A Flexible and Efficient Protocol for Multi-Scope Service Registry Replication

Weibin Zhao *and* Henning Schulzrinne
Columbia University
New York, NY 10027
{zwb,hgs}@cs.columbia.edu

Abstract

Service registries play an important role in service discovery systems by accepting service registrations and answering service queries; they can serve a wide range of purposes, such as membership services, lookup services, and search services. To provide fault tolerant, and enhance scalability, availability and performance, service registries often need to be replicated. In this paper, we present Swift (Selective anti-entropy WIth FasT update propagation), a flexible and efficient protocol for multi-scope service registry replication. As consistency is a less of concern compared with availability in service registry replication, we choose to build Swift on top of anti-entropy to support high availability replication. Swift makes two contributions as follows. First, it defines a more general and flexible form of anti-entropy called selective anti-entropy, which extends the applicability of anti-entropy from full replication to partial replication by selectively reconciling inconsistent states between two replicas, and improves anti-entropy efficiency by fine controlling update propagation within each subset. Selective anti-entropy is the first that we are aware of in using anti-entropy to support generic partial replication. Secondly, Swift integrates service registry overlay networks with selective anti-entropy. Different topologies, such as full mesh and spanning tree, can be used for constructing service registry overlay networks. These overlay networks are used to propagate new updates quickly so as to minimize inconsistency among replicas. We have implemented Swift for replicating multi-scope Directory Agents in the Service Location Protocol. Our experience shows that Swift is flexible, efficient, and lightweight.

1 Introduction

Service registries play an important role in service discovery systems [8, 15, 23, 18] by accepting service registrations and answering service queries from clients. A client of a service registry can be a service provider or a service user (or both such as in a peer-to-peer discovery system [18]). Being used in different ways, service registries can serve for a wide range of purposes, such as (1) a membership service that maps a group identifier to the members in the group, for instance, the ENR server in Reliable Server Pooling [20]; (2) a lookup service that maps a service key to a service entry; and (3) a search service that maps a service description to one or multiple service keys. Note that a lookup service provides a one-to-one mapping, whereas a search service often results in a one-to-many mapping. Normally, service registries only maintain service information, real services are provided elsewhere. But service registries can also store service proxies which can be downloaded and be used to access services [23], or store service contents in which case a service registry serves as a content server as well. In summary, service registries are very flexible to support diverse application requirements.

To provide fault tolerant, and enhance scalability, availability and performance, service registries often need to be replicated. A simple way for service registry replication is to just deploy multiple service registries, and rely on clients to register with as many service registries as they can to improve the chances of being discovered. This simple approach is inefficient in that a client needs to keep track of all service registries so as to register with them. It is also hard to guarantee any consistency among service registries, which totally depends on how clients perform their registrations. A better way for service registry replication is to use a replication protocol among a set of service registries such that a client only needs to register with one service registry, then the registration will be propagated automatically to the rest service registries.

As service registries are used primarily for the query purpose, they should be available at any time. Compared with availability, consistency is a less of concern since inconsistent service registration states among replicas may only result in a less optimal service selection, but un-availability of service registries will lead to no service information at all. Therefore, an asynchronous [2] (or weakly consistent) replication should be used for service registries, where a registration update is delivered to one replica first, then it is propagated to other replicas later.

We choose to use anti-entropy [7] as the basic mechanism to build high availability service registry replication for three reasons. First, it ensures eventual consistency among a set of replicas. When there is no more updates, replicas are guaranteed to reach consistent states. Second, it is efficient in that only the differences among replicas are exchanged. Third, it is flexible as both the differences of states and the differences of updates can be exchanged. Normally, when two replicas perform anti-entropy between them for the first time, they exchange the differences of their initial states; in all subsequent anti-entropy sessions, they exchange their newly received updates.

However, two requirements in service registry replication are not well supported by anti-entropy, namely partial replication and fast update propagation. Anti-entropy was designed and used for supporting full replication; little work has been done in applying it to partial replication. Moreover, anti-entropy only guarantee eventual consistency; it dose not specify any mechanism to propagate updates quickly so as to minimize inconsistency among replicas. To remedy this situation, we designed Swift (Selective anti-entropy WIth FasT update propagation), a flexible and efficient replication protocol, which extends existing anti-entropy schemes [14, 19] in two important aspects: formulating a more generic form of anti-entropy to support partial replication, and efficiently integrating fast overlay propagation with anti-entropy.

The rest of this paper is organized as follows. We first overview full replication, partial replication, and anti-entropy in Section 2, and describe multi-scope service registry replication in Section 3. Then we identify the summary problem in applying anti-entropy to partial replication, and present a novel way to solve the summary problem by using selective anti-entropy in Section 4. Section 5 discusses how to use registry overlay network to propagate updates quickly, Section 6 describes other replication issues, including registration version resolution, registry membership service, and the

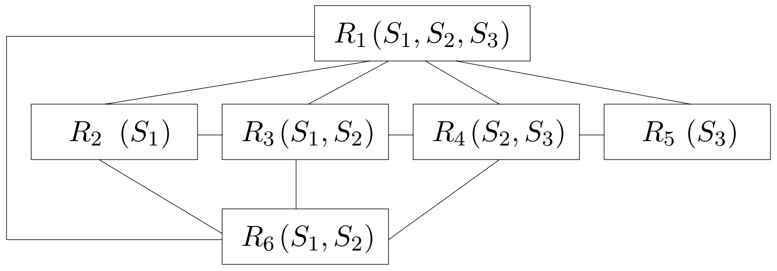


Figure 1: An example of $\mathcal{P}(6, 3)$, where replicas are R_1 to R_6, scopes are S_1 to S_3, and an edge between two replicas means that they share scopes.

use of soft states. We discuss implementation issues and Swift messages in Section 7, present performance study on selective anti-entropy in Section 8, list related work in Section 9, and conclude in Section 10.

2 Background

2.1 Full Replication vs. Partial Replication

Full Replication In a full replication system, all replicas maintain the same states. We denote a full replication system with r replicas as $\mathcal{F}(r)$.

Partial Replication In a partial replication system, the whole state set is divided into subsets referred to as *scopes*. We denote a partial replication system with r replicas and s scopes as $\mathcal{P}(r, s)$.

