INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Replicated Directory Service for Weakly Consistent Distributed Caches

Mesaac Makpangou, Guillaume Pierre,

Christian Khoury and Neilze Dorta

No 3514

Octobre 1998

THÈME 1

Rapport de recherche

ISSN 0249-6399

Replicated Directory Service for Weakly Consistent Distributed Caches

Mesaac Makpangou, Guillaume Pierre,
Christian Khoury and Neilze Dorta

Thème 1 — Réseaux et systèmes
Projet SOR — http://www-sor.inria.fr/

Rapport de recherche n ° 3514 — Octobre 1998 — 19 pages

Abstract: Relais is a replicated directory service that improves Web caching within an organization. Relais connects a distributed set of caches and mirrors, providing the abstraction of a single consistent, shared cache. Relais is based on a replication protocol that exploits the semantics of user requests to guarantee cache coherence. The protocol also exploits the semantics of origin servers to optimize bandwidth requirements. At the entry point for a client, Relais maintains a directory of the contents of each Relais data provider; this minimizes look-up latency, by making it a local operation. The cost of maintaining the directories consistent is minimized thanks to a weak consistency protocol. Relais has been prototyped on top of Squid; it has been in daily use for over a year.

Key-words: directory service, cooperative caches, weak consistency.

(Résumé : tsvp)

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : 01 39 63 55 11 - International : +33 1 39 63 55 11
Télécopie : (33) 01 39 63 53 30 - International : +33 1 39 63 53 30

Un service de répertoire répliqué pour des caches distribués faiblement cohérents

Résumé : Relais est un service de répertoire répliqué qui permet un meilleur fonctionnement du Web. Relais fédère un ensemble de dépôts intermédiaires (caches et de miroirs) répartis dans l'entreprise, pour créer l'abstraction d'un cache partagé unique. Relais est basé sur un protocole de réplication qui exploite la sémantique des requêtes des utilisateurs afin de garantir la cohérence des caches. Le protocole exploite également la sémantique des serveurs pour optimiser les besoins en bande passante. Relais maintient, à chaque point d'accès utilisateur, un répertoire indiquant le contenu de tous les dépôts de documents ; les opérations de recherche peuvent donc être effectuées en local, ce qui en réduit la latence. Le coût du maintien de cohérence des répertoires est réduit grâce à un protocole de cohérence faible. Relais a été prototypé au-dessus de Squid ; il est utilisé en exploitation depuis plus d'un an.

Mots-clé : service de répertoire, caches coopératifs, cohérence faible.

1 Introduction

It is nowadays common to see the personnel of different divisions of large organizations share documents provided by servers located abroad. Examples include general purpose documents (e.g. electronic news papers, stock quotes) and documents related to the enterprise activities (e.g. standards, information on clients and suppliers). However, accessing electronic documents located abroad is slow and expensive. In addition, these documents are not always available. Problems are due to asynchronous communication, server load, network congestion, communication latency, and the characteristics of the Internet (network partitions, server disconnections and communication failures, for example).

To improve the access response time while reducing the economic cost, divisions cache external documents at nearby intermediate providers (e.g. mirrors or caching proxies [5]). We argue that a location service capable of discovering all documents cached within the organization's boundaries and of binding each user's request to the replica offering the best response time is needed in order to benefit from these intermediate providers. To achieve that, intermediate providers need to cooperate in order to share contents. The cooperation must preserve the autonomy of each division. That is, each division's intermediate provider should function even if it can no longer communicate with other divisions' intermediate providers. Letting different intermediate providers progress independently may lead to the presence of different versions of a document in the enterprise. This raises the problem of which version should be returned when a client requests the document. A blind choice may lead to delivering a version which is older than one already retrieved. This behavior is confusing and could encourage users to bypass intermediate providers for future accesses.

Although the need of such a location service has already been identified [1], existing solutions, especially in the context of the Web [3, 4, 6, 10, 15], have limitations. First, although they provide transparent access to caching proxies, accessing a mirror site is not transparent. Second, the query procedure introduces additional latency during document access. Third, existing systems do not provide guarantees regarding the consistency of document versions delivered. In particular, a client may retrieve a version which is inconsistent with respect to previously retrieved versions.

This paper presents Relais, a directory service for large organization-wide distributed caches. Relais connects together the intermediate providers, offering the abstraction of a shared distributed document cache. Figure 1 shows an example of a distributed cache composed of two caching proxies and two mirrors.

Relais provides a distributed *directory service*, i.e. it resolves the name of a document (a URL) into a location; furthermore it provides the "best" location among a set of replicas for that document within the organization's boundaries. To do so, it maintains a *directory* of the contents of all intermediate providers in the organization. The directory is replicated, ensuring that name resolution is always a cheap, local operation.

