A Framework for Consistent, Replicated Web Objects

Anne-Marie Kermarrec
Ihor Kuz
Maarten van Steen (contact)
Andrew S. Tanenbaum

Internal Report IR-431

September 1997 (revised: March 1998)

In Proc. 18th Int'l Conf. on Distributed Computing Systems [1]

Abstract. Despite the extensive use of caching techniques, the Web is overloaded. While the caching techniques currently used help some, it would be better to use different caching and replication strategies for different Web pages, depending on their characteristics. We propose a framework in which such strategies can be devised independently per Web document.
A Web document is constructed as a worldwide, scalable distributed Web object. Depending on the coherence requirements for that document, the most appropriate caching or replication strategy can subsequently be implemented and encapsulated by the Web object. Coherence requirements are formulated from two different perspectives: that of the Web object, and that of clients using the Web object. We have developed a prototype in Java to demonstrate the feasibility of implementing different strategies for different Web objects.

Keywords: *distributed systems, object replication, object coherence, World-Wide Web, worldwide scalable systems*

vrije Universiteit

Faculty of Mathematics and Computer Science

1 Introduction

With the continuing growth of the Web's popularity, we are increasingly confronted with its limited scalability. In particular, problems are emerging with respect to accessing and transferring pages across the Internet. For example, Web servers are often unreachable due to their getting more requests than they can handle. Likewise, bandwidth limitations and unreliable links lead to long downloading times, especially with pages containing image, audio, or video. The solution to these problems is to apply traditional scaling techniques, such as caching and replication [10]. In the Web, much attention has been paid to caching. Recently, it has also been recognized that caching alone is not enough. In particular, replication techniques by which updates are pushed to clients are needed as well [2].

Unfortunately, caching and replication inherently lead to *consistency problems*: when a page is cached or replicated, a modification of one copy makes that copy different from the others. In Web caching [4], a simple consistency protocol has been widely adopted to manage consistency: whenever a page is retrieved from a cache, the cache checks when that page was last updated at its server. If the page was updated at the server subsequent to its being cached at the client, the cache entry is refreshed by fetching the page from the server. Otherwise, the currently cached version is handed over to the client. Provided the caching and update times are known correctly, this scheme never returns an outdated page.

A weaker form of consistency can also be maintained. For example, many Web proxies assume a page that has just been cached remains valid until some expiration time. Only when that time expires, will the cache entry need to be refreshed. Of course, in this scheme, it is possible that a cached page is stale and the client will be handed an out-of-date page.

To improve the Web's scalability, alternative caching and replication schemes have been proposed, each containing a solution for handling consistency problems. These schemes generally assume (1) that some form of consistency should be supported, and (2) that a single consistency model is required for *all* Web pages. No distinction is made between different pages, so the same caching strategy is applied to all of them. However, with the large variety of Web pages currently existing, and the increasing alternative applications of Web technology, it is questionable whether these assumptions are valid even now, let alone in the near future.

Consider, for example, a personal home page. In general, although it may be worthwhile to let an individual browser cache such a page, site-wide caching by a Web proxy is less likely to improve performance. In contrast, home pages of commonly popular organizations or events should perhaps be cached only by Web proxies. A browser's cache could then be reserved for pages that are popular only by a specific client. Further distinctions can be made with respect to deciding how modifications should be made visible. Irregular updates to pages for cultural or scientific events may require only a notification mechanism, after which an interested client may decide to pull the modification to a nearby cache. Magazine-like documents that are updated periodically, may benefit from a push strategy to servers in areas with a relatively large number of subscribers.

In our view, achieving scalability of the Web requires that we take a look at individual documents, where we consider a document as a collection of one or more pages. In particular, we first need to know what kind of coherence is actually required. Then, based on these coherence requirements, a suitable replication strategy should be applied. This approach requires that we look at each Web document separately. In other words, coherence *and* its implementation is to be considered strictly on a per-document basis. The current Web infrastructure is inadequate to support this approach. In particular, it provides hardly any facilities for adopting different strategies for separate classes of Web objects. As an alternative, we propose to use an object-based infrastructure, called Globe [6, 13], in which each Web document is modeled as a separate, self-contained object. Such an object encapsulates not only the content of one or more pages, but also implementations of policies for replicating and distributing those pages.

