all requests/transactions submitted by the users and managing user specified transactions. The client interacts with this component by submitting requests and requesting results for its requests. This component takes care of actually performing all the operations required to satisfy the request, enforcing semantics for all transactions submitted by the users, and taking care of scheduling the transactions and requests as per the constraints specified by the user(s). It also takes care of access protection for user/client state and authentication of clients.

3.2 Server Specific Translator

To perform a non-interactive web transaction, all servers providing a certain service must support the same interface. Since, in today's web, each server has a completely different interface, we need server specific translation to translate between this service-centric interface and the HTTP/HTML based interface exported by each web-server. From the point of view of the system all the servers export a superset of a standardized set of services. The protocol specifies how a client can interact with the servers and use the services.

3.3 Protocol Enforcer

The final component in our architecture is the entity that enforces the protocol and specifies the standardization of the services. This component provides a system wide yellow pages service that provides information about the servers and services in the system. The servers are grouped into overlapping domains, and each domain enforces a set of services that each server within the domain must provide. Additionally, for each required service in the domain, the domain also provides a service schema which specifies a uniform interface to access the service. Thus, the set of requirements provided by the domain includes a set of services each server in the domain must provide (though it may provide additional services) and a minimal service schema for each of the services in the domain, which must be supported by all servers but which may be extended by any server to support server specific service extensions.

In addition to the servers exporting services, the domains as well as the root of the directory are first class entities and export services of their own. These services are called "meta-services". Instead of operating on a single server such meta-services can potentially operate on multiple servers within the directory or the domain. Examples of specific services and meta-services can be found in the next section.

4 Prototype Architecture and Implementation

4.1 Prototype Architecture

Figure 3 illustrates in detail our Web& architecture with the following components: the root and domain servers and the directory, which provide directory wide meta-services, domain wide meta-services and protocol enforcement respectively; the client proxy, which provides