Relais relies on an abstract document model; interface objects are responsible for converting the service API, for instance HTTP requests or the filing service interface, into the Relais directory generic interface. This architecture enables Relais to connect different kinds

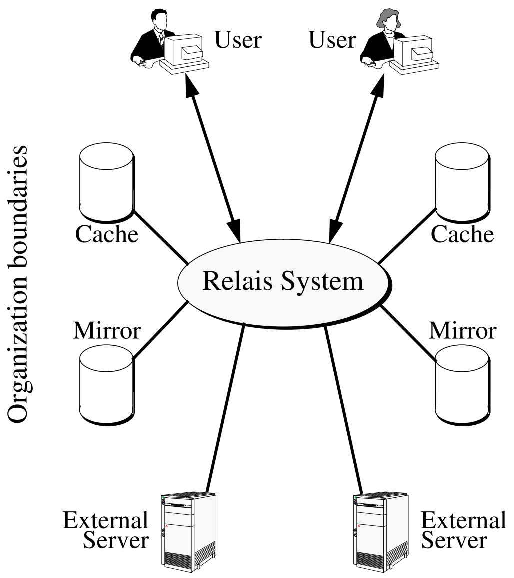

Figure 1: Concrete Representation of a Distributed Cache

of intermediate providers (e.g. Web caching proxies, replicated file systems, and replicated databases).

Relais offers its users two consistency guarantees: *Read Monotonic* and *Read Your Writes* [13].[1] Relais also guarantees that once a newer version of a document has been noticed by any intermediate provider, all its neighbors will rapidly be informed in order to update their copies as well; this improves individual cache consistency with respect to the servers and the group coherency as well. Group coherency is important for example when several persons from different divisions need to have a common view on some evolving documents located abroad.

Finally, Relais tolerates network partitions and disconnections; it implements a lightweight protocol to recover from short disconnections and a heavyweight protocol to recover from long disconnections or failures.

The rest of the paper is organized as follows: Section 2 presents the background to our work. Section 3 describes the directory replication protocol. Section 4 discusses related work. Finally, Section 5 draws some conclusions and points at future work.

[1] The former guarantees that a client sees monotonically increasing versions of a document (i.e. time never moves backwards). The latter guarantees that each client sees its own modifications in the appropriate order.

2 Background

We consider a *document* to be any kind of resource. It might be a file, a Web document, a memory segment, an instance of an abstract data type, or a cluster of objects. Each document is designated by a URL. The document is the unit of caching and consistency management. Each document is owned by a single server, called its *origin server*.

A document can evolve over time (i.e. new versions of documents can be created at any time). We assume that, when a document is requested from its origin server, the server delivers an up-to-date copy together with a timestamp that identifies the returned version.

Document copies can be stored at *intermediate providers* within the organization's boundaries. Examples of intermediate providers include Web caching proxies, mirrors, archives, file systems, etc. We refer to locations of intermediate providers as the organization *replication sites*. Locations where users submit their document requests are called *access points*. Note that a single location may play both roles. For instance, a mirror provides information but isn't an access point; a non-caching proxy is only an access point; a caching proxy plays both roles.

When an intermediate provider retrieves a version of a document from the server, it becomes a *primary replica site* for this version. Each document cached within the system has a primary replica site: the notion is independent of whether or not this site actually owns a replica of the version. Note that a version may have several primary replica sites if several intermediate providers independently retrieved the same document from the server.

The remaining of this section introduces the Relais architecture. Relais comprises three kinds of components: *provider agents*, *location proxies* and *directory managers* (see Figure 2). Provider agents interface intermediate providers with other components of Relais, whereas location proxies enable users to access documents from the shared distributed cache. Directory managers replicate the shared directory.

2.1 Provider Agents

A *provider agent* allows an intermediate provider to register documents in their possession with the shared distributed directory. An intermediate provider notifies its provider agent when adding or removing a document, and when detecting a newer version of a document.

Provider agents manage document name aliases. For example, a document stored in a mirror has a different name (i.e. URL) than the original document. However, the users wish such renaming to remain transparent, i.e. to designate all copies uniformly. To solve this problem, the provider agent registers a replica under its original name. It then takes care of the mapping when a replica is requested under its global name.

2.2 Location proxies

A *location proxy* enables a user to transparently access the group of intermediate providers that make up the organization-wide distributed cache. A user submits a document name