In this paper, we present a framework for scalable Web objects. The paper is organized as follows. First, we determine the coherence requirements for a Web object before applying caching and replication techniques. The coherence model of an object expresses the consistency of the object's state when multiple clients perform read and write operations on that state. In addition, we distinguish client-based coherence models, which express the consistency that a single client requires from an object. Coherence of Web objects is discussed in Section 3.

Web objects are expressed as distributed shared objects. These objects are physically distributed, meaning that they reside at multiple machines at the same time. In contrast to other object-based models, distributed shared objects fully encapsulate state, operations on that state, as well as implementations for caching, replicating, and migrating state. We discuss our object model in Section 2.

To demonstrate that we can indeed easily combine different models and implement them on a per-object basis, we have built a prototype in Java, which is briefly described in Section 4. We conclude and discuss future work in Section 5.

The main contribution of the research described in this paper, is that we show how scaling techniques, namely caching, replication, and distribution, can be effectively implemented for each Web document separately. As such, we demonstrate how truly scalable objects can be constructed, an issue that is not supported by the current Web infrastructure.

2 The Globe Approach

To support worldwide applications, we are currently developing a wide-area distributed system called Globe [6, 13]. Globe is constructed as a middleware layer on top of existing networks and operating systems. Our architecture consists of an object model and a collection of basic support services. In this section, we briefly discuss Globe's distributed-object model in the context of Web applications.

The general idea behind Globe, is that processes interact and communicate through **distributed shared objects**. Objects are passive, but multiple processes may simultaneously access the same object. When applying this model to Web applications, we model each Web document as a separate distributed shared object. A Web document consists of a collection of HTML pages, together with files for images, applets, etc., which jointly comprise the **state** of the distributed shared object. The state itself is hidden from clients behind one or more **interfaces**, each consisting of a number of methods. For example, an interface of a Web object consists of a method for selecting a page, and reading it in HTML format so that it can be subsequently interpreted by a browser. Likewise, we offer a method for replacing one of the document's pages.

Changes to an object's state made by one process are visible to the others. Our distributed shared objects are physically distributed, meaning that an object's state may be partitioned and replicated across multiple machines at the same time. However, client processes are not aware of this: state and operations on that state are completely encapsulated by the object. All implementation aspects, including communication protocols, replication strategies, and distribution and migration of state, are part of the object and are hidden behind its interfaces. In order for a process to invoke an object's method, it must first **bind** to that object by contacting it at one of the object's contact points. Binding results in an interface belonging to the object being placed in the client's address space, along with an implementation of that interface. Such an implementation is called a local object. This model is illustrated in Figure 1.

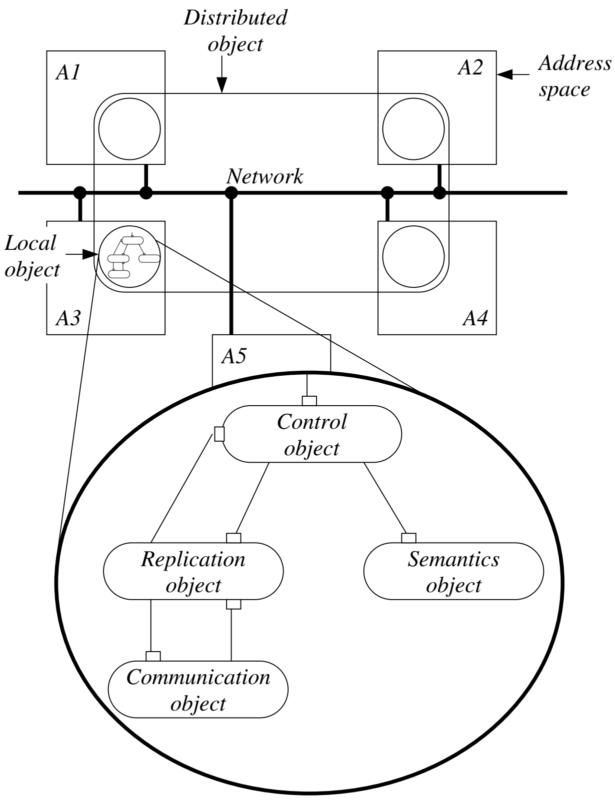

Figure 1: Example of an object distributed across four address spaces.

A **local object** resides in a single address space and communicates with local objects in other address spaces. It forms a particular implementation of an interface of the distributed object. For example, a local object of a distributed Web object may implement an interface by forwarding all method invocations to a central location where the files of the Web document are stored, as in RPC client stubs. However, a local object in another address space may implement that same interface through operations on local replicas of those files. Such implementation details are transparent to the client processes: they see only the interface to the distributed object as offered by the local object. Each local object is composed of several sub-objects, and is itself again fully self-contained as also shown in Figure 1. A minimal composition consists of the following four components.