In $\mathcal{P}(r, s)$, a replica may support any number of scopes from 1 to s, and a scope of states may be replicated to any number of replicas from 2 to r. We assume that any state should be replicated to at least two replicas. In other words, the replication degree of any state is at least two. When all replicas support s scopes, or all states are replicated to r replicas, $\mathcal{P}(r, s)$ is equivalent to $\mathcal{F}(r)$. In this sense, we can view $\mathcal{F}(r)$ as a special form of $\mathcal{P}(r, s)$. $\mathcal{P}(r, s)$ is more flexible than $\mathcal{F}(r)$ in that different scopes may have different replication degrees, for instance, a popular scope may have a higher replication degree than a less popular scope. Figure 1 shows an example of $\mathcal{P}(6, 3)$.

In $\mathcal{F}(r)$, any two replicas are equivalent, but in $\mathcal{P}(r, s)$, four different types of relationships may exist between two replicas based on their shared scopes: equivalent, subset, overlap, and non-overlap; please refer to Table 1 for definitions and examples.

2.2 Overview of Anti-Entropy

The rational behind anti-entropy is simple. If each replica periodically reconciles its states with the rest replicas, all replicas will eventually become consistent when there is no further update occurred.

R_x vs. R_y	$S(R_x)$ vs. $S(R_y)$	Example in Figure 1
equivalent	$S(R_x) = S(R_y)$	R_3 vs. R_6
subset	$S(R_x) \subset S(R_y)$ or $S(R_y) \subset S(R_x)$	R_4 vs. R_1
overlap	$S(R_x) \cap S(R_y) \neq \phi$	R_3 vs. R_4
non-overlap	$S(R_x) \cap S(R_y) = \phi$	R_2 vs. R_5

Table 1: Relationships between replica R_x and R_y in $\mathcal{P}(r, s)$, where $S(R_x)$ and $S(R_y)$ denote the scopes of R_x and R_y, respectively.

In anti-entropy, two replicas only exchange their differences. To determine the differences between two replicas, an order is defined for all updates and states. An update sequence id (USID) in the form of (R, t) is assigned to each update, where R is the originating replica that accepts the update from the client, and t is the originating timestamp when the update is accepted at its originating replica. Note that all originating timestamps assigned by the same originating replica must be monotonically increasing. The USID of a state is the same as that of the last update applied to it. Two USIDs are comparable only if they have the same originating replica, and the order is determined by their originating timestamps.

New updates or states are always propagated in order of their USIDs, which enables two replicas to quickly find out their differences by comparing their *summary vectors*. A summary vector at a replica includes all the maximum USIDs it has received so far; for each unique originating replica in the received USIDs, there is a corresponding maximum USID in the summary vector. For example, if replica R_i has a summary vector of $((R_1, t_1), (R_2, t_2), ..., (R_r, t_r))$, then R_i has received all updates originally accepted by replica R_j up to timestamp t_j, where $1 \leq j \leq r$.

3 Multi-Scope Service Registry Replication

Unlike the hierarchical namespace and corresponding hierarchical arrangement of servers in DNS [17] and LDAP [22], service key often has a roughly flat structure (e.g., both Jini [23] and UDDI [8] use UUID [13] as service key), and a service registry can accept any registration in the whole service key namespace. To facilitate service management and search, services are often categorized into scopes, which are administrative groupings of services based on geographical locations, administrative departments, or some other categories. For example, SLP uses service scopes to group services in addition to service types, and UDDI provides comprehensive classifications for business web-based services, such as industry codes, product codes and geography codes – all these can be used for service groupings. The use of scopes effectively partitions a flat service space into sub-spaces, which provides an efficient, flexible, and scalable way to deploy service registries.

3.1 Multi-Scope Service Registries

There are three different ways to deploy service registries in multiple scopes. The first one is to have each service registry serve all scopes, which will lead to a coarse grain full replication. The

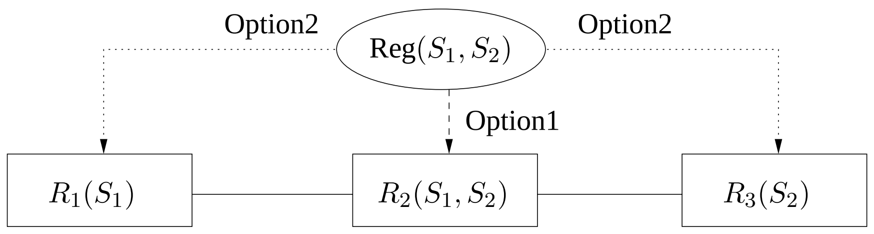


Figure 2: Options for sending a multi-scope service registration to registries

second one is to have each service registry serve a single scope, which will lead to a fine grain full replication. The third one is to allow a service registry to serve any number of scopes, from one to all scopes, which will lead to partial replication.

The first configuration is convenient for clients since each service registry is capable to answer any service query regardless of scopes. But this configuration is heavy-weight as each service registry needs to support all scopes. It is inefficient in some cases, for instance, when a popular scope needs to have a higher replication degree than a less popular one, and when a service registry serves clients that are only interested in some (not all) scopes.

The second configuration is neither efficient nor convenient. First, a multi-scope service registration needs to be sent to several single-scope registries each serving one of its scopes, whereas it only needs to be sent to a single multi-scope registry that serves all of its scopes. Secondly, for a search that targets multiple scopes or all scopes (using a scope wildcard), it needs to query multiple single-scope registries, whereas it only needs to query a multi-scope registry or a registry that serves all scopes. Finally, a multi-scope registry is more efficient than several single-scope registries in terms of resource usage.

The third configuration is the most flexible one, which can provide a better efficiency and convenience. In this configuration, two registries may share partial scopes, thus partial replication is needed. For example, assume that registry R_1 serves scopes S_1, S_2, and S_3, and registry R_2 serves scopes S_2, S_3, and S_4, then R_1 and R_2 need to replicate each other partially – only in their shared scopes S_2 and S_3.

3.2 Multi-Scope Service Registrations

A multi-scope service registration needs to be sent to a number of service registries such that the union of these service registries' scopes covers its scopes. For example, in Figure 2, a multi-scope service registration in scope S_1 and S_2 may be sent to registry R_2 (serving scope S_1 and S_2), or to both registry R_1 (serving scope S_1) and registry R_3 (serving scope R_2). Clearly, the first option (using R_2) is more convenient for the client. Note that in the second option, R_2 will receive two copies of *Reg* propagated from R_1 and R_3, respectively, and R_2 will install the first copy but ignore the second copy as they have the same version id (VID, see Section 6.1).