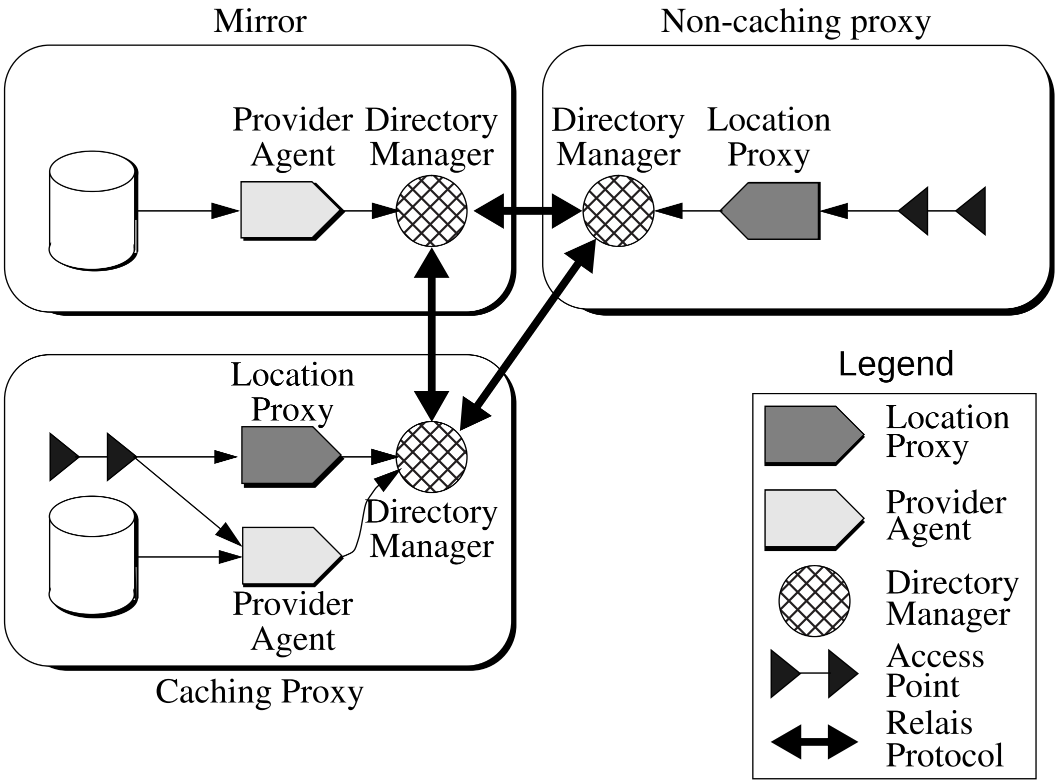

Figure 2: The Relais Architecture

to his associated location proxy. We make the hypothesis that a user doesn't change his location proxy.

The location proxy first looks up the name in the directory. If the document is available in the system, the directory manager returns a list of locations where replicas can be retrieved. The location proxy chooses a preferred replica, either by itself or by delegating the choice to an external service such as SPAND [11].

A location proxy also informs the local directory manager when new versions of documents are observed or when users update originals.

2.3 Directory Managers

The directory is replicated towards the organization access points by the *directory managers*. They perform registration and removal of document locations, invalidation of cached copies when their originals are updated, and lookup of available locations.

Lookup requests are resolved by simply consulting the local copy of the directory. All other requests (register, remove, invalidate) trigger notification messages to be sent to other managers to keep directory replicas consistent.

The main contribution of Relais is the directory replication protocol. The rest of the paper focuses on that aspect.

3 Directory Replication Protocol

The replication protocol is implemented by the group of directory managers. They cooperate with one another by exchanging notifications. Notification types include: `Add`, `Delete`, `Invalidate` and `Probe`. The three former are notifications of application-related events, whereas the latter is used by the protocol itself (for example, `Probe` notifications may be used to request the partner's time).

3.1 Logical Clocks

Relais relies on logical clocks to timestamp notifications exchanged within the group of directory managers. Relais requires each directory manager to maintain a logical clock that allows it to timestamp notifications that it generates during its lifetime. Timestamps totally order the notifications from a given manager, such that the order is consistent with the order in which the notified events occurred.

In practice, each directory manager maintains a local logical clock of two components: a session component s and a counter c. The session component corresponds to the last restart time of the site. The counter component is initialized at site restart time, then incremented at each occurrence of an interesting event (i.e. registration, removal, invalidation or probe).

3.1.1 Global Virtual Time

Relais implements a reliable FIFO delivery of notifications. Since notifications carry their timestamps, each directory manager can approximate the time of its partners based on the timestamps of notifications received from these partners. These approximations constitute what we call the local manager's *global virtual time.*

More precisely, each manager approximates the global virtual time of the group with a vector of clock values, one entry per partner. The entry associated with each remote partner contains the timestamp of the most recent event notified by that partner; the entry corresponding to the local manager contains its logical clock.

3.2 Basic Replication protocol

For the purpose of this article, we assume that Relais is deployed on n sites, numbered $1, ..., n$. To simplify the presentation, we also assume that every site is both a replication site and an access point, and that there exists one intermediate provider per site.

This section concentrates on the basic protocol. Initially we also assume the absence of failures and disconnections, and do not address consistency issues. These assumptions will be relaxed later in the paper.

An example scenario depicts the basic replication protocol in Appendix A.

3.2.1 Basic Notifications

A local copy of the shared directory is up-to-date if the locations of all the accessible documents in the group are referenced in the directory, and there are no dangling references. For that, Relais managers must be informed whenever a new copy is added to, or removed from, an accessible intermediate provider.

When the directory manager receives a registration (resp. removal) request, a `Add` (resp. `Delete`) notification is multicast to the group. The `Add` (resp. `Delete`) notifies the arrival of a new (resp. removal of an old) copy at an intermediate provider.

Each basic notification carries the following information: its type (`Add` or `Delete`); the document URL; the provider agent address; the version timestamp of the notified copy; the identifier of the primary replica site of the version; and the logical clock of the document's primary replica site at the registration time.[2]

3.2.2 Notification Protocol

The notification protocol enforces reliable delivery of notifications in generation order. The Relais notification system has two layers: the queuing layer and the transport layer.