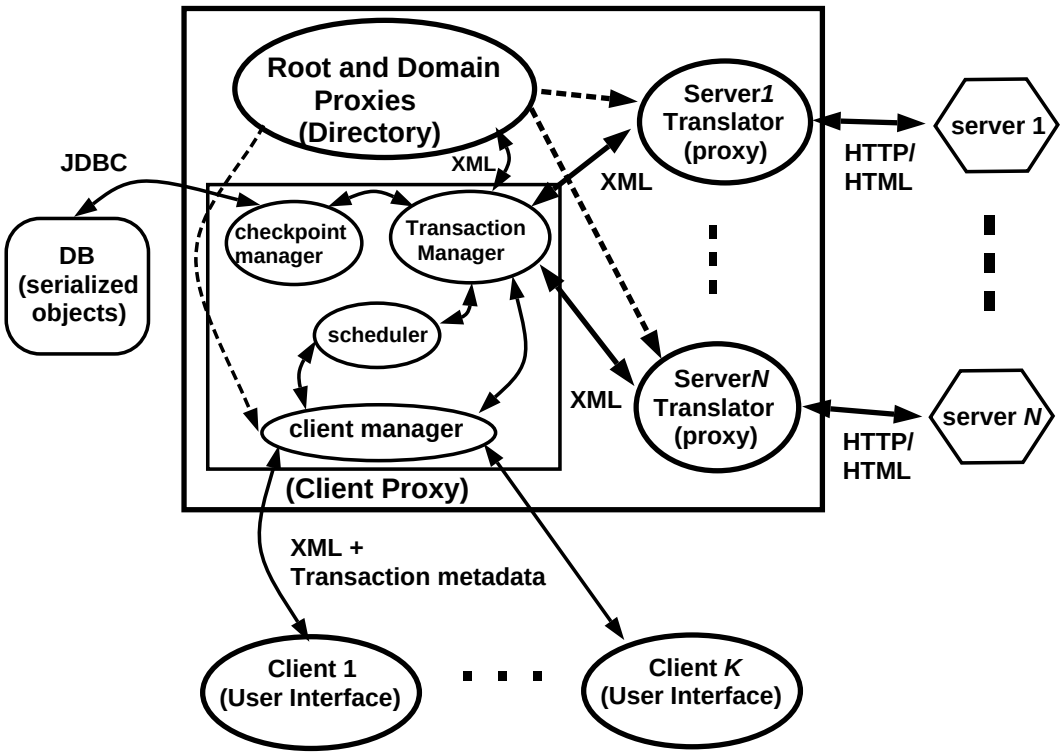

Figure 3: Prototype Architecture

client state and transaction management; the server proxies, which handle server specific translation; and a user interface. We discuss each of these components, their roles and the way they interact with other components, in the following sections.

4.1.1 The Root and Domain Servers

The servers for the directory root (the root server), and for each domain (domain servers) are necessary to provide directory wide and domain wide meta-services. These meta-services can be used to provide indexing services and search engine like services in the system. More importantly these servers export meta-services that are in effect the directory structure.

4.1.2 The Directory

The directory provides a generalized yellow pages service to the entire system. In particular it provides a list of domains served by the system, a list of services each domain provides, a list of servers providing service in the domain, and standard form templates (service schemas) that can be used by the clients to "talk" to the servers.

The "language" used by the clients to talk to each of the server proxies can be any standard language. As of now we use XML [17] both to express the service schemas in the Directory as well as to actually formulate the service requests. We call the XML service schemas *service form templates*. These templates must be "filled" by the clients to actually request a service. In order to manipulate these forms the server and client proxies first translate them into

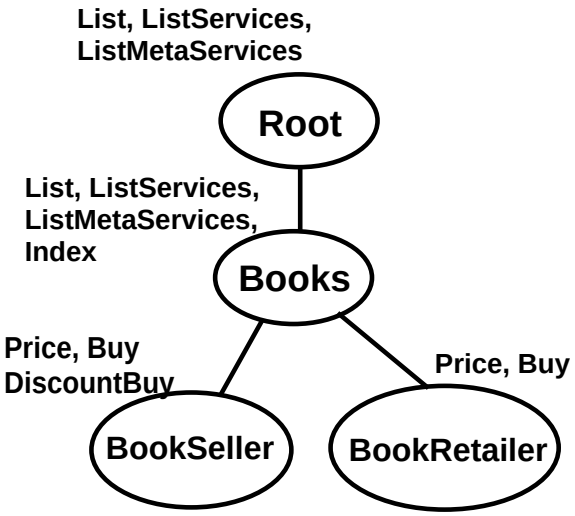

Figure 4: Directory Example

DOM [16], documents, operate on the DOM documents and then translate the DOM back into XML.

Each server on the web registers (advertises) with the directory, providing the basic list of services, and is listed under one or more domains (subdomains), based on what the server provides. By registering under a domain the server agrees to provide all services mandated by the domain and to honor the protocol/service schemas enforced by the directory for that domain. In addition the server may provide additional services and list them. The directory allows these special services as server specific services.

In our system, the directory structure actually exists as a set of services exported by root and domain servers. Both the root and each domain must at least export (i.e. implement) a minimal set of services which consists of the following services: *List*, *ListServices*, *GetInterface*. We call these services primary services or operations in our system. The *List* service returns a list of entities within the entity exporting the service: i.e. at root level it returns a list of domains, whereas at domain level it returns a list of servers in that domain. (The architecture can be easily extended to include sub-domains, but for simplicity we do not include them in this discussion). *ListServices* is a list of services that all constituent entities (i.e. those returned by *List*) must provide. *GetInterface* is a list of services that this entity provides. (While obtaining the service schemas is logically a part of *ListServices* as well as *GetInterface*, in practice we need an additional meta-service/operation *GetServiceSchema* to actually obtain the schema for using a particular service.)

These operations can be used to build the entire directory structure. Thus, a *List* on the root server returns a list of domains, whereas a *List* on each domain server returns a list of servers (or sub-domains) within a domain. The *ListServices* operations can be used to obtain the operations mandated by the root and each domain server. As a backup to the root and domain servers, our prototype also includes a web hosted directory.