Semantics object. This is a local object that implements (part of) the actual semantics of the distributed object. In the case of Web objects, the semantics object encapsulates the files that comprise the Web document. The developer is responsible only for the construction of those files, and encapsulating them into a semantics object with the appropriate interfaces. All other parts can either be obtained from libraries, or are generated from interface specifications.

Communication object. This is generally a system-provided local object. It is responsible for handling communication between parts of the distributed object that reside in different address spaces. Depending on what is needed from the other components, a communication object may offer primitives for point–to–point communication, multicast facilities, or both.

Replication object. The global state of the distributed object is made up of the state of its various semantics objects. Semantics objects may be replicated for reasons of fault tolerance or performance. In particular, the replication object is responsible for keeping these replicas consistent according to some (per-object) coherence strategy. Different distributed objects may have different replication objects, using different replication algorithms. A replication object is controlled by the control object.

Control object. The control object takes care of invocations from client processes, and controls the interaction between the semantics object and the replication object. Incoming invocation requests are also handled by the control object.

A key role, of course, is reserved for the replication object. An important observation is that communication and replication objects are unaware of the methods and state of the semantics object. Instead, both the communication object and the replication object operate only on invocation messages in which method identifiers and parameters have been encoded. This independence allows us to define standard interfaces for all replication objects and communication objects.

3 Replication and Coherence for Web Objects

In this section, we first introduce a system model that will allow us to express coherence for distributed objects that represent Web documents. We then present a collection of coherence models to express coherence requirements of current and future Web applications. Finally, we briefly present a taxonomy of strategies for implementing coherence.

3.1 A System Model

In Globe, a Web object encapsulates its own policies for replication and distribution of the files that comprise its state. In particular, each Web object will offer its own **coherence model** to client processes that bind to the object. To express such coherence models, we need an underlying system model that can capture the fact that the state of a Web object is replicated at different locations.

In our model, the files that comprise the state of a Web object can be kept at different **stores**. For simplicity, we distinguish only general **read** and **write** operations on these files, which are always performed by **clients**. Clients may perform read and write operations at any store. Three different classes of stores are distinguished:

Permanent stores implement persistence of a distributed Web object. This means that if there is currently no client process bound to the object, the object's state will be kept at its associated permanent stores. The permanent stores keep replicas consistent according to the coherence model that the Web object offers to its clients, as we describe in Section 3.2. A Web server is an example of a permanent store.

Object-initiated stores are installed as the result of the object's global replication policy. Replicas are kept consistent independent of clients although these stores may, for performance reasons, support a weaker coherence model than the one guaranteed by the object's permanent stores. A typical example of an object-initiated store is a mirrored Web site.

Client-initiated stores are comparable to caches. They are installed independent of the replication policy of the object, and fall under the regime of the client processes that read and write the object's state. A site-wide cache at a Web proxy is an example of a client-initiated store.

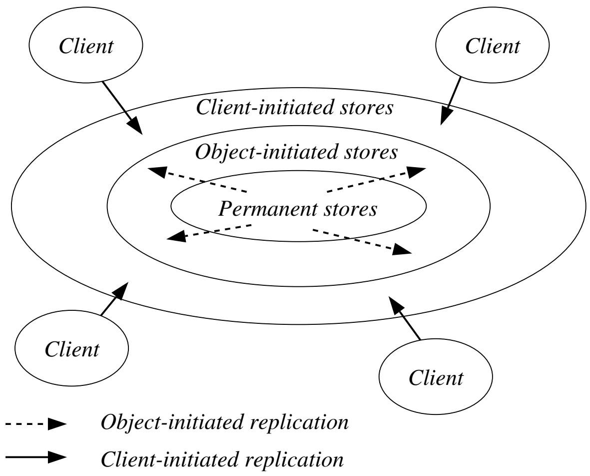

Figure 2: A system model for replicated Web objects.

Stores are organized in a layered fashion as shown in Figure 2. This architecture allows us to separate replicas managed by servers (permanent and object-initiated stores) from those managed by clients (client-initiated stores). Whereas permanent stores are responsible for implementing an object's coherence model, object-initiated and client-initiated stores may offer weaker coherence, but perhaps offering the benefit of higher performance. Effectively, for some applications, some delay in propagating a change is often acceptable. It is generally up to the client to decide to which replica he will bind.