The queuing layer maintains, for each destination, a queue of notifications yet to be sent to this destination. Each notification in the queue is timestamped with the local logical clock of the notified event. A notification is removed from the queue only after its delivery has been acknowledged. Note that the length of the queue of outstanding notifications for some destination provides a hint about this partner's connectivity: the longer it is, the higher the probability that the partner is disconnected.

For each partner, periodically (or when the length of the queue reaches some limit) the transport layer sends a message. Each message contains a sequence of queued notifications, starting from the oldest outstanding notification for that partner.

To reduce bandwidth consumption, timestamps attached to notifications are not packed in the transport message. Rather, the transport message contains only the timestamp of the youngest notification contained in the message. This is sufficient because the sequence of notifications is delivered to the destination atomically; in addition, the delivery system indicates the order in which the notifications were originally sent.

The transport message also carries the timestamp of the most recent event that occurred at the destination site and which has been notified so far. This timestamp acknowledges previous notifications received from this destination. Hence, each manager knows which notifications have already been delivered to its partners and can remove them from outstanding notification queues.

[2]The last two items are needed by the heavyweight recovery protocol; see Section 3.4.2 for details.

Conversely, if a message carries notifications that were already delivered, the receiver will detect them by comparing the virtual time of the sender and its view of that time. When the view is greater than the announced virtual time, the message is ignored.

3.3 Monotonic Progress in Document Histories

Here, we consider a consistency guarantee, the *Read Monotonic* guarantee. This guarantees that no client will ever see successive versions of a document that go back in time. To achieve that, each manager should remember a *version threshold* for each document it has already delivered to its users; this is a timestamp that identifies the most recent version that was delivered to its users. Only a copy with a version timestamp greater or equal to this threshold may be referenced by a manager.

One problem with this solution is memory consumption. The number of documents accessed from each access point is likely to grow quickly over time, but not all documents that have been accessed so far need to be remembered.

When a document is no longer available within the organization, it is pointless to maintain an entry in the shared distributed directory for that document. However, entry reclamation must preserve the guarantee of monotonic progress. To achieve this, a manager may only reclaim an entry in its copy of the shared distributed directory if the following two conditions are satisfied: (i) no copy of the corresponding document is referenced any more; (ii) the manager is certain that the current threshold version is *stable* within the group of managers; this means that all managers know either this version or a more recent one.

We can detect stable versions within a group of n managers using a $n * n$ array that we name GS and which is constructed as follows. Raw i or GS correspond to the last known global virtual time of manager i. To update this row, each manager periodically pushes its global virtual time to the entire group, using a `Probe` notification.

To determine whether a version carried by an `Add` notification a, generated by manager k at the local time t, is stable (that is, has already been received by all managers), we only need to check the following condition:

$$\forall j \in \{1, \ldots, n\}, GS[j, k] \geq t$$

Any notification carrying the version may be used to check the stability of a version within the group.

Hence, to limit memory consumption, managers allocate directory entries only for documents currently cached in at least one intermediate provider, or for documents with unstable version thresholds. Once a manager has noticed the stability of that version, it can no longer receive an `Add` notification concerning a version older than the stable one. When a manager receives a request for a document that it doesn't reference, it defaults to the origin server, which in turn always delivers an up-to-date document. Therefore the protocol preserves the Read Monotonic guarantee.

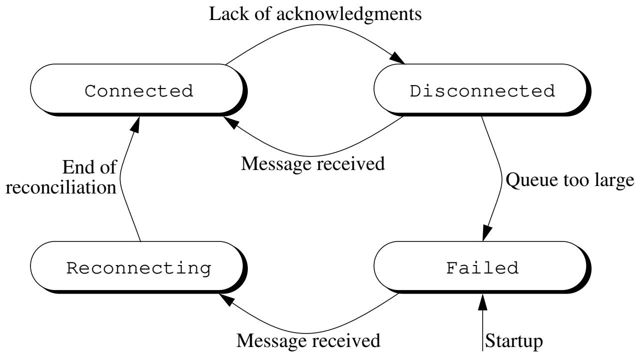

Figure 3: Manager State Transition Diagram

3.4 Recovery Protocols

In this section, we consider that Relais managers can disconnect from one another. We want to avoid consuming network and memory resources with notifications to disconnected partners. For that, each manager monitors its connections with its peers. Upon reconnection of a partner presumed disconnected, managers run a recovery protocol. Relais implements a lightweight protocol to recover from disconnections, and a heavyweight protocol to recover from failures (or long disconnections)

3.4.1 State Transition Rules

From a manager's viewpoint, another manager is considered to be in one of four states: `Connected`, `Disconnected`, `Failed` and `Reconnecting`. Figure 3 shows the state transition diagram.

Let M_1 and M_2 be two Relais managers. M_1 considers M_2 as `Connected` if M_2 acknowledges notifications sent by M_1. If M_2 does not acknowledge M_1's notifications in time, M_1 will consider M_2 to be `Disconnected`. M_1 will stop sending notifications to M_2. Instead these notifications will be stored in a queue.