For example, consider the simple directory structure in figure 2. Here, we have a directory with one domain, **Books**, which includes two servers: **BookSeller** and **BookRetailer**. Both the directory and **Books** export all the mandatory meta-services described before. In

	Root	**Books**	**BookSeller**	**BookRetailer**
List	{**Books**}	{**BookSeller**, **BookRetailer**}	{}	{}
ListServices	{*List*, *ListServices*, *GetInterface*}	{*Price*, *Buy*}	{}	{}
GetInterface	{*List*, *ListServices*, *GetInterface*}	{*List*, *ListServices*, *GetInterface*, *Index*}	{*List*, *ListServices*, *GetInterface*, *Price*, *Buy*, *DiscountBuy*}	{*List*, *ListServices*, *GetInterface*, *Price*, *Buy*}

Table 1: Operation Table for the Book Domain Example

addition to these meta-services, **Books** also exports an *Index* service. An index service allows the client to perform membership queries based on a key and a predicate. In our example, the *Index* service takes a key (i.e. title, isbn, etc.) and returns a list of servers that provide the products (books) that match this key. **Books** domain enforces two services on each server listed in it: *Price* and *Buy*, which allow the client to find out the price of (possibly many) book and buy a book, respectively. Both **BookSeller** and **BookRetailer** must provide these two services. In addition to these **BookSeller** also exports a service **DiscountBuy** which allows certain clients to buy books at discount prices. Table 1 describes the result of all the operations at each level of the directory structure.

This simple example illustrates the relationship between *ListServices* and *GetInterface* at each level of the directory structure (with level 0 being the root and level 2 the servers). A *GetInterface* operation on any entity at a given level k must return a superset of *ListServices* at level k-1. Also, at leaf nodes both *List* and *ListServices* return null sets since leaf nodes do not have any children, and these two operators operate on the child nodes of a given node. Further, note that support for *List*, *ListServices* and *GetInterface* is optional for the leaf servers. However, if a server wishes to export a server specific service (e.g. *DiscountBuy* in our example), it must at least support *GetInterface*.

With this design, aside from acting as a yellow pages service for the system, the directory is also capable of providing a powerful set of services. The client who contacts the directory can traverse from root, go to a domain, get a list of basic services available for that domain and a list of servers who provide those services. For each server, the client can then find a list of server specific services for that server. The client can also use directory wide indexing services to look for domains and servers providing a given service.

4.1.3 The Client Proxy

The client proxy is responsible for maintaining the state of a client. This allows the client to submit web transactions to the system and disconnect. The client proxy is responsible for executing the transactions on behalf of the client. On a fixed always connected machine, this proxy can co-exist with the user interface on the client itself.

The client proxy guarantees certain basic properties for user submitted transactions: it guarantees atomicity of the transactions, it ensures that results of the transactions are durable and that the transaction itself is preserved across failures. It provides checkpointing facilities to ensure that user state is never lost.

The client proxy consists of the following components:

- Transaction Manager: The transaction manager is responsible for maintaining and tracking transaction state. In particular it guarantees transactional properties for transactions in the system. An instance of the transaction manager is associated with each transaction in the system. The actual activation of the transaction manager instances is controlled by the transaction scheduler.

  For guaranteeing failure semantics of the transactions, the transaction manager uses the checkpoint manager to checkpoint state at the completion of each "basic" operation. These checkpoints are used to recover state whenever the client proxy restarts after a failure. The client manager also casts non-atomic web-operations into sub transactions, so as to make each operation (i.e. request for a service) atomic.

- Transaction Scheduler: The transactions are scheduled for processing by the transaction scheduler. Apart from supporting normal transactions which are scheduled based on the scheduling policy used in the transaction scheduler, the system also supports arbitrarily schedulable transactions which may be scheduled as per user requirements. Thus a user may specify a periodic transaction (say to inform him/her of the weather at the beginning of each day), specify a triggered transaction (buy tickets for a particular show when they become available), or specify a transaction to execute at a particular time instant in the future. The scheduler is responsible of invoking these transactions as per the parameters specified by the user.