3.2 Coherence Models

A coherence model describes the effect of read and write operations by different clients on a possibly replicated object, as viewed by clients of that object. A coherence protocol describes an implementation of such a model. Consequently, before describing coherence protocols, it is necessary to first consider the kind of coherence models applicable to Web objects. We make a distinction between coherence as offered by an object to its set of clients, and coherence as required by a client. These two classes of coherence models are discussed next.

3.2.1 Object-based Coherence Models

Current Web cache coherence protocols assume Web pages are modified only by their owner. They provide no support for shared applications that allow concurrent updates on shared state by different clients (like shared white-boards). However, we expect that such applications will become increasingly popular. Therefore, it is important that our framework offers models that express the coherence of an application shared by concurrently operating clients.

Many studies, especially in the distributed shared memory systems community have addressed the definition of such coherence models. We believe the following are relevant for current and future Web applications.

The sequential coherence model [8] requires a global ordering of operations on an object. Although such a coherence model is hard to implement efficiently, many applications will actually need it. For performance reasons, its implementation may be restricted to permanent stores, combined with only a simple propagation of updates to other store layers.

The PRAM coherence model [9] guarantees that writes made by a given client appear on every store in the order in which they have been issued. As this coherence model does not require global coordination neither between stores or clients, it can be implemented efficiently by tagging every update with the updater and a local sequence number. This model is particularly useful when clients update Web objects incrementally. Consider, for example, a shared bibliographic database. A client may decide to add a new record to the database, and later to update one of its fields. The PRAM coherence model prescribes that the field update at a store is delayed until the record has been added to that store's replica of the database. The **FIFO coherence model** is an optimization of the PRAM model. In this case, a write request from a client is honored if it is more recent than the latest write from that same client. Otherwise, the request is simply ignored. This model will prove better performance when clients overwrite a Web object instead of performing incremental updates.

The causal coherence model [1, 7] is also a weaker form of coherence since the ordering of operations must be guaranteed only between causally related operations. For example, such a coherence model could be applied to a Web forum, like a newsgroup, where a participant's reaction makes sense only if the audience has received the message that triggered the reaction. This ordering must be ensured at every store.

The eventual coherence model is the weakest form of coherence since it ensures that eventually updates are propagated but without any ordering constraints.

After deciding on a Web object's coherence model, a programmer selects a coherence protocol. The programmer also specifies which store layers have to support the model. For example, a groupware editor requires strong coherence at every store layer. On the other hand, a home page may require strong consistency only at permanent stores; it may be sufficient to support eventual consistency at the other two layers.

3.2.2 Client-based Coherence Models

A client-based coherence model allows a client to express his own coherence requirements. This is useful either when the object-based model is not sufficient from a client's point of view, or when no coherence is offered. Client-based coherence is applied separately for each client possibly in combination with the object-based coherence model.

Client-based coherence models are derived from the session guarantees developed in the Bayou system [12]. Bayou provides mobile users weak consistency support in a replicated database. We have retained their models, although there are a number of differences. First, we attempt to *guarantee* coherence rather than only check whether the coherence requirements are satisfied, as is done in the Bayou system. Second, we combine client-based models with object-based ones in a single framework.

The client-PRAM coherence model (equivalent to the Monotonic Writes scenario in Bayou) is the same as the object-based PRAM model but now restricted to only one client.

The client-causal coherence model (equivalent to the Writes Follow Reads scenario in Bayou) allows a client to view the execution of its operations with respect to those of other clients on which its own operations depend. The writes appear on every store in the same order respecting dependencies between operations. For example, if a client reacts to an electronic newspaper article, the article and then the reaction must appear in that order on every store to make any sense.

The Read Your Writes coherence model as defined in Bayou, is local to a client and to the store it accesses. An object is consistent according to this model if the effects of every write by a client are visible to all subsequent reads by that client. We return to this model in Section 4.

The Monotonic Reads Coherence model has also been defined in Bayou. The model prescribes that two subsequent reads, possibly at different stores, are based on copies updated by all writes preceding the first read operation. For example, consider a Web page replicated at two different stores S_1 and S_2. If a client first reads the page from S_1 and later again from S_2, then the second copy should be the same as the one read on S_1, or an updated version thereof, but not an earlier version.