Periodically, M_1 will send a probe notification. Probe notifications contain the acknowledgment of the last notification received from M_2, the local clock of M_1 and the presumed state of M_2 (`Disconnected`). When M_2 receives a probe saying that it is considered `Disconnected`, it simply acknowledges the probe along with any outstanding notifications destinated to M_1.

If M_1 receives a message from M_2, it re-enters a normal notification mode with respect to M_2. That is, M_2's state record is changed to `Connected`, and M_1 sends any outstanding notifications to M_2. We call this the lightweight recovery protocol.

However, if M_2 stays silent for a long period, the queue of outstanding notifications destined for M_2 might become too large. When the cost of maintaining these notifications

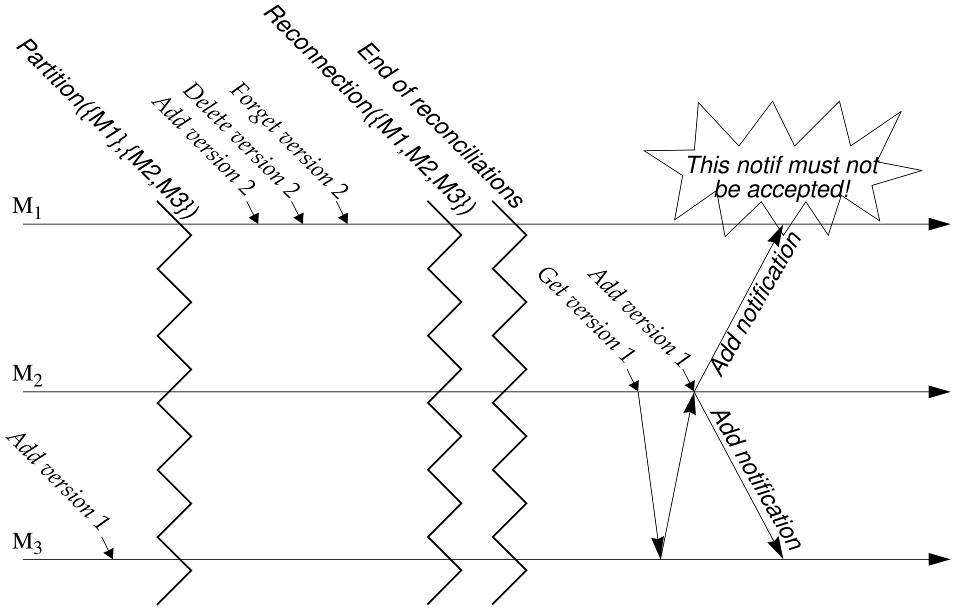

Figure 4: Scenario that may lead to the violation of read-monotonic guarantee after a recovery

becomes too high, M_1 declares that M_2 has `Failed`, and deletes all resources dedicated to M_2. This decision is local and concerns only M_1. One important consequence of this decision is that M_2 will no longer consider M_1 when checking whether a version threshold of a document is stable. In other words, M_2 may decide to reclaim an entry allocated to a document without any information about the view of M_1.

Later on, if M_1 receives a probe or any other notification from M_2, it will initiate the heavyweight recovery protocol.

3.4.2 Heavyweight Recovery Protocol

When two managers reconnect after a long period of disconnection, each one may have documents that could interest the other. Also, each manager may have reclaimed certain entries of the shared repository during the disconnection. Hence the heavyweight recovery protocol has two objectives.

Firstly, it allows the managers to inform each other of the documents in their possession. Interesting documents are those to which a manager can bind its users, without violating the Relais consistency guarantees. A conservative solution is, for a manager which has considered the other `Failed`, to only reference documents (in the possession of the presumed failed partner) for which it still remembers the most recent version delivered to its users.

Secondly, the recovery protocol should guarantee that, after the recovery, the recovered manager will not introduce inconsistencies. For instance, after the reconnection of a system that has been partitioned, if different partitions have observed different versions of a document there is a risk of forcing certain sites that were in a separate partition to "backup" to their past.

Figure 4 shows a scenario that illustrates this problem. In this scenario, the organization has three sites. M_1, M_2 and M_3 are managers of the replicated directory. Let us suppose that the system has been partitioned for some time; the partitions are $\{M_1\}, \{M_2, M_3\}$. In our scenario M_2, after its recovery, retrieves a copy of version $(v1)$ of document d from M_3. This respects the Relais guarantees even though version $v1$ is outdated. M_2 then decides to conserve a copy of d in its cache; once cached, it sends an `Add` notification to M_1 and M_2. If this notification were accepted by M_1, there would be a risk of serving this document version to a user who had already observed a more recent version $(v2)$.

To avoid this problem, Relais managers implement a conservative solution: after manager M_2 recovers from a `Failed` state (from the point of view of M_1), `Add` notifications from M_2 are acceptable for M_1 only if the primary replica site of the document had retrieved it after M_2's recovery. The problem is now to determine that a copy was retrieved after M_2's recovery. At the end of the recovery procedure, the manager that had suspected the other to be in the `Failed` state sends a `Probe` notification p to request the others' local time. The vector clock $\Pi t(p)$ which captures the logical time at each partner, at the moment it receives the `Probe` notification p, partitions the `Add` notifications from each partner into two categories (see Figure 5): those that occurred before the reception of p and those that occurred after.