A multi-scope service registration should always carry its complete scope list whether it is sent from a client to a registry or propagated form a registry to another one. A service registry can accept

State	Scope	USID
a	S_1	(R_6, t_{61})
b	S_2	(R_6, t_{62})
c	S_3	(R_6, t_{63})

Table 2: The states and their scopes and USIDs at R_6, where $t_{61} < t_{62} < t_{63}$.

a multi-scope service registration only if it supports whole or partial of the scope list of that service registration. For example, if a service registration has a scope list of "S_1, S_2", then a service registry can accept this registration if its serving scope list comprises "S_1", or "S_2", or both. Furthermore, a registry propagates a service registration to another registry R_x only if R_x supports whole or partial of the scope list of that registration.

4 Selective Anti-Entropy

We refer to traditional anti-entropy [14, 19] as *complete anti-entropy*, where two replicas reconcile all inconsistent states between them in a two-way session or two one-way sessions. Complete anti-entropy works well in full replication, but may cause a summary problem – a replica cannot summarize its received updates using a summary vector, in partial replication.

4.1 Motivating Example

Let's use an example to show why complete anti-entropy is not sufficient for partial replication. In Figure 1, we assume that R_2 and R_3 have no state yet; R_6 has received three states a, b and c (their scopes and USIDs are shown in Table 2) from clients. Consider three sessions among R_2, R_3 and R_6 as follows:

- In the session between R_2 and R_6, R_2 gets new states a and c in scope S_1 from R_6, then R_2 changes its summary vector from () to $((R_6, t_{63}))$.
- In the session between R_3 and R_2, R_3 gets new states a and c in scope S_1 from R_2, then R_3 changes its summary vector from () to $((R_6, t_{63}))$. But the summary for R_6 is wrong since R_3 has not received b yet, which has a USID of (R_6, t_{62}).
- In the session between R_3 and R_6, R_3 wants to get new states in scope S_1 and S_2 from R_6, but since R_3 and R_6 have the same summary vector of $((R_6, t_{63}))$, R_6 will not send b to R_3. The anti-entropy fails due to the incorrect summary.

The reason leading to the wrong summary is that updates in different scopes are propagated separately, but may not in order of their USIDs. For example, R_3 receives state c before state b (which is not in order of their USIDs) since state c is in scope S_1, state b is in scope S_2, and R_3 receives updates in scope S_1 first.

Type	Number of Subsets	Direct/Indirect
select-one-direct	1	direct
select-one-indirect	1	indirect
select-multiple	≥ 2 but $< r-1$	indirect/mixed
select-all	$r-1$	mixed

Table 3: Types of selective anti-entropy sessions, where r is the number of replicas in the replica set.

There are two ways to fix this incorrect summary problem. One way is to use a summary matrix of size $r \times s$, which has a timestamp summary for each replica and scope combination. But this will significantly complicate the summary management, especially when there are many scopes, or/and when new scopes can be added dynamically. We believe a better and simpler approach is to keep the summary as a vector and use selective anti-entropy.

4.2 Selective Anti-Entropy

Selective anti-entropy is to selectively reconcile inconsistent states between two replicas in a session, i.e., a replica specifies the subsets of updates it solicits, and thus controls what it receives from another replica.

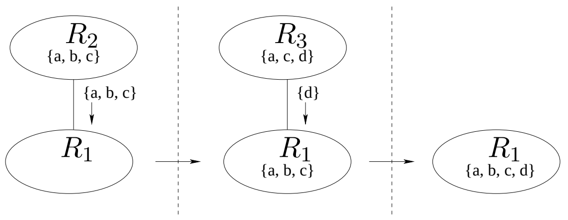
In selective anti-entropy, all updates that are originally accepted by the same replica R_x belong to the same subset $\overline{R_x}$. The subset of a state is the same as that of the last update applied to it. Thus, if there are r replicas accepting updates from clients, we can divide all updates and states into r subsets, denoted as $\overline{R_1}$, $\overline{R_2}$, ..., $\overline{R_r}$.

We use $\Theta(R_x, \{\overline{R_{x_1}}, \overline{R_{x_2}}, ..., \overline{R_{x_k}}\}, R_z)$ to denote a selective anti-entropy session where R_x requests new updates in $\overline{R_{x_1}}$, $\overline{R_{x_2}}$, ..., $\overline{R_{x_k}}$ from R_z. All selective anti-entropy sessions can be categorized into four types (listed in Table 3) based on the number of subsets requested and whether the session is direct or indirect. We assume that a replica R_x does not request updates in $\overline{R_x}$ from other replicas. A session is direct if it only requests updates in $\overline{R_z}$ from R_z (e.g., $\Theta(R_x, \{\overline{R_z}\}, R_z)$), otherwise the session is indirect (e.g., $\Theta(R_x, \{\overline{R_y}\}, R_z)$) or mixed (e.g., $\Theta(R_x, \{\overline{R_y}, \overline{R_z}\}, R_z)$). A select-multiple or select-all session can be viewed as comprising k ($k \geq 2$) select-one sessions, in which at most one is select-one-direct session, and the rest are select-one-indirect sessions. A select-all session is equivalent to a complete anti-entropy session.

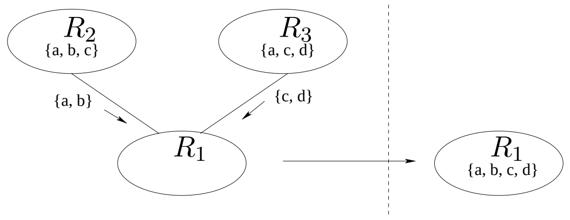
Figure 3 shows how selective anti-entropy differs from complete anti-entropy. Figure 3(a) gives the initial states at replica R_1, R_2 and R_3: R_1 has no state yet; R_2 has received two states a and b from clients, and has propagated a to R_3; and R_3 has received two states c and d from clients, and has propagated c to R_2. Figure 3(b) and Figure 3(c) illustrate how R_1 gets new states from R_2 and R_3 via complete and select-one-direct anti-entropy, respectively.