The strength of our approach is that we can combine object-based and client-based models into a single framework. Object-based models represent a developer's view on consistent replication of an object; client-based models express the preferences of a single client. Moreover, we have separated the functionality of an object from consistency issues in the face of caching and replication, by isolating coherence protocols in replication subobjects. The standardized interfaces offered by our model allow us to dynamically update strategies. For example, when a client binds to a store and requests support for some client-based coherence model, the replication subobject of the store is easily augmented to integrate the implementation of the new coherence model. Not every combination of object-based and client-based model makes sense, however. For example, if the object offers sequential consistency, then it automatically offers every client-based model as well. On the other hand, if only PRAM consistency is offered, a client may decide to impose the Monotonic Reads model as well. An example of how these coherence models may be combined usefully is given in Section 4.

The flexibility offered by our approach need not always be experienced as a benefit to the casual user. Developers and clients of Web documents have to be aware of the effects of the various coherence models, which may not always be obvious. We plan to provide default solutions and simple guidelines to assist users where necessary.

3.3 Implementation Parameters

Which protocol should actually be used for a specific model may depend on such issues as read/write ratios, the number of clients simultaneously bound to an object, etc. We have defined a set of **implementation parameters** that are used to specify *when*, *how*, and *by whom* coherence is managed. An overview of the most common implementation parameters is shown in Table 1. These parameters must be set by the programmer of a Web object at initialization once the object-based coherence model has been chosen.

In addition to these common parameters, we have defined two other parameters tightly related to object-based and client-based models. They refer to the reaction of a store when it notices that coherence requirements for a given model are not satisfied. A store containing an outdated replica may either passively *wait* until an update arrives, or, alternatively, *demand* that its copy is immediately updated. A store's reaction to a copy that becomes outdated, is modeled by an **outdate reaction** parameter.

The choice of implementation is important since it may have a large effect on performance. For example, if a highly replicated Web object is often modified, it may be more efficient to implement a periodic update in which several updates are aggregated, instead of an immediate one. In contrast, if the Web object is seldom modified, then an immediate coherence transfer type avoids unnecessary network traffic. Ideally, the implementation parameters can be modified dynamically as the usage

Table 1: Implementation parameters for replication policies

Parameters	**Values**	**Meaning**
Consistency propagation	- update - invalidate	This parameter specifies how coherence is managed: either by updating or invalidating replicas when changes occur on an object.
Store	- permanent - permanent and object-initiated - all	This parameter specifies which kind of store implements the object-based coherence model.
Write set	- single - multiple	This parameter gives the number of simultaneously writers.
Transfer initiative	- pull - push	This parameter describes who is in charge of the propagation of coherence information: either coherence information is *pushed* to the replicas or they *pull* it from other replicas.
Transfer instant	- immediate - lazy (periodic or other criteria)	This parameter specifies when the coherence is managed: either as soon as a change occurs, or periodically whereby successive updates can be aggregated.
Access transfer type	- partial - full	This parameter specifies whether only part of the Web document or the entire document is retrieved when accessed.
Coherence transfer type	- notification - partial - full	This parameter specifies whether coherence is managed on only part of the Web document, or on the entire document. The notification value means that no invalidation or update is sent, but only a message to inform a store that a change occured.

characteristics of an object changes. However, self-adaptive policies are beyond the scope of this paper; they are a subject of future research.

4 An Example: Web Objects

To show how a Web object can be implemented as a distributed shared object and how the two levels of coherence can be integrated, we have implemented a prototype in Java, described in this section.

4.1 Combining PRAM and Read Your Writes Consistency Models

Consider a conference home page giving information about the technical program, registration, author guidelines, accommodations, and so on. The Web master of the conference incrementally updates the page when new information becomes available. Figure 3 depicts how the system is modeled. Using our terminology, the Web master of the page is represented by a client (client M). Interested participants are also modeled as clients (client U). The Web server where the page is stored is represented by a permanent store. Clients have caches modeled as client-initiated stores. Pages are cached on demand as usual. There are no object-initiated replicas in the system.