Formally, if manager M_i is the sender of p, $\Pi t(p)$ is defined as follows:

$$\begin{cases} \forall j \in (1, ..., n), j \neq i, \Pi t(p)[j] = V_j[j] \\ \Pi t(p)[i] = V_i[i] \end{cases}$$

where $V_j[j]$ is the logical time of manager M_j at the reception of p, and $V_i[i]$ is the logical time of manager M_i at the occurrence of p.

An `Add` notification, denoted a, from M_2 is safe for M_1 if

$$\Pi t(p)(a.prs) < a.prs\_time$$

where $a.prs$ is a's primary replica site and $a.prs\_time$ is the local time of $a.prs$ at the time of the retrieval from the server (both information are contained in the `Add` notification; see Section 3.2.1).

To summarize, each manager keeps, for each other partner, a vector clock that provides the global virtual time at the moment it last reconnected to that partner. Each vector clock partitions the `Add` notifications from the corresponding partner into two categories and only those concerning versions retrieved after the last reconnection with that partner are considered safe. Unsafe notifications are rejected by each partner.

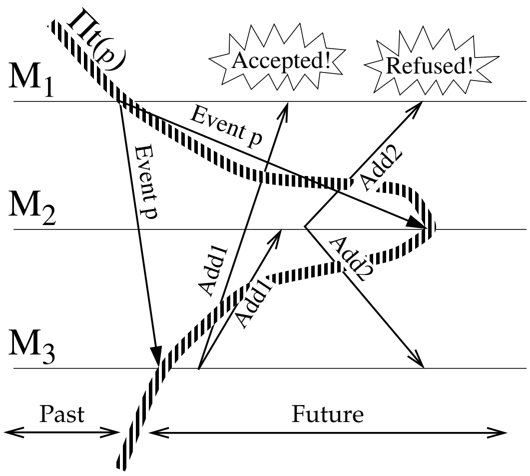

Figure 5: Event Global Time

Note that, at startup time, each manager considers other partners to be in the `Failed` state. The manager then builds an up-to-date local copy of the shared directory by running the previously described heavyweight recovery procedure with each partner that it can contact.

3.5 Distributed Cache Coherency

Relais offers the abstraction of a distributed document store. One objective of Relais is to ensure the rapid progress of the overall distributed cache on each document history, while letting each intermediate provider implement its own consistency protocol.

Let's consider two provider agents P_1 and P_2, and a document d. Consider the following scenario:

1. P_2 retrieves a copy of d from the origin server and informs its associated manager M_2. M_2 in turn, notifies the storing (an `Add` notification) of the retrieved copy.
2. Later on, P_1 receives an invalidation request from the origin server indicating that d has changed. P_1 invalidates its cached copy, then informs its associated manager, M_1 for example, which in turn updates its local copy of the shared directory to no longer point to obsolete copies; M_1 then notifies its peers, using the `Invalidate` notification type.

For example, if the server doesn't indicate the current version to $P1$, and if the `Add` notification sent by M_2 arrives after the treatment of the `Invalidate` notification by M_1, there is a risk of installing a reference to a copy of an obsolete version of the document.

The problem we face here is how to determine whether a copy referenced by the directory was effectively retrieved before or after an invalidation. Furthermore, `Invalidate` and `Add` notifications concerning the same document may be concurrent.

When an `Invalidate` notification carries the current version, the problem is easier and is treated as an `Add` notification that does not indicate the address of a provider agent. However, when the new version is not specified (which is common for most invalidation-based cache consistency protocols), things get complicated.

In the current design, Relais implements a simple solution. An `Invalidate` notification carries the old version being invalidated and, if available, the up-to-date version. When a manager receives an `Invalidate` notification which specifies the new version, it treats it as an `Add` carrying a new version: it forgets current locations and updates the version threshold. If an `Invalidate` notification doesn't indicate the new version, each manager invalidates its local copy of the notified document if its version is older or equal to old version specified by the notification. Removals of copies are notified to the group using a normal `Delete` notification.

4 Related Work

Our work tackles a number of issues that were addressed by several other systems. These include the directory service abstraction in large-scale settings, cooperative cache protocols, consistency guarantees in a weakly consistent system, and update propagation in a disconnected environment. Hereafter, we point out some of these systems and compare their objectives and/or solutions to ours.

4.1 General Purpose Location Service

A number of general purpose location systems have been proposed in the past, including the ClearingHouse [9], DNS [7, 8] and the Globe location service [14]. These systems operate at a worldwide scale, which implies managing a huge quantity of information. Consequently, they use tree architectures, and propagate the location requests along the branches of the tree.[3] As the location mappings that they manage don't change often, caching techniques can easily be used to minimize the number of requests to the location system.

In contrast, the location information managed by Relais changes very often. Relais places its use in smaller-scale systems: each group should link together a relatively small number of directory managers. This enables us to totally replicate the location directory, and permits local lookups. However, the size of Relais' directory is still an issue when managing large numbers of documents. We will address it in our future research.

[3] The depth of the tree is fixed to 2 in ClearingHouse, and is variable in the other systems.