- Checkpoint Manager: The checkpoint manager serializes transactional state to a back-end database. This manager is invoked whenever a transaction manager wishes to checkpoint state. Each checkpoint is guaranteed to have ACID properties since it is written in a single database transaction. These checkpoints can be used to provide the system with fault tolerance, as well as to provide the transaction manager with support to provide more complex transactional properties.

- Client Manager: This component of the client proxy is responsible for maintaining client specific state; receiving, authenticating and handling user requests; parsing user requests and generating transactions to submit to the transaction manager. This component is also responsible for protecting user state from unauthorized access by other users. Finally, the client manager exports a standard programmable interface to all clients. This can be used by the clients in conjunction to a form based interface to program in web based transactions. The client manager then parses these programs into transactions and a schedule that are submitted to the transaction manager and scheduler for execution.

4.1.4 The Server Proxy

The server proxy provides a bridge between the client proxies and the servers. Since servers do not "speak" the standard language mandated by the directory, this proxy provides a standardized interface to the client proxy, and performs on the fly translations between the standard forms used by the client proxy the HTML documents recognized by the servers.

The server proxies must recognize and translate all forms that are valid instances of the service schemas provided by the directory.

4.1.5 The Client User Interface

The user interface on the client is an interface that allows users to specify web transactions. These web transactions are then submitted to the client proxy for execution. The user interface itself is not particularly powerful and relies on the client proxy for most operations. This is particularly interesting because it enables thin clients to do complex processing on the web using the Web& architecture. The client could be a PDA, a cellphone or a two-way pager, connecting to the client proxy. Thus, a user, waiting in an airport, could submit queries from his PDA and request the results (which could possibly be huge) to be sent to a fax machine at the airport. Currently, we are also working on a web browser based interface to the system.

4.2 Implementation

Currently, we have a simple prototype implementation of all of these components in JAVA [5]. We use XML [17] as the language for the service form templates as well as the forms themselves. The form templates are simply XML forms with certain tags left empty. These tags must be "filled" before the form is accepted by the server proxy. The checkpointing mechanism uses JDBC [1] to connect to a backend database and serialize state information. The information is stored in form of CLOBs in a column in the database. Each time the client proxy is restarted the database is queried and state is restored. As of now we only have support for queries (read-only transactions), but we are rapidly working our way to transactions (read-write transactions).

The prototype itself uses the dynamic class loading facilities provided by JAVA to incorporate a certain amount of dynamic service specification. In particular, we are able to create new domains and deploy new servers without having to restart the proxy engine. We are investigating a web based interface to dynamically incorporate servers and services into the system.

5 Design Issues

5.1 Client Proxy Design Issues

The client proxy must implement a highly scalable architecture capable of supporting a large number of client requests. It must incorporate a standard interface that the clients can use to submit web transactions. It must be fault tolerant so that client state is never lost, and that transactional properties are preserved.

The client proxy implementation is particularly challenging since it is impossible to guarantee atomic operations on the web. Thus, standard database notions of logging and recovery do not apply directly. Instead, we must first make individual operations atomic and then use these to build our transactions, leading to a nested or multilevel transactional model [6, 12] or transaction workflows [6] like model. Further complicating matters is the fact that

most web operations can not be rolled back, rather they must be compensated for. To see this, consider a transaction in which a client wants to buy a ticket to a holiday locale and book a hotel room at that locale. This is a transaction since the non-completion of either purchase would render the entire transaction unusable. However, once an airline ticket has been bought or a hotel room booked, we can't simply roll back these operations, rather we must cancel them (compensate for their effects) potentially incurring a real monetary cost. Thus, we can't directly depend on a database backend to provide transactional support. We need to incorporate additional semantic structure into the system.