R_1	R_2		R_3	
	State	USID	State	S_{id}
ϕ	a	(R_2, t_{21})	a	(R_2, t_{21})
	b	(R_2, t_{22})	c	(R_3, t_{31})
	c	(R_3, t_{31})	d	(R_3, t_{32})

(a) The states and their USIDs at R_1, R_2 and R_3



(b) R_1 performs complete anti-entropy with R_2 and R_3



(c) R_1 performs select-one-direct anti-entropy with R_2 and R_3

Figure 3: Complete anti-entropy vs. Select-one-direct anti-entropy

4.3 Safe Anti-entropy Sessions

An anti-entropy session is *safe* if it will not cause incorrect summaries at the summary vector. Using selective anti-entropy, a replica can choose safe sessions in all cases. The following selective anti-entropy sessions are safe.

- Any select-one-direct anti-entropy session, such as $\Theta(R_x, \{\overline{R_z}\}, R_z)$, is safe.
- A select-one-indirect anti-entropy session $\Theta(R_x, \{\overline{R_y}\}, R_z)$ is safe if $S(R_x) \cap S(R_y) \subseteq S(R_z)$, where $S(R_x)$, $S(R_y)$ and $S(R_z)$ denote the scopes of R_x, R_y and R_z, respectively. In other words, if $S(R_x) \cap S(R_y) \supset S(R_z)$, $\Theta(R_x, \{\overline{R_y}\}, R_z)$ may cause incorrect summaries. In Figure 1, for example, $\Theta(R_3, \{R_1\}, R_6)$ is safe because $S(R_3) \cap S(R_1) = \{S_1, S_2\} = S(R_6)$, but $\Theta(R_3, \{R_6\}, R_2)$ may not be safe since $S(R_3) \cap S(R_6) = \{S_1, S_2\} \supset S(R_2) = \{S_1\}$.
- A select-multiple or select-all anti-entropy session is safe if each of its select-one-indirect sessions is safe.

In contrast, a replica cannot guarantee a complete anti-entropy session to be safe in some cases. For example, consider two complete anti-entropy sessions, s_1 and s_2, in Figure 1: s_1 is between R_3 and R_6, s_2 is between R_3 and R_2, and s_1 is performed before s_2. R_3 cannot guarantee s_2 to be safe since between s_1 and s_2, R_2 may perform another session with R_6 and get new updates in $\overline{R_6}$.

Message Type	Anti−Entropy Type	Number of Subset Entries
Subset Entry 1	$\cdots$	Subset Entry k

(a) The message format

Originating Replica ID	Lower Bound Timestamp	Upper Bound Timestamp

(b) The subset entry format

Figure 4: The anti-entropy request message

4.4 Parallel Anti-entropy Sessions

A replica needs to perform complete anti-entropy sessions sequentially, but it can perform multiple selective anti-entropy sessions in parallel, such as all select-one-direct sessions. In general, R_x can perform k selective anti-entropy sessions ($s_1, s_2, ..., s_k$) in parallel if and only if the requested update subsets ($u_1, u_2, ..., u_k$) in these sessions do not overlap, i.e., $u_1 \cap u_2 \cap ... \cap u_k = \phi$. For example, assume $k = 2$, $u_1 = \{\overline{R_2}, \overline{R_4}\}$, $u_2 = \{\overline{R_3}, \overline{R_5}\}$, then session s_1 and s_2 can be performed in parallel because $u_1 \cap u_2 = \phi$.

Running anti-entropy sessions in parallel at a replica can improve performance (see Section 8) since the replica does not need to wait until a session is finished before it can start another session.

4.5 A Generic Message Format for Anti-entropy Requests

While selective anti-entropy can avoid incorrect summaries, complete anti-entropy is simple and works well for full replication. Thus, it is advantageous to support both selective and complete anti-entropy sessions in one system. This can be achieved by using a generic message format for anti-entropy requests, shown in Figure 4(a). This message has four components: message type, anti-entropy type, number of subset entries, and subset entries. If the anti-entropy type is selective, then selective anti-entropy is performed, in which only new updates that are in the specified subsets are requested; otherwise complete anti-entropy is performed, in which both new updates that are in the specified subsets and all updates that are not in the specified subsets are requested. Clearly, when the same subset entries are used in an anti-entropy request, the selective anti-entropy type may request less updates than the non-selective one.

In complete anti-entropy, each subset entry has two components: replica id and lower bound timestamp. An unspecified subset has a default lower bound timestamp as zero. To facilitate further fine selection of new updates within each subset, the subset entry in Swift has three component: replica id, lower bound timestamp, and upper bound timestamp (Figure 4(b)). This way an exact range of new updates (greater than lower bound but less than upper bound) within each subset can be specified, which is useful in some cases, for instance, when fast overlay propagation is used along with anti-entropy (see Section 5). Note that an upper bound timestamp is ignored if it is zero.

5 Service Registry Overlay Networks

5.1 Motivations

Normally, anti-entropy sessions are scheduled periodically (e.g., once a day) among all replicas, where new updates are transferred in batch form one replica to another. As a state may remain inconsistent between two replicas for as long as the anti-entropy interval, it would be beneficial to have a way to propagate updates quickly between anti-entropy sessions in order to minimize inconsistency among replicas.

One way to propagate updates quickly is to use multicast: when the originating replica receives an update from a client, it multicasts the update to other replicas. However, this simple approach has two limitations. First, replicated service registries may be deployed across the whole Internet, but multicast is not readily available in the Internet. Secondly, multicast may be unreliable, and does not guarantee any order on message arrivals. In contrast, anti-entropy transfers messages reliably and in order. These mismatches make the integration of multicast with anti-entropy difficult.

A better way to propagate updates quickly is to use service registry overlay networks at application layer. An overlay network is a connected graph: each node is a service registry, and each edge is a connection that provides reliable and in order message transfer, such as TCP connection. Different overlay network topologies have different update propagation controls; we will discuss this issue in section 5.5. As both anti-entropy and overlay networks use connections to transfer messages reliably and in order, it is easier to integrate them together. Note that anti-entropy and overlay networks differ in two aspects in using connections. First, the connections in overlay networks are persistent, though they may fail due to network partitions and registry failures, whereas the connections used by anti-entropy sessions are created and torn down on demand. Secondly, overlay networks may use different topologies, such as full mesh, spanning tree, and ring, whereas a replica normally performs anti-entropy with other replicas in a full mesh style. Nevertheless, if there is an overlay connection existed between two registries, they shall use this connection in anti-entropy instead of creating another one.