4.2 Cooperative Cache Protocols

Relais can be used to implement cooperative caches. A number of cooperative cache protocols have already been proposed, first in the file system area, and later for the World-Wide Web.

Cooperative file system caches place themselves in local area networks [2]. As the LAN is faster than the disks, when requesting a file (or a block), the server can redirect the request to a neighboring cache rather than accessing its disk. This approach differs from ours: the location information can be centralized at the server point (possibly reusing the invalidation callback references).

Cooperative Web caches are much closer to the issue we address: considering that accessing the servers is costly, how do a number of caches share their contents?

The simplest mechanism is "request on demand": when a cache misses a document, it queries its neighbors in order to locate nearby copies. This technique is used by the ICP protocol [15]. The drawback of this protocol is that it suffers significant latency and generates a lot of traffic. To reduce this problem, Crisp [4] proposes a mapping server that maintains location related information. Now, when a cache misses a document, it queries only the mapping server to obtain alternative locations of the document. The Crisp mapping server may be centralized, partitioned or replicated at several sites.

Summary caches [3] and Cache Digests [10] pursue the same goal as Relais: to make the shared directory available locally to each member of the group so as to minimize the location cost. While with Relais each cache logically notifies all interesting events to its partners, Summary caches and Cache Digests propose that each cache computes periodically a compressed representation of its contents and sends it to its partners. The drawback is that the content representations must be exchanged often in order to avoid excessive inconsistencies.

None of these protocols address the document consistency issue. However, distributed caching introduces problems such as the Read Monotonic guarantee violation. We believe that such counter-intuitive cache behaviors should be avoided; if we don't address this point conveniently, users might decide to bypass their caches.

5 Conclusion

We present Relais, a directory service that permits efficient and transparent location of document copies cached within the organization boundaries. Relais is a basic building block for efficient sharing of information among well-delimited groups of users, possibly distributed at geographically separate sites. It exploits efficiently the internal resources of an organization (such as communication links, storage space, and CPU time) to improve performance and availability, while reducing access to external resources (e.g. public networks, Web servers, and databases).

Relais has been prototyped on top of Squid [12]. This prototype has been used to locate Web documents in clusters of caching proxies. In particular, it has been in daily use in our institute for more than a year. Although we didn't yet study the benefits of providing the Read Monotonic and Read Your Writes consistency guarantees, we consider them essential to users' acceptance of caches.

Looking to the future, our objectives include three points. Firstly, we will address the scalability issue. With the current design, each manager has a huge state to maintain. We also want to get rid of the total replication of the directory. We are working on structuring the cooperation group into subgroups for building a kind of "cooperation hierarchy". Secondly, we are considering the use of Relais for other application domains. In particular, we plan to use Relais to enable efficient sharing in a large-scale file server, when the number of clients is large. Another planned case study is a large-scale distributed object store for cooperative engineering in virtual enterprises. Thirdly, we are considering the refinement of the protocol to remove current limitations; in particular, we should relax the requirement for a user to be bound to a single location proxy once and for all.

6 Acknowledgements

We would like to thank Marc Shapiro, Ian Piumarta, Georges Brun-Cottan and Jan Vitek for their helpful comments on the paper. We would also like to thank Éric Bérenguier and Stéphane Dugelay who participated to the design and implementation of Relais.

References

[1] C. M. Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, and Michael F. Schwartz. The Harvest information discovery and access system. *Computer Networks and ISDN Systems*, 28(1-2), December 1995.

[2] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 267–280, 1994.

[3] Li Fan, Pei Cao, Jussara Almeida, and Andrei Broder. Summary cache: a scalable wide-area Web cache sharing protocol. In *Proceedings of the SIGCOMM '98 conference*, September 1998. `http://www.acm.org/sigcomm/sigcomm98/tp/abs_21.html`.

[4] Syam Gadde, Michael Rabinovich, and Jeff Chase. Reduce, reuse, recycle: an approach to building large Internet caches. In *Proceedings of the HotOS '97 Workshop*, May 1997. `http://www.cs.duke.edu/ari/cisi/crisp-recycle.ps`.

[5] Ari Luotonen and Kevin Altis. World-Wide Web proxies. In *Proceedings of the 1st International WWW Conference*, Geneva, Switzerland, May 1994. `http://www1.cern.ch/PapersWWW94/luotonen.ps`.

[6] Jean-Marc Menaud, Valerie Issarny, and Michel Banâtre. A new protocol for efficient transversal Web caching. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC '98)*, September 1998.

[7] P. Mockapetris. Domain names - concepts and facilities. RFC 1034, November 1987. `ftp://ftp.inria.fr/rfc/rfc10xx/rfc1034.Z`.

[8] P. Mockapetris. Domain names - implementation and specification. RFC 1035, November 1987. `ftp://ftp.inria.fr/rfc/rfc10xx/rfc1035.Z`.

[9] Derek C. Oppen and Yogen K. Dalal. The Clearinghouse: a decentralized agent for locating named objects in a distributed environment. In *ACM Transcations on Office Information Systems*, volume 1, July 1983.