Supporting arbitrarily schedulable transactions is another challenging issue. There is a potential denial of service and fairness problem if a client tries to exploit the feature by specifying tight time constraints or by giving queries to be repeated very frequently (say every 30 secs). We could make the scheduler enforce some restrictions on these time parameters. The system should be either powerful enough to process such transactions or smart enough to identify such situations and restrict the client from doing so.

5.2 Server Proxy Design Issues

The server proxy must incorporate a standard interface to translate instances of the form templates provided by the directory to HTTP requests that servers can recognize and vice versa. Since a homepage on the web is often an evolving entity, we end up making changes to the proxy each time the web page changes. We are currently investigating mechanisms to compensate for this, including a scripting language for generating the proxy code. We hope XML will become the standard for information exchange in the web, in which case the servers could either speak XML and be accessible directly from the client proxy or take the responsibility of providing the XML based server proxy interface for their HTML/HTTP servers.

The other issue is to provide support for dynamic services and service specification. We are currently investigating ways to incorporate these from a fully service based architecture and through a web browser. We can already integrate new servers into the system using JAVA's dynamic loading functionality, we are currently extending the system to allow us to add servers through a web based interface and by using special services exported by the domains.

5.3 Directory Design Issues

The directory structure itself poses several issues. The directory must be easily accessible to potential clients preferably using standard protocols. The directory must incorporate mechanisms that allow dynamic addition of domains and services as well as servers exporting the services in a given domain. In addition form templates must be specified in a standard language such as XML to allow clients to communicate with the servers.

The exact organization of the servers into domains is an open problem. Many servers serve more than one logical domain. We handle this by multiply listing each server in each domain it can belong to. On the other hand there are servers which serve products that can't be easily classified into our domain based system. We are currently investigating ways of incorporating dynamic virtual domains into our architecture.

The final issue is that of semantic consistency. The directory currently specifies what each server must support, but does not really enforce this. Thus a server can potentially reject a request for a mandatory service. Thus, when a server/service is incorporated into a domain, we need a mechanism to perform conformance testing to ensure that all the servers in this domain export all the mandated services.

6 Related Work

Web& is designed to make the web experience of any user more productive. There is a large body of work on other approaches designed to make the web more accessible for a variety of devices and users.

Agents (for example, see [3]) have been used to provide adaptive web browsing support to enhance the capabilities of a web user. Agents have the capacity to learn user specific "meta-information" such as browsing habits, preferences, etc., and to tailor the user's web experience according to the information collected. Agents can also be used to execute complex tasks on behalf of a client. Our goals in Web& are significantly different from the agent approaches. In particular, Web& will perform exactly the actions specified by the user and does not attempt to modify the execution of a web task.

Active proxies [4] have been used to adapt web interaction for resource constrained devices. The proxy can filter and distill information to make it renderable on a specific device. In particular, such proxies can be used to remove images or reduce image complexity for PDAs and low bandwidth links and perform text filtering on a per device basis. The client still maintains control of the web interaction, the proxy only operates on the content. Web& is different in that we do not try to match the capability of each device, rather the heterogenity problem is solved by exporting a simple but standard interface to all clients and it is upto the client to decide how to render this interface. Furthermore, in case of Web& the client relinquishes control over web interaction and offloads this responsiblity to the client proxy.

Disconnected operation has been supported in a variety of databases (Microsoft's SQL Server, Sybase's SQL Anywhere and Oracle's OracleLite) as well as file systems such CODA [11]. In such systems, the client downloads enough state from the server to be able to operate even while it is disconnected. Web& is significantly different from these architectures. A Web& client is not responsible for any interaction, rather it only hosts the user interface. The responsibility of performing web tasks is with the client proxy which does not run on a disconnected machine.

Considerable work has also been done on service architectures, for example, JINI [15] and E-Speak [7]. These architectures provide support for clients to discover and access services provided by servers. Such architectures can be used as an underlying foundation to deploy a Web& implementation; however, Web& attempts to achieve far more than just service discovery. In particular, we are also interested in providing a transactional framework where services are the basic operations and to provide fault tolerance and persistance.