5.2 Fast Overlay Propagation vs. Anti-Entropy

Propagating updates quickly via service registry overlay networks is referred to as fast overlay propagation. When fast overlay propagation is used, anti-entropy does not need to be performed periodically. Instead, anti-entropy becomes a backup mechanism, which is used only for exchanging initial states and for catching up new updates after failures. In normal condition where there is no network partition, no registry failure, and no new registry join, fast overlay propagation is sufficient to distribute any new update from its originating replica to the rest replicas.

Should fast overlay propagation fail, a replica needs a way to detect this failure. For example, if replica R_x has not received any update originated from replica R_y for a long time, R_x needs to distinguish a failure (e.g., R_y failure or network partition) from no update at all originated form R_y. To achieve this, each replica periodically sends a keepalive message via overlay networks. If a replica's keepalive message has been timeout, then some failure has occurred, and anti-entropy needs to be scheduled.

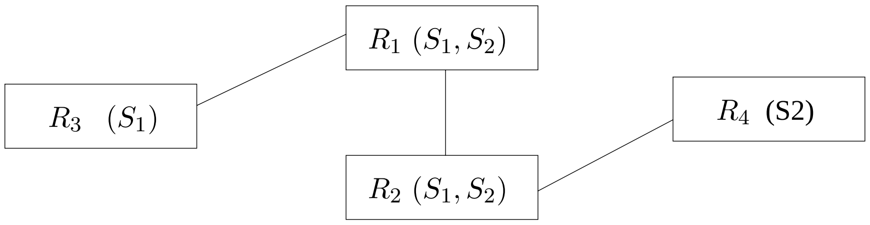


Figure 5: An example of service registry overlay networks for two scopes S_1 and S_2: each scope has its own overlay network, but they share one connection that is between registry R_1 and R_2.

5.3 Overlay Networks for Multiple Scopes

Service registry overlay networks are formed on scope basis, i.e., each scope has its own registry overlay network. However, the overlay networks for two different scopes may share one or more connections. For example, Figure 5 shows the overlay networks for scope S_1 and S_2; they share one connection that is between registry R_1 and R_2. In general, only one overlay network connection is needed between any two replicas regardless of how many scopes they share. In other words, if a connection is between two replicas that share multiple scopes, then this connection is shared by multiple overlay networks.

5.4 Detecting Missed Updates

A replica sends new updates in order of their USIDs in both fast overlay propagation and anti-entropy. However, the policies for sending new updates in these two cases are different: pull[1] in anti-entropy, whereas push in fast overlay propagation. Different sending policies at the sender side lead to different requirements on detecting missed updates at the receiver side. When an update is received via fast overlay propagation, a mechanism is needed to detect missed updates. This is because when a replica newly joins (or re-joins) an overlay network, it will begin to receive new updates via fast overlay propagation right away, but it may have missed some early updates that have a USID less than that of the current update. For anti-entropy, in contrast, the detection of missed updates is not needed: when a replica receives an update with a USID as x, it can assume that it has received all updates with a USID less than x (i.e., no update is missed). This is because (1) new updates are sent from the right starting points specified by an anti-entropy request[2], and (2) updates are sent reliably and in order.

As described in Section 2.2, each update carries its USID, which has two components: originating replica, and originating timestamp. All originating timestamps assigned by the same originating replica are monotonically increasing, but their values are not necessarily increasing by exactly one. Thus, from an update's USID, a receiving replica cannot tell whether it has missed some early updates. To remedy this situation, we let each update carry both its USID and its preceding USID so

[1]A replica sends new updates to another replica only when it receives an anti-entropy request from that replica.

[2]The starting point of requested updates in a subset is specified via the lower bound timestamp in the subset entry, see Figure4(b). If multiple subsets are specified in an anti-entropy, there will be multiple starting points.

Topology	Full Mesh	Spanning Tree	Other Topologies
Propagation Control	One hop	Forwarding from the incoming connection to the rest connections	Detecting duplicate updates, and forwarding non-duplicate updates from the incoming connection to the rest connections
Advantages	Simple, fast	Scalable	Balanced scalability and robustness
Disadvantages	Poor scalability	Poor robustness	A replica may receive duplicate updates

Table 4: Comparison of Different Overlay Network Topologies

that when a replica receives an update having x and y as its USID and preceding USID, respectively, the replica can ensure that it does not miss any update if it has already received an update having y as its USID. Since an update's USID and preceding USID have the same originating replica, we represent them in a compact format (R, t, t_p) referred to as EUSID (extended USID), where R is the originating replica, t is the originating timestamp, and t_p is the preceding originating timestamp. In an EUSID (R, t, t_p), t_p is less than t except when the EUSID is the first one assigned by R, where t_p equals t.

There are two advantages to use EUSID rather than USID. First, EUSID provides a way to detect missed updates. In other words, EUSID can be used to determine whether a received update is in order. An update is delivered and the summary vector is changed only when the update is in order, otherwise the update is buffered for later processing. Secondly, EUSID provides a way to trigger anti-entropy in addition to periodic scheduling. Should missed updated are detected, a replica needs to send an anti-entropy request to the corresponding originating replica to solicit missed updates. Here a range of updates (with lower and upper bound timestamps) needs to be specified, for which the subset entry format shown in Figure 4(b) is very useful.

5.5 Topologies and Propagation Controls

Different overlay network topologies have different update propagation controls. Table 4 compares the advantages and disadvantages of different topologies. Note that all replicas in the same overlay network must agree on using the same topology.

5.5.1 Full Mesh Topology

The full mesh topology is simple. It provides the quickest way to propagate new updates: when a replica receives an update from a client, the update only needs to be propagated one hop[3] to reach the rest replicas. Also, a full mesh overlay network can be used for anti-entropy. However, the full

[3]The hop used here is the overlay network hop from one replica to another; it is not the network routing hop.

mesh topology has poor scalability; it is only suitable to a small number of replicas in an overlay network.