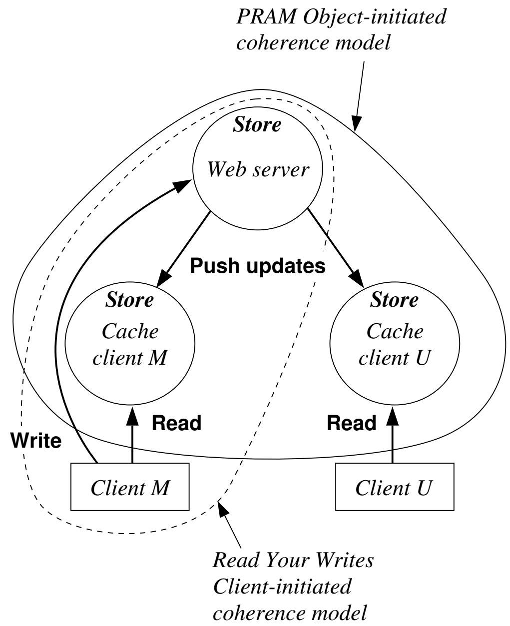

Figure 3: Overview of the example Web object's design.

As updates are performed incrementally and because we have a single permanent store, we have chosen to apply the *PRAM object-based coherence model*. The PRAM coherence model is applied to all replicas (store implementation parameter is thus set to *all*). The Web master writes directly to the Web server whereas all reads are performed from the cache. When the Web master updates the Web server, he must be able to check whether the write has been done correctly. For that purpose, the Web master uses a *read your writes (RYW) client-based coherence model*. As the PRAM object-based coherence model is sufficient for clients, no other client-based coherence model is specified in the system. The partial updates are pushed periodically to caches (transfer instant parameter set to *periodic*, transfer initiative parameter set to *push* and coherence transfer type parameter set to *partial*). When the Web server detects that PRAM coherence requirements have been violated, it simply waits until a new write arrives (object-outdate reaction parameter set to *wait*). However, if the cache M detects a violation in the RYW coherence requirements, it sends a request to the Web server to obtain the correct update (client-outdate reaction parameter set to *demand*). The implementation parameters are summarized in Table 2.

Table 2: Replication strategy parameter values for the example

Coherence propagation:	update
Store:	all
Write set:	single
Transfer initiative:	push
Transfer instant:	lazy (periodic)
Access transfer type:	full
Coherence transfer type:	partial
Object-outdate reaction:	wait
Client-outdate reaction:	demand

4.2 Globe Implementation

We have built a prototype in Java 1.1 on top of the Internet. We assume reliable communication but ordering is managed at the application level. We return to this below. The conference home page is implemented as a distributed shared object. Each client, or store, is represented as a local object and all entities have the same semantics object. The communication objects are all the same as well and implement point-to-point communication facilities (*send*, *receive* and *send/receive* functions). Figure 4 depicts the system implemented in Globe. The Web master and client applications are existing Web browsers.

Clients

The clients do not implement the semantics object. Basically, clients only translate method calls to messages which are sent to the caches (or server) to retrieve (or write) data.

Client U implements no coherence model at all. Read requests are immediately forwarded and executed.

Client M handles method calls from the Web master application. Its replication object implements PRAM object-based coherence as well as RYW client-based coherence. For PRAM coherence, a unique write identifier (*WiD*) is assigned to each new write, composed of the client's identifier and a sequence number ($WiD = (client\_id, sequence\_number)$). Write identifiers are used by stores to check the ordering of writes. To implement RYW coherence, the identifier of the last performed write and the identifier of the store on which it has been performed has to be saved. This *dependency* $(WiD, store\_id)$ is transmitted with a read request to the cache.

Stores

In contrast to clients, stores implement the Web page by means of semantics object. Each store implements a version number ($expected\_write[client]$) that contains the value of the sequence number of the last performed write or update for each client.