T-Spaces [18] provide a middleware architecture with database capabilities. This approach is complimentary to our approach in the sense that it addresses a number of orthogonal issues. In particular, TSpaces can be used to create a Web& implementation.

7 Conclusions

The web is growing exponentially, both in size and capability. Exploiting the resources provided by the web is becoming frustrating to do manually. We need an architecture to automate synchronous web tasks using a system which can operate on the web and perform such tasks on behalf of the user even when he/she is disconnected or otherwise unwilling to control the interaction. We achieve this by providing the blueprint for a novel non-interactive web architecture called Web&.

The Web& architecture includes a proxy engine which takes client's requests and executes them on the web, a directory that enforces a structure on the web and defines protocols for using it, a group of server proxies providing other components with a uniform interface of the web. We have provided a prototype implementation based on JAVA, JDBC and XML for each of these components.

The Web& architecture allows the user to execute web related tasks asynchronously and concurrently. As a result, Web& uniformly supports disconnected, mobile and heterogenous clients. Furthermore, since the system is asynchronous Web& is able to hide network/server latencies and failures from the user. Web& also provides composite services by supporting the notion of schedulable web transactions. All of this leads to a highly enriched and fruitful web experience for the user.

We anticipate that the deployment of Web& implementations shall make the web more accessible to a large variety of users. As the web migrates from soft data to hard data, architectures like Web& must become more pervasive to allow the web experience to remain fruitful. Furthermore, as more and more users take to using heterogenous devices, Web& shall have an important role in providing a flexible mode of web access to such users.

References

[1] The JDBC 1.2 specification. Technical report. http://java.sun.com/products/jdbc/.

[2] B. R. Badrinath and K. Ramamritham. Semantics-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems*, 17(1):163–199, 1992.

[3] A. Chavez and P. Maes. A real-life experiment in creating and agent marketplace. *Proceedings of the Second International Conference on the Practical Application of Intelligent Agents and Multiagent Technology*, April 1997.

[4] A. Fox, S. D. Gribble, T. Chawathe, and E. A. Brewer. Adapting to network and client variation using active proxies: Lessons and perspectives. *IEEE Communications*, August 1998.

[5] J. Gosling and H. McGilton. *The JAVA Language Environment*. Sun Microsystems. http://java.sun.com/docs/white/langenv/.

[6] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kauffmann, 1993.

[7] Hewlett Packard. *E-speak Architectural Specification*. http://www.e-speak.hp.com/pdf/archspec.pdf.

[8] IETF. *RFC 1777: Lightweight Directory Access Protocol*. http://www.ietf.org/rfc/rfc1777.txt.

[9] IETF. *RFC 2068: HTTP 1.1*. http://www.ietf.org/rfc/rfc2068.txt.

[10] D. M. Ritchie and K. Thompson. The unix time-sharing system. Technical report. http://cm.bell-labs.com/cm/cs/who/dmr/cacm.html.

[11] M. Satyanarayanan. Coda: A highly available file system for a distributed workstation environment. *Proceedings of the Second IEEE Workshop on Workstation Operating Systems*, pages 447–459, Sept. 1989.

[12] A. Silberschatz, H. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 1997.

[13] Sun Microsystems. *The JAVA Platform*. http://java.sun.com/docs/white/platform/javaplatformTOC.

[14] Sun Microsystems. *Jini Technology Architectural Overview*. http://www.sun.com/jini/whitepapers/architecture.html.

[15] Sun Microsystems. *Jini Technology Specification*. http://www.sun.com/jini/specs/.

[16] W3C. *W3C Recommendation DOM 1.0*, Oct 1998. http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/.

[17] W3C. *W3C Recommendation XML 1.0*, Feb 1998. http://www.w3.org/TR/1998/REC-xml-19980210.

[18] P. Wycroft, S. W. McLaughry, T. J. Lehman, and D. A. Ford. Tspaces. *IBM Systems Journal*, August 1998. http://www.research.ibm.com/journal/sj/373/wyckoff.html.