The creation of a full mesh topology is easy: a replica just needs to set up a connection to each of the rest replicas. The membership information about replicas in each scope can be obtained using the techniques described in Section 6.2.

The update propagation in a full mesh overlay network is performed as follows. When a replica receives an update from a client, it assigns an EUSID to the update, then sends the update to the rest replicas. When a replica receives an update from another replica, no further propagation for the update is needed.

5.5.2 Spanning Tree Topology

The spanning tree topology is scalable, but less robust in terms of connectivity. The creation of a spanning tree topology has been investigated in a number of systems, such as INS resolver network [3] and VIA gateway spanning tree [5].

The update propagation in a spanning tree overlay network is performed as follows. When a replica receives an update from a client, it assigns an EUSID to the update, then sends the update to all overlay network connections its has. When a replica receives an update from another replica via an overlay network connection, it forwards the update to the rest[4] overlay network connections it has. Since there is no loop in the spanning tree topology, the propagation of an update will stop when the update reaches all replicas.

5.5.3 Other Topologies

The full mesh topology has the maximum connections, whereas the spanning tree topology has the minimum connections. Other connected topologies, such as ring, have neither the maximum nor the minimum connections.

The creation of an arbitrary connected topology can be done via configurations. Its update propagation control is similar to that of the spanning tree topology, but with duplicate detection. When a replica receives an update from a client, it assigns an EUSID to the update, then sends the update to all overlay network connections its has. When a replica receives an update from another replica via an overlay network connection, if it has not received the update before, it forwards the update to the rest overlay network connections it has; otherwise, the update is discarded.

6 Other Issues

6.1 Registration Version Resolutions

Swift supports two types of service registrations: fresh registration and incremental registration. In a fresh registration, a client provides complete information about a service aiming to create a new registration state or overwrite an existing registration state, whereas in an incremental registration,

[4]Except the one from which the update is received.

a client only gives partial information about a service aiming to update an existing registration state. The rule for performing these two types of service registrations is as follows. When a client uses a new service registry to register services, first it needs to perform a fresh registration for each service, then it can perform follow-up incremental registrations at the same service registry.

When service registrations are propagated among replicas[5], their arrival timestamps at a replica cannot be used for registration version resolutions. Considering two fresh registrations for the same service, the one that arrivals later does not necessarily mean it is a new version. For example, if a client sends a registration (v_1) to replica R_1 first, and a new version of the same registration (v_2) to replica R_2 later, then when v_1 and v_2 are propagated, the arrival timestamp of v_1 at R_2 is later than that of v_2, but v_1 should not overwrite v_2 at R_2 as v_2 is a newer version.

We assume that all registrations for the same service are issued by the same client. In Swift, registration versions are resolved as follows. At the client side, each client assigns a version id (VID) in the form of (C, t_v) to each registration that it issues, where C is the originating client's id (e.g., UUID), which issues the registration, and t_v is the version timestamp. Note that all version timestamps assigned by the same originating client must be monotonically increasing. At the service registry side, a registry accepts a registration if (1) the registration has a newer version timestamp and the same originating client id as that of the corresponding registration state; or (2) it is a fresh registration, and there is no corresponding registration state existed.

6.2 Membership Service

To support replication, a registry needs to know other registries in its scopes. One way to achieve this is to deploy a super registry which maintain registry membership information about each scope [3]. Here we describe another approach to provide membership service via multicasting advertisement and exchanging peer list. If two registries share scopes, they are peer registries. A registry periodically multicasts its advertisement, which indicates its scopes and other information.

When two peer registries know each other for the first time, they exchange their advertisements and peer lists. This enables a registry to learn about its peers incrementally from its known peers. The interactions between registry R_1 and R_2 are as follows: (1) R_1 learns about R_2 for the first time, R_1 sends its advertisement to R_2; (2) R_2 replies with its advertisement to R_1; (3) R_1 sends its peer list to R_2; and (4) R_2 replies with its peer list to R_1. Note that the peer list message (shown in Figure 6) carries a list of advertisement entries, one advertisement entry for each peer. Each peer has two type flags indicating that it is an active registry[6], or an originating registry[7], or both. The peer list type field indicates that the peer list is complete or partial. Normally R_1 sends a complete common peer list to R_2, whereas R_2 only replies with a partial common peer list to R_1, which indicates the peers that R_2 knows but R_1 does not know.

[5]Note that a registration state is propagated via a fresh registration

[6]A registry is an active registry if its keepalive message has not been timeout.

[7]R_x is an originating registry for R_1 if R_1 maintains registrations that are originated from R_x. This information about originating registries is useful for selective anti-entropy.

Message Type	Peer List Type	Number of Advertisement Entries
Advertisement Entry 1	$\cdots$	Advertisement Entry k

(a) The message format

Active Flag	Originating Flag	Advertisement

(b) The advertisement entry format

Figure 6: The peer list message

6.3 Soft State

Service registries [15, 23, 25] often maintain registrations as soft states [6], which have a lifetime, and will expire unless it is refreshed periodically. A registration can be deregistered, but this is not required since an expired registration will be removed from the service registry automatically. The soft state mechanism provides a good support for adapting to dynamically changed service environments gracefully. Note that when a soft state registration is propagated, it carries its remaining lifetime.

7 Implementation Overview

7.1 Control Data Structures for Each Registration

Each registration has two control data structures: EUSID and VID. EUSID $= (R, t, t_p)$ (see Section 5.4) is assigned by the originating replica R that accepts the registration from the client. VID $= (C, t_v)$ (see Section 6.1) is assigned by the originating client C that issues the registration. When a registration is issued by a client, it only carries its VID, but when registration is propagated from one registry to another, it carries both its VID and its EUSID.

7.2 Control Data Structure for All Registrations

Each registry maintains a summary vector (see Section 2.2) to summarize all registrations that it has received. A registry consults its summary vector in detecting duplicate and missed registrations, and constructing anti-entropy requests. When an in-order registration is accepted at a registry, the registry updates it summary vector.

7.3 Control Data Structures for Peer Registries

Peer registries share scopes. Each registry maintain two control data structures for its peer registries: overlay connection table, and peer advertisement table. The overlay connection table keeps information about all overlay network connections that a registry has. A registry consults this table when it performs fast overlay propagation, and when it performs anti-entropy with directly connected peer registries. The peer advertisement table contains advertisement information about all peer registries. A peer registry can be an active registry, or an originating registry, or both (see Section 6.2).