[10] Alex Rousskov and Duane Wessels. Cache digests. In *Proceedings of the 3rd International WWW Caching Workshop*, June 1998. `http://wwwcache.ja.net/events/workshop/31/rousskov@nlanr.net.ps`.

[11] Srinivasan Seshan, Mark Stemm, and Randy H. Katz. SPAND: Shared passive network performance discovery. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97)*, Monterey, CA, December 1997. `http://http.cs.berkeley.edu/~stemm/publications/usits97.ps.gz`.

[12] Squid home page. `http://squid.nlanr.net/Squid/`.

[13] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, pages 140–149, Austin, Texas, September 1994.

[14] Maarten van Steen, Philip Homburg, Andrew S. Tanenbaum, and Franz J. Hauck. Locating objects in wide-area systems. *IEEE Communications Magazine*, pages 104–109, jan 1998. `ftp://ftp.cs.vu.nl/pub/papers/globe/commag.98.ps.Z`.

[15] Duane Wessels and K. Claffy. Internet cache protocol (ICP), version 2. RFC 2186, September 1997. `ftp://ftp.inria.fr/rfc/rfc21xx/rfc2186.Z`.

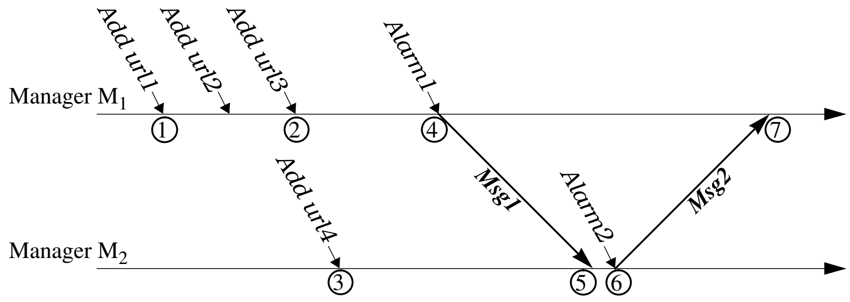

Figure 6: Scenario timeline

A Example scenario

We consider a system composed of two Relais managers M_1 and M_2. For illustration purpose, assume that a transport message can carry two notifications at a time. Figure 6 depicts a scenario where three new documents are registered on M_1 before the transport layer sends a message to M_2; concurrently, one document is registered on M_2 before the reception of notifications from M_1.

This section details the actions performed by each manager at the occurrence of each event in the scenario. Both caches are initially empty and

$$\forall i, j \in \{1, 2\}, V_{M_i}(M_j) = 0$$

where V_{M_i} designates M_i's global virtual time.

① Registration of url_1:
M_1 adds the corresponding entry in its local copy of the shared directory and increments its local clock: $V_{M_1}(M_1) = 1$. M_1 also generates an `Add` notification and adds it to M_2's queue.

$$\text{Notif 1 : } \boxed{\texttt{ADD}\, url_1 \ \texttt{version}\,1\,\texttt{in}\,\texttt{cache}\, M_1}$$

② Registration of url_2 and url_3:
Similar to ①. At this point, M_1's local clock is $V_{M_1}(M_1) = 3$ and there are three notifications (Notif 1, Notif 2 and Notif 3) in the queue to M_2.

③ Registration of url_4:
Similar to ① but at manager M_2. At this point, M_2's local clock is now $V_{M_2}(M_2) = 1$ and there is one notification (Notif 4) in queue to M_1.

④ M_1's transport layer sends a message:

$$\text{Msg 1 : } \boxed{\text{Notif 1} \mid \text{Notif 2} \mid \texttt{Youngest}\,\texttt{notif} = 2; \texttt{Remote}\,\texttt{clock} = 0}$$

The message contains two notifications, Notif 1 and Notif 2, and two timestamps that identify:

- the youngest transported notification: 2;
- the youngest notification previously received from B: $V_{M_1}(M_2) = 0$.

Notifications remain in the queue until the acknowledgment is received.

⑤ M_2 receives Msg 1:

- M_2 updates its view of M_1's local clock with the timestamp of the youngest notification of the message: $V_{M_2}(M_1) = 2$. Note that this view doesn't correspond to M_1's clock. It is only the occurrence date of the youngest notification received from M_1.
- M_2 removes the notifications that are acknowledged by the message from its waiting queue. In this case, there are none.
- M_2 processes all the notifications contained in the message: it adds url_1 and url_2 in its copy of the shared directory.

⑥ M_2's transport layer sends a message:

$$\mathsf{Msg\ 2} : \boxed{\mathsf{Notif\ 4} :|\ \texttt{Youngest notif} = 1; \texttt{Remote clock} = 2}$$

The message contains one notification: Notif 4, plus two timestamps indicating the youngest notification of this message, 1; and the youngest notification received from M_1, 2.

⑦ M_1 receives Msg 2:

- M_1 updates its view of M_2's local clock: $V_{M_1}(M_2) = 1$.
- M_1 removes notifications acknowledged by the message from its waiting queue. The only remaining notification in the queue is Notif 3.
- M_1 processes the notification contained in the message: it adds url_4 to its copy of the shared directory.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399