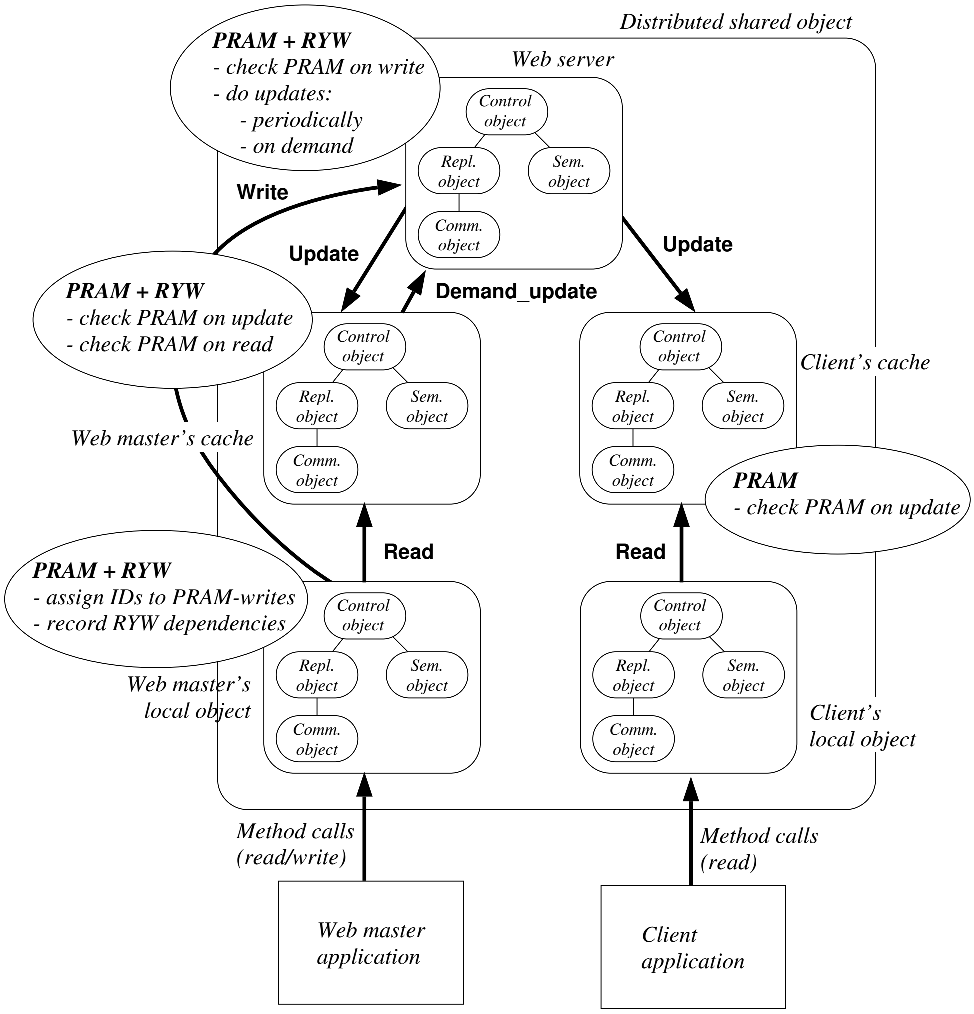

Figure 4: Implementation of the example in Globe.

The replication object of cache U implements PRAM coherence. Upon receipt of an update (which can come only from the Web server), the sequence number of the incoming update's *WiD* is compared to the client's version number ($expected\_write[clientM]$). If they are equal, then all previous updates have been performed and the new update is performed as well. Otherwise, the update request is buffered and the store waits until the next one.

The replication object of cache M (the Web master's cache) implements object-based PRAM and client-based RYW coherence. PRAM coherence is implemented as in the user's cache. To implement RYW coherence, $expected\_write[clientM]$ is compared to $dependency.sequence\_number$ on each incoming read request. If the write expected by the read operation has already been propagated, the read is immediately performed. Otherwise, the cache first demands an update from the Web server (*demand_update* message).

The replication object of the Web Server is responsible for performing incoming writes according to the PRAM coherence model. This is done exactly the same as in the caches. In addition, the replication object has to multicast updates either periodically or when an update has been demanded from cache M. Therefore, the communication object of the Web server offers a multicast facility in addition to the point-to-point communication facility.

It is important to note that the replication objects all have the same interface. This means that the flow of control within the local object is more or less the same everywhere. However, the internals of the replication objects differ as each implements its own part of a coherence protocol.

In our prototype, we have used TCP/IP for the sake of simplicity to provide reliable communication. The ordering that a TCP connection provides, however, does not interest us since we leave the ordering guarantees to the application. Effectively, a coherence protocol inherently leads to ordering constraints. This means that as soon as a coherence model is implemented for a Web object, all ordering constraints imposed by that coherence model are also implemented. Moreover, ordering constraints may even allow reliable communication to be implemented at no additional cost. For example, in our prototype, we could have used UDP, instead of TCP/IP, for more efficiency and directly use the PRAM object-based model to implement reliability. Then, simply by changing the object-outdate reaction parameter from *wait* to *demand*, reliability comes as a side-effect of the coherence model. These observations are in line with the various discussions on end-to-end arguments in system design [11], and show the impact that a coherence model and its implementation may have on implementing scalable solutions.

5 Conclusions

In this paper, we have presented a framework for consistent, replicated Web objects. This framework uses a symmetric object-based model which allows the encapsulation of policies for distribution, replication, and coherence on a per-object-basis. The goal of our architecture is to achieve scalability and efficiency in any worldwide distributed system.