7.4 Control Data Structure for Overlay Topology

The administrator of a replication system chooses a topology for each overlay network. Swift supports three topologies: full mesh, spanning tree, and other. Different topologies have different fast overlay propagation controls (see Table 4): the full mesh topology uses one-hop propagation by distinguishing a registration sent from a client (only has VID) from a registration propagated from a registry (has both VID and EUSID); the spanning tree topology utilizes the overlay connection table to propagate updates; and any other topology uses the summary vector to detect duplicate and missed updates, and use the overlay connection table to forward updates.

7.5 Swift Messages

The Swift protocol employs four messages and two message extensions. The four messages are advertisement, peer list, anti-entropy request and anti-entropy reply. The two message extensions are VID and EUSID. The advertisement message carries registry ID, scope list, boot timestamp, authentication information, and other optional attributes. The format of the peer list message is shown in Figure 6. The format of the anti-entropy request message is shown in Figure 4. The anti-entropy reply has the same format as that of the anti-entropy request message, except with a different message type; it is used to indicate a corresponding anti-entropy request has been completed.

Swift is designed as a lightweight server-to-server protocol, which can be used to provide server replication for existing client-server registry protocols. For example, we have implemented Swift for the Mesh-enhanced Service Location Protocol [26]; the source code can be found at [10]. Currently our prototype implementation supports all Swift messages and extensions, but only supports the full mesh topology. We will add the support for spanning tree and other topologies soon.

8 Performance Study for Selective Anti-entropy

Since a select-all anti-entropy session is equivalent to a complete anti-entropy session (see Section 4.2), selective anti-entropy can be regarded as a generalization of complete anti-entropy. Although the main goal of using selective anti-entropy is to solve the summary problem in partial replication efficiently rather than for pure performance purpose, selective anti-entropy can achieve equivalent or better performance than that of complete anti-entropy.

8.1 Performance Analysis

Two main factors affect the anti-entropy performance: connection properties (such as bandwidth, delay and loss rate), and replica failures. When connections differ among replicas, a replica should properly arrange the order of its sequential anti-entropy sessions so that it can get most updates from

Number of Registries (r)	3	6	10
Registrations at each of the r registries	15005	30005	50005
Registrations the new registry will have	15015	30030	50050

Table 5: Number of registrations at each of r registries and new registry

replicas that are close to it. Replica failures need to be considered as select-one-direct anti-entropy cannot be performed with failed replicas.

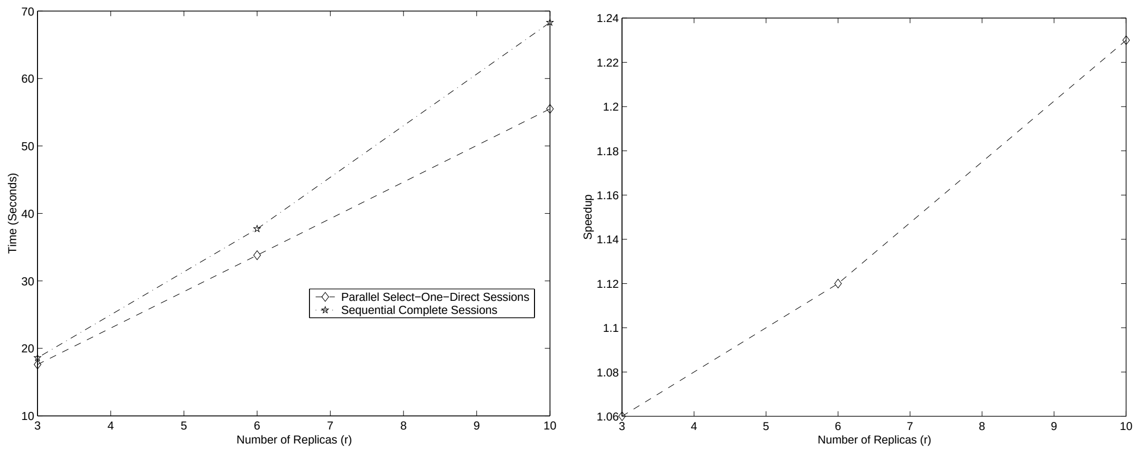
For the simple case where all connections are roughly equivalent (this is typical in the LAN environment) and there is no replica failure, parallel select-one-direct sessions can perform better than sequential complete sessions because (1) the setup time of these sessions can overlap; (2) any replica only needs to check new updates originated from itself rather than from $r - 1$ replicas (assuming total r replicas); and (3) if a replica has sufficient in-bound bandwidth and processing power, it can receive data at a higher speed in parallel sessions than that in sequential sessions. For other cases where connections differ, or/and there are replica failures, selective anti-entropy can operate similarly as complete anti-entropy by using sequential sessions and arranging the order of these sessions properly, and achieve equivalent performance.

8.2 Experiments

To validate that parallel select-one-direct sessions can perform better than sequential complete sessions, we set up a testbed in our department workstation cluster: all machines are Sun Ultra Sparc (Ultra-1, Ultra-2 or Ultra-10) workstations running Solaris 5.7; and they are connected via 10Mb/s Ethernet. We chose r machines as existing registries, one machine as new registry, and one machine as client. The client feeds 5005 different registrations to each of the r registries; and each registry propagates 5000 registrations originated from it to the rest registries. Thus each registry has $5000 * r + 5$ registrations. Considering the new registry join, it gets registrations from the existing registries. Table 5 lists the number of registrations at each registry for different r values. Figure 7(a) shows the times that the new registry needs to accomplish anti-entropy with r registries by using parallel select-one-direct sessions and sequential complete sessions. We did each test three times, and use the average of their values. To further illustrate that parallel select-one-direct sessions perform better than sequential complete sessions, we define speedup as the ratio of the selective anti-entropy running time over that of complete anti-entropy, and show speedups in Figure 7(b). For instance, the speedup is 1.23 for $r = 10$.

9 Related Work

To our knowledge, Swift is the first anti-entropy system that supports generic partial replication, and use application layer overlay networks for fast update propagation. Swift builds on previous work on anti-entropy and overlay networks; it is also related to reliable multicast.



(a) Time (b) Speedup

Figure 7: Parallel select-one-direct sessions vs. sequential complete sessions

9.1 Anti-entropy

Using anti-entropy for high availability replication was first described in [7]. Since then, much work has been done to enhance this mechanism. The time-stamped anti-entropy (TSAE [14]) proposed the summary vector technique, and suggested using multicast to propagate updates quickly. Bayou [19] enhanced anti-entropy flexibility by using uni-direction pair-wise sessions. Swift is closely related to Bayou and TSAE, but provides two further enhancements: generic partial replication support and fast overlay propagation. A recent replication system that employs anti-entropy is UDDI [8], which is a registry framework for description, discovery and integration of web services using XML [9] and SOAP [1]. Although UDDI only supports full replication, two similarities exist between UDDI and Swift: (1) Swift uses overlay networks to propagate updates quickly, whereas UDDI uses a communication graph (e.g., a spinning circle) to perform anti-entropy; and (2) the response-limit vector and changes-already-seen vector in UDDI is roughly equivalent to the subset entry with lower and upper bounds in Swift. Note that our work on selective anti-entropy is described briefly in [27].

9.2 Overlay Networks

Overlay networks have been used for various purposes, including M-bone [11] for multicast, 6-bone [12] for IPv6, X-Bone [24] for automated overlay network deployment, and resilient overlay network (RON) [4] for fault detection and recovery. The use of overlay networks in Swift is to propagate update quickly, and thus improve replication consistency. We focus on update propagation controls in arbitrary connected topology. For topology creation, we present an automated way for full mesh, use existing work (e.g., INS [3] and VIA [5]) for spanning tree, and base on configuration for other topologies. Note that multi-scope registry overlay networks in Swift is similar to multiple concurrent overlays in X-Bone.

9.3 Reliable Multicast

Swift uses overlay networks to propagate an update from its originating replica to the rest replicas, which achieves the same effect as that of reliable multicast [16, 21]. In Swift, a receiver is responsible to detect duplicate and missed updates (via EUSID and summary vector) and use anti-entropy to retrieve missed updates from their originating replicas. This bears some similarity to the negative acknowledgment (NACK) based reliable multicast protocols.

10 Conclusions

In this paper, we described Swift, a flexible and efficient protocol for multi-scope service registry replication. Swift makes two contributions: using selective anti-entropy to support generic partial replication, and employing overlay networks to propagate update quickly. Although Swift is discussed in the context of service registry replication, it can be applied to other applications which need partial replication and fast update propagation. For future work, we will investigate the effect of different overlay topologies on performance and reliability, in addition to our current focus on update propagation controls. Another issue we plan to explore is to use bulk data transfer in anti-entropy, which is to encode multiple updates into one message aiming to improve efficiency when lots of updates need to be transferred.

References

[1] Simple Object Access Protocol (SOAP) 1.1. http://www.w3.org/tr/soap/.

[2] ACM. Communications of the ACM 39(4), special issue on group communications systems, April 1996.

[3] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proc. ACM Symposium on Operating Systems Principles*, December 1999.

[4] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *The 18th ACM Symposium on operating systems principles*, Chateau Lake Louise, Banff, Canada, October 2001.

[5] P. Castro, B. Greenstein, R. Muntz, C. Bisdikian, P. Kermani, and M. papadopouli. Locating application data across service discovery domains. In *ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)*, July 2001.

[6] David D. Clark. The design philosophy of the DARPA internet protocols. In *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 106–114, Stanford, California, August 1988. ACM. also in *Computer Communication Review* 18 (4), Aug. 1988.

[7] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *The Sixth Annual ACM Symposium on Principles of distributed computing*, pages 1–12, Vancouver, Canada, August 1987.

[8] Universal Description Discovery and Integration. http://www.uddi.org/.

[9] Extensible Markup Language (XML) 1.0 (Second Edition). http://www.w3.org/tr/rec-xml.

[10] Service Location Protocol Enhancements. http://www.cs.columbia.edu/˜zwb/project/slp.

[11] H. Eriksson. Mbone: The multicast backbone. *Communications ACM*, 37(8):54–60, August 1994.

[12] Testbed for deployment of IPv6. http://www.6bone.net/.

[13] ISO International Organization for Standardization). ISO/IEC 11578:1996. Information technology – open systems interconnection – remote procedure call (RPC).

[14] Richard Golding. *Weak-consistemcy group communication and membership.* PhD thesis, Computer Science, University of California at Santa Cruz, December 1992.

[15] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service location protocol, version 2. RFC 2608, Internet Engineering Task Force, June 1999.

[16] B. N. Levine and J. J. Garcia luna Aceves. A comparison of reliable multicast protocols. *Multimedia Systems*, 6(5):334–348, September 1998.

[17] P. V. Mockapetris. Domain names - concepts and facilities. RFC 1034, Internet Engineering Task Force, November 1987.

[18] Napster. http://www.napster.com.

[19] Karin Petersen, Mike J. Spreizer, Douglas B. Terry, Marvin M. T heimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *The Sixteenth ACM Symposium on operating systems principles*, pages 288–301, Saint Malo, France, October 1997.

[20] Reliable Server Pooling. http://www.ietf.org/html.charters/rserpool-charter.html.

[21] Reliable Multicast Transport. http://www.ietf.org/html.charters/rmt-charter.html.

[22] M. Wahl, T. Howes, and S. Kille. Lightweight directory access protocol (v3). RFC 2251, Internet Engineering Task Force, December 1997.

[23] Jim Waldo. The Jini architecture for network-centric computing. *Communications ACM*, 42(7):76–82, July 1999.

[24] The X-Bone. http://www.isi.edu/x-bone/.

[25] Y. Yaacovi, M. Wahl, and T. Genovese. Lightweight directory access protocol (v3): Extensions for dynamic directory services. RFC 2589, Internet Engineering Task Force, May 1999.

[26] W. Zhao, H. Schulzrinne, and E. Guttman. Mesh-enhanced service location protocol. Internet Draft, Internet Engineering Task Force, November 2001. Work in progress.

[27] Weibin Zhao and Henning Schulzrinne. Selective anti-entropy. In *The Twenty-First ACM Symposium on Principles of Distributed Computing (PODC'02)*, Monterey, California, July 2002.