In our architecture, both clients and servers participate in implementing an adequate coherence protocol for a given Web object. Servers exploit their knowledge about the nature of the Web object and its access patterns, to optimally replicate the state of the object and apply an appropriate coherence protocol. In addition, clients can express their own coherence requirements. This approach will contribute to better global performance. A practical implementation of a conference Web home page has been developed in Java to show the flexibility of our approach.

Some related work has been done to change current Web caching, such as giving more flexibility or more control to the server. In Wessels' approach [14], servers are allowed to explicitly grant or deny a client permission to cache an object. In push-caching [5], servers use their knowledge of access patterns to optimally distribute popular objects to other servers. This approach is comparable to installing object-initiated stores in our architecture. However, it does not address coherence models for Web objects. The W3objects system [3] has similar aims as ours. However, in the W3objects model, coherence protocols are not encapsulated within an object but are under the control of a separate object administrator, thus limiting the scalability of the approach. Moreover, their goal is to provide a highly visible caching mechanism whereas we aim at maximum transparency. Also, W3objects do not address coherence models for Web objects as we do.

The main contribution of the research described in this paper is to first address coherence models required for current and future Web objects. As interfaces are fully standardized in our object-based infrastructure, new coherence models can easily be integrated within our framework. Thus our approach provides the flexibility needed in an evolutionary system such as the Web. We are currently working on a complete implementation of the framework. Future research consists of defining self-adaptive policies by which implementation parameters can be changed dynamically. We plan to exploit our approach for building wide-area distributed Web servers.

References

[1] M. Ahamad, M. Raynal, and G. Thia-Kime. "An Adaptive Architecture for Causally Consistent Distributed Services." Technical Report PI-1039, IRISA, Rennes, France, July 1996.

[2] M. Baentsch, L. Baum, G. Molter, S. Rothkugel, and P. Sturm. "Enhancing the Web's Infrastructure: From Caching to Replication." *IEEE Internet Comput.*, 1(2):18–27, Mar. 1997.

[3] S. Caughey, D. Ingham, and M. Little. "Flexible Open Caching for the Web." In *Proc. Sixth Int'l WWW Conf.*, Santa Clara, CA, Apr. 1997.

[4] A. Dingle and T. Pártl. "Web Cache Coherence." *Comp. Netw. ISDN Syst.*, 28(7–11):907–920, 1996.

[5] J. Gwertzman and M. Seltzer. "The Case for Geographical Push-Caching." In *Proc. Fifth HOTOS*, Orcas Island, WA, May 1996. IEEE.

[6] P. Homburg, M. van Steen, and A. Tanenbaum. "An Architecture for A Scalable Wide Area Distributed System." In *Proc. Seventh SIGOPS European Workshop*, pp. 75–82, Connemara, Ireland, Sept. 1996. ACM.

[7] P. Hutto and M. Ahamad. "Slow memory : Weakening consistency to enhance concurrency in distributed shared memories." In *Proc. Tenth Int'l Conf. on Distributed Computing Systems*, pp. 302–311. IEEE, 1990.

[8] L. Lamport. "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs." *IEEE Trans. Comput.*, C-28(9):690–691, Sept. 1979.

[9] R. Lipton and J. Sandberg. "PRAM : A Scalable Shared Memory." Technical Report CS-TR-180-88, Princeton University, Sept. 1988.

[10] B. Neuman. "Scale in Distributed Systems." In T. Casavant and M. Singhal, (eds.), *Readings in Distributed Computing Systems*, pp. 463–489. IEEE Computer Society Press, Los Alamitos, CA., 1994.

[11] J. Saltzer, D. Reed, and D. Clark. "End-to-End Arguments in System Design." *ACM Trans. Comp. Syst.*, 2(4), Nov. 1984.

[12] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welsh. "Session Guarantees for Weakly Consistent Replicated Data." In *Proc. Third Int'l Conf. on Parallel and Distributed Information Systems*, pp. 140–149, Austin, TX, Sept. 1994. IEEE.

[13] M. van Steen, P. Homburg, and A. Tanenbaum. "The Architectural Design of Globe: A Wide-Area Distributed System." Technical Report IR-422, Vrije Universiteit, Department of Mathematics and Computer Science, Mar. 1997.

[14] D. Wessels. "Intelligent Caching for World-Wide Web Objects." In *Proc. INET '95*, Honolulu, Hawaii, June 1995. Internet